The Atomic Rules Approach for Describing Black-Box Testing Methods and its Evaluation

Submitted by
Tafline Murnane, BEng (Hons)

A thesis submitted in total fulfilment

of the requirements for the degree of

Doctor of Philosophy

School of Engineering and Mathematical Sciences

Faculty of Science, Technology and Engineering

La Trobe University
Bundoora, Victoria 3086
Australia

October 2009

Table of Contents

List of Tables

List of Figures

Abstract

Ideally, any black-box testing method would be interpreted in the same way by different testers, such that when it is applied to a program specification, it results in an identical test set, regardless of each tester's domain knowledge and experience. Further, each method should be complete and lead to the generation of all possible test cases that are derivable by the method for each specification. In reality, inconsistencies, ambiguities and a lack of precision in existing definitions of methods like Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) can lead to differing interpretations and thus varying test set quality. The absence of precise definitions also makes verification of test quality and conformance to method guidelines difficult.

Furthermore, while prescriptive methods like EP and BVA can be used effectively by experienced testers for defect detection, evidence suggests that non-prescriptive approaches like Exploratory Testing can allow testers to detect more defects. Domain knowledge utilised during this process often cannot be shared or reused since the approaches lack procedures for capturing this information. Black-box testing effectiveness can also depend on the quality of program specifications, since incomplete or ambiguous specifications can lead to inadequate testing.

In this thesis, the 'Atomic Rules' approach is introduced to provide a prescriptive notation for black-box testing methods, to resolve their ambiguities and improve their usability and effectiveness. This approach decomposes each method into test selection rules that cover partition selection, test data selection, test data manipulation and test case construction. Each Atomic Rule is represented in a characterisation schema, allowing them to be prescriptively defined, while a four-step test case design process provides the methods with a uniform notation.

Three experiments were conducted to validate this approach. The outcomes suggest that Atomic Rules is effective for teaching black-box testing methods to novice and experienced testers and can enable more effective testing.

A customisation approach called Systematic Method Tailoring (SMT) allows new Atomic Rules to be defined and supports domain knowledge capture during non-prescriptive testing. A specification technique called Goal/Question/Answer/Specify/Verify (GQASV) facilitates definition of precise input/output data specifications, enabling more effective testing. A prototype testing tool demonstrates automation of Atomic Rules, SMT and GQASV.

Statement of Authorship

Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis submitted for the award of any other degree or diploma.

No other person's work has been used without due acknowledgement in the main text of the thesis.

The thesis has not been submitted for the award of any degree of diploma in any other tertiary institution.

All experimental research reported in the thesis was approved by the human ethics committee of the Faculty of Science, Technology and Engineering at La Trobe University.

Signed: _____ Date: _____

Tafline Murnane, BEng (Hons)

Glossary

The following terms and abbreviations have been used throughout this thesis.

Table 1: Terms and abbreviations.

Term / Abbreviation	Meaning
Acceptance Testing	A level of test conducted from the viewpoint of the user or customer, used to establish criteria for acceptance of a system. Typically based upon the requirements of the system. (Craig & Jaskiel 2002)
Ad Hoc Testing	Testing carried out using no recognised test case design technique. (BS 7925-1)
American National Standards Institute (ANSI)	The American National Standards Institute.
Atomic Rule	One individual test case design rule from a test case design method that can be used to design an equivalence class, select a test data value, manipulate/mutate a test data value or contract a test case.
Backus-Naur Form (BNF)	A metalanguage for specifying computer language syntax. (Rosen & Michaels 2000)
Black-Box Testing	See *Black-Box Testing Method*.
Black-Box Testing Method	Test case design that is based on an analysis of the specification of the system under test without reference to its internal workings. (adapted from (BS 7925-1))
Boundary Value Analysis (BVA)	A test case design method in which test cases are designed to cover the boundary values of a component or system (adapted from (BS 7925-1))
British Standards Institution (BSI)	The British Standards Institution.
Compatibility Testing	Testing whether the system is compatible with other systems with which it should communicate. (BS 7925-1)
Component	A minimal software item for which a separate specification is available. (BS 7925-1)
Coverage	A metric that describes how much of a system has been (or will be) invoked by a test set. Coverage is typically based upon the code, design, requirements, or inventories. (Craig & Jaskiel 2002)
Data-Item Selection Rule (DISR)	One Atomic Rule from a test case design method that describes how to select a test data value from an equivalence class.
Data-Item Manipulation Rule (DIMR)	One Atomic Rule from a test case design method that describes how to manipulate or mutate a test data value.
Data-Set Selection Rule (DSSR)	One Atomic Rule from a test case design method that describes how to select an equivalence class.

Term / Abbreviation	Meaning
Decision Tables (DT)	Tables that list all possible conditions (inputs) and all possible actions (outputs). (Craig & Jaskiel 2002)
Defect	See *Fault*.
Dynamic Testing	Testing of an object with execution on a computer.
Effectiveness (in the context of software testing and test case design methods)	The accuracy and completeness with which testers achieve specified test case design goals.
Efficiency (in the context of software testing and test case design methods)	Resources expended in relation to the accuracy and completeness with which testers achieve specified test case design goals.
Equivalence Class	A portion of the component's input or output domains for which the component's behaviour is assumed to be the same from the component's specification. (BS 7925-1)
Equivalence Partitioning (EP)	A test case design method in which test cases are designed to execute representatives from equivalence classes. (adapted from (BS 7925-1))
Error	A human action that produces an incorrect result, such as software containing a fault. (ISO/IEC 24765:2009)
Error Guessing (EG)	A test case design method where the experience of the tester is used to postulate what faults might occur, and to design tests specifically to expose them. (adapted from (BS 7925-1))
Exploratory Testing (ET)	A testing approach where the test design and execution are conducted concurrently. (adapted from (Craig & Jaskiel 2002))
Expected Result	The behaviour predicted by the specification of an object under specified conditions. (called the 'predicted outcome' in (BS 7925-1))
Fault	A manifestation of an error in software. (ISO/IEC 24765:2009)
Failure	An event in which a system or system component does not perform a required function within specified limits (ISO/IEC 24765:2009)
Failure Detection Effectiveness	The ability of a test case design method to detect failures in software.
Functional Testing	See *Black-Box Testing Method*.
Grey-Box Testing	Test case selection that is based on an analysis of the specification and source code of the system under test.
The Institute of Electrical and Electronic Engineers (IEEE)	The Institute of Electrical and Electronic Engineers, Inc. Publisher of engineering standards. (Craig & Jaskiel 2002)
Independent Testing	An organizational strategy where the testing team and leadership is separate from the development team and leadership. (Craig & Jaskiel

Term / Abbreviation	Meaning
	2002)
Input	A variable (whether stored within a component or outside it) that is read by the component. (BS 7925-1)
Input Data	See *input value*.
Input Data Specification	A specification of the input data of a program.
Input Domain	The set of all possible inputs to a program. (adapted from (BS 7925-1))
Input Value	An instance of an input. (BS 7925-1)
Integration Testing	A level of test undertaken to validate the interface between internal components of a system. Typically based upon the system architecture. (Craig & Jaskiel 2002)
The International Organization for Standardisation (ISO)	The International Organization for Standardisation. Publishes of industry standards.
Learnability (of a Test Case Design Method)	Attributes of a test case design method that determine the effort required by a tester to learn how to apply the method competently.
Metalanguage	Language used to describe another language (for example, XML is the metalanguage for XHTML). (Bidgoli 2004)
Module	See *Component*.
Nominal Value	The mid-point of an equivalence class. (Jorgensen 1995)
Non-Functional Testing	Testing of those requirements that do not relate to functionality. i.e. performance, usability, etc. (BS 7925-1)
Operability (of a Test Case Design Method)	Attributes of a test case design method that determine the effort required by a tester to use the test method competently.
Outcome	The outcome of a test case.
Output	A variable (whether stored within a component or outside it) that is written to by the component. (BS 7925-1)
Output Domain	The set of all possible outputs of a program. (adapted from (BS 7925-1))
Output Value	An instance of an output. (BS 7925-1)
Partition	See *Equivalence Class*.
Random Testing (RT)	Testing using data that is in the format of real data, but with all of the fields generated randomly. (Craig & Jaskiel 2002)
Regression Testing	Retesting previously tested features to ensure that a change or bug fix has not affected them. (Craig & Jaskiel 2002)

Term / Abbreviation	Meaning
Requirement	A condition or capability that must be met or possessed by a system, product, service, result, or component to satisfy a contract, standard, specification, or other formally imposed document. Requirements include the quantified and documented needs, wants, and expectations of the sponsor, customer, and other stakeholders. (PMI 2004)
Requirements	See *requirement*.
Requirements Traceability	Demonstrating that all requirements are covered by one or more test cases. (Craig & Jaskiel 2002)
Requirements Traceability Matrix (RTM)	A matrix used to track requirements coverage.
Satisfaction (in the context of software testing and test case design methods)	Freedom from discomfort and positive attitudes towards the use of a testing case design method.
Specification	A description of a component's function in terms of its output values for specified input values under specified preconditions. (BS 7925-1)
Specified Input	An input for which the specification predicts an outcome. (BS 7925-1)
Stakeholder	Individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations. (ISO/IEC 12207:2008)
State Transition Testing	A test case design method in which test cases are designed to execute state transitions. (adapted from (BS 7925-1))
Static Testing	Testing of an object without execution on a computer. (BS 7925-1)
Syntax Testing (ST)	A test case design method for a component or system in which test case design is based upon the syntax of the input. (adapted from (BS 7925-1))
System Testing	A (relatively) comprehensive test undertaken to validate an entire system and its characteristics. Typically based upon the requirements and design of the system. (Craig & Jaskiel 2002)
Test Case	A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to execute a particular program path or to verify compliance with a specific requirement. (ISO/IEC 24765:2009).
Test Case Construction Rule	One Atomic Rule from a test case design method that describes how to design a test case.
Test Case Design Method	A method used to derive or select test cases. (BS 7925-1)
Test Condition	An item or event of a component or system that could be verified by one or more test cases, e.g. a function, transition, feature, quality attribute, or structural element. (ISTQB 2005)
Test Data	Data (including inputs, required results, and actual results) developed or used in test cases and test procedures. (Craig & Jaskiel 2002)

Term / Abbreviation	Meaning
Test Data Value	A single item of test data.
Test Method	See *Test Case Design Method*.
Test Method Usability	The extent to which a test case design method can be understood, learnt and used by software testers to achieve specified test case design goals effectively, efficiently and with satisfaction, within the context of applying software testing methods.
Test Oracle	An oracle provides a method to generate expected results for the test inputs and compare the expected results with the actual results of execution of the system under test. (adapted from (Naik & Tripathy 2008))
Test Procedure	Detailed instructions for the setup, execution, and evaluation of results for a given test case. (IEEE 1012:2004)
Test Script	Commonly used to refer to the automated test procedure used with a test harness. (BS 7925-1)
Test Set	A collection of one or more test cases for the software under test. (adapted from (BS 7925-1))
Understandability (of a Test Case Design Method)	Attributes of a test case design method that determine the effort required by a tester to recognise the logical concept of the method and its applicability.
Unit Testing	A level of test undertaken to validate a single unit of code. Typically conducted by the programmer who wrote the code. (Craig & Jaskiel 2002)
User Acceptance Testing	See *Acceptance Testing*.
Validation	Confirmation, through the provision of objective evidence, that the requirements for a specific intended use or application have been fulfilled. (ISO/IEC 15288:2008)
Verification	The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. (IEEE 1012:2004)
White-Box Testing	Test case design that is based on an analysis of the internal structure of the source code of a program. (adapted from (BS 7925-1))

Acknowledgments

I would like to acknowledge and thank everyone who contributed their time, energy, love and friendship to my thesis. Without you by my side, my thesis, and my life, would not be what it is today.

To my parents, John and Patricia Murnane, and my brothers, Tristrim and Tarquin Murnane, I thank you for your ongoing love and support. I especially thank my father John, for the countless hours of consultation you set aside to discuss my ideas and to review my thesis.

My heartfelt thanks to my supervisors Karl Reed and Richard Hall. Without your unwavering guidance and support, I would never have been able to see the completion of my thesis, or the beginnings of a career that continues to bring me so much happiness and inspiration.

I thank my friends and colleagues, present and past, for their support throughout this journey. I thank Ben Bonanno, Sarah Pulis, Helen Trevithick, Kylie Pepyat, Tasha Findlay, Gillian Moxom, Stuart Reid, Bernard Baudoin, Anne Mette Hass, Linda Reeves, Kelvin Ross, Mark Pedersen, Paul Grimsley, Paul Strooper, Bob Glass, Erik Petersen, Ayla Barutchu, Tony Brown, Daniel Powell, Fevzi Belli, George Gimian, Benjamin Mintern-Lane, Jayanthi Ramachandran, Philip Lee, Lee Elms, Julie Main, Mary Witten, members of the La Trobe University Staff and Postgraduate IT Research Group (SPRIG) including Mark Dawes, James Munro, Anne Hannington, Jessica Müller, Jeanette Auer, Raymond Matthews, Michael Hill, Robert Matterson, Lorien Dunn, Benji Sasson, Peter Donald and Jean Hall, members of the Melbourne Association of Software Testers (MAST) including Sophie Hiotis, Jared Quinert, Erik Petersen, Paul Szymkowiak and Stuart Moncrieff, all the eager participants of my university and industry experiments and the kind people who critiqued my ideas at various conference and university presentations.

Last but not least, I thank my dearly adored cats, Missie and Starrie, whose warmth, love and support during those late nights and long days shall always be remembered.

To my friends and family who have supported me throughout this long and arduous journey…

…with all of my heart, I thank you.

“We all tend to tie our self-esteem strongly to the quality of the product we produce - not the quantity of the product, but the quality.”

DeMarco and Lister, 1999.

Chapter 1

Introduction

"It can be frustrating, as a tester, to watch someone who has no experience in testing spend five minutes using a piece of software and crash it."

Ron Patton, 2006

1.1 Overview

In 1969, Dijkstra famously observed that "Program testing can be used to show the presence of bugs, but never to show their absence!" (Dijkstra 1969), meaning that no amount of software testing can guarantee that a program is fault-free. On the other hand, software testing can be used to prove that a program both meets its requirements and does not fail when given specific types of inputs. Meeting this objective can be defined as achieving '*quality*', which the Institute of Electrical and Electronic Engineers (IEEE) define as "the degree to which a system, component, or process meets specified requirements" (IEEE 829:2008). This definition of quality refers to the term '*requirement*', which the joint technical committee of the International Organization for Standardisation (ISO) and the International Electrotechnical Commission (IEC) define as "a condition or capability needed by a user to solve a problem or achieve an objective" (ISO/IEC 24765:2008). One approach to determining whether *faults* or *quality* (i.e. satisfied user requirements) are present in a program is through the use of 'test case design methods' (also known as 'testing techniques' (Dustin 2003, Copeland 2004, Burnstein 2003)). Since 'exhaustive testing' (e.g. testing all source code paths or input combinations) has long been considered impractical (Goodenough & Gerhart 1975), test case design methods arose to facilitate the selection of 'effective' test cases that 'cover' program source code and requirements in various ways. The results of testing can be used to communicate the current level of program quality to relevant stakeholders, including users, the financers of software development projects, project managers, test managers, business analysts, developers and testers. In the most general sense, testing reduces uncertainty about the quality of a program and its release readiness.

Test case design methods can be divided into thee classes: black-box, white-box and grey-box (see Section 1.3). While each of these concepts could support a thesis in its own right, the focus of this thesis is on the usability and failure-detection effectiveness of black-box testing methods (see Section 1.4). Black-box testing methods can be divided into two classes: prescriptive and non-prescriptive. The term 'non-prescriptive testing' is used in this thesis to describe any unscripted test case design approach that is based on a tester's domain knowledge and experience, which could be drawn from their knowledge of prescriptive testing methods, but where the tester does not explicitly follow any specific test case design guidelines. This

has also been called ‘reactive testing’ (Black 2007), ‘lateral testing’ (after De Bono’s lateral thinking techniques) (Fewster & Graham 2000), ‘unstructured testing’ and ‘ad hoc testing’ (Nguyen et al. 2003).

In *prescriptive* black-box testing, test cases are designed by applying ‘test case design rules’ (also called ‘test case design guidelines’ (Myers 1979) and ‘heuristics’ (DeMillo et al. 1987)) from black-box testing methods to program requirements. Prescriptive methods intend to add precision and procedure to the testing process, and can provide a basis for measuring test coverage and adequacy (Zhu, Hall & May 1997). On the other hand, *non-prescriptive* testing is unstructured and unscripted, being based on a tester’s unique domain knowledge and experience. This can stem from a tester’s knowledge of, or experience with, prescriptive testing methods (Craig & Jaskiel 2002), testing heuristics (Watkins 2001), tests that previously detected faults (Watkins 2001), program implementation and design (Bertolino 2004, Watkins 2001), hardware (Mosley 1993), platforms (Bertolino 2004) and programmer assumptions (Myers 1979). Non-prescriptive approaches like Error Guessing are said to compensate for the “inherent incompleteness” of EP and BVA (Mosley & Posey 2002). In one empirical study of failures detected in independently developed launch-intercept control software, it was established that 83-90% of faults and 90-97% of failures were detected by ‘special values’ (Wild, Chen & Eckhardt 1989), which can be chosen through non-prescriptive testing. Non-prescriptive testing approaches are believed by some to be among the most popular in the software testing industry (Jorgensen 1995). Empirical data supporting this includes a survey of software testing practices in Australia, which revealed that out of 65 organisations interviewed, just over one third (35.4%) chose to use non-prescriptive approaches to testing over prescriptive black-box testing methods (Ng et al. 2004).

The aim of this thesis is to investigate and improve the *usability* and *failure-detection effectiveness* of prescriptive and non-prescriptive approaches to black-box testing. The investigation is focussed on resolving seven problems with existing definitions of black-box testing methods (see Section 1.4), which can affect the usability and failure-detection effectiveness of the methods. Problems resolution is achieved through the creation of a new approach to describing black-box testing methods called ‘Atomic Rules’ (see Section 1.5). In the Atomic Rules approach, black-box testing methods are decomposed into individual test case design rules called ‘Atomic Rules.’ Each Atomic Rule is defined using a characterisation schema, while a four-step test case design process provides a common notation for describing all black-box testing methods. The Atomic Rules approach has been evaluated via two classroom experiments and an industrial experiment, as well as through the implementation of a proof-of-concept testing tool (see Section 1.7). Two approaches called Systematic Method Tailoring (SMT) and Goal/Question/Answer/Specify/Verify (GQASV) are also introduced, to further improve the effectiveness of black-box testing.

1.2 Test Method Usability

Before exploring problems that affect the usability of black-box testing methods, the term *usability* needs to be defined within the context of software testing and the study of test case design methods. This facilitates identification of qualitative and quantitative attributes that can be used to assess test method usability. Benchmark definitions from software engineering are provided first and are then redefined within the context of software testing.

Within the context of software engineering, ISO/IEC define the term *usability* as follows (ISO/IEC 24765:2008):

> **"Usability. The extent to which a product can be used by specified users to achieve specified goals with effectiveness efficiency and satisfaction in a specified context of use."**

In this thesis, the term 'effective' is used within two contexts. A test method can be 'effective' if a tester can use it to derive an 'accurate' and 'complete' test set (see Section 1.2.1). A test method has a 'failure-defection effectiveness' that can rate its ability to detect program failures (see Section 1.2.2).

ISO/IEC further defines three sub-attributes of usability as *understandability*, *learnability* and *operability* (ISO/IEC 9126-1:2005), as follows.

> **"Understandability. Attributes of software that bear on the user's effort for recognising the logical concept and its applicability."**
>
> **"Learnability. Attributes of software that bear on the user's effort for learning its application (for example, operation control, input, output)."**
>
> **"Operability. Attributes of software that bear on the user's effort for operation and operation control."**

ISO/IEC 9126-1:2005 also includes 'attractiveness' and 'usability compliance' under its definition of usability. Attractiveness is excluded here since this can be assessed under 'satisfaction' (see Section 1.2). Usability compliance is excluded as there are no laws or regulations in software testing that dictate how a test method should be represented.

Within the context of software testing and the study of test case design methods, the objective is for a tester to design an adequate (e.g. effective) set of test cases. Thus, in the definitions below, the terms *product* and *software* have been replaced by the term *software testing method*, while *user* has been replaced by the phrase *software tester*. This results in the following definitions.

> **Understandability. Attributes of a test case design method that determine the effort required by a tester to understand a test method and to understand if, when and how it applies to the program under test[1].**
>
> **Learnability. Attributes of a test case design method that determine the effort required by a tester to learn how to apply the method competently.**
>
> **Operability. Attributes of a test case design method that determine the effort required by a tester to use the test method competently.**

The ISO/IEC definition of *usability* also refers to the terms *effective*, *efficient* and *satisfaction*. ISO/IEC provide benchmark definitions for these, as follows (ISO/IEC 25062:2006, ISO/IEC24765:2008).

[1] The definition of understandability does not include the effort to use the test method, as this is covered by operability.

"Effectiveness. The accuracy and completeness with which users achieve specified goals."

"Efficiency. Resources expended in relation to the accuracy and completeness with which users achieve goals."

"Satisfaction. Freedom from discomfort, and positive attitudes towards the use of the product."

Within the context of software testing and the study of test case design methods, these terms can be redefined as follows.

Effectiveness. The accuracy and completeness with which testers achieve specified test case design goals during the application of a test case design method[2].

Efficiency. Resources expended in relation to the accuracy and completeness with which testers achieve specified test case design goals during application of a test case design method.

Satisfaction. Freedom from discomfort and positive attitudes towards the use of a test case design method.

While the first two measures (effectiveness and efficiency) can be measured objectively, the third (satisfaction) is included since subjective reactions to a test case design method may effect the extent to which that method is adopted.

Combining the above definitions, *test method usability* can be defined as follows.

Test Method Usability. The extent to which a test case design method can be understood, learnt and used by software testers to achieve specified test case design goals effectively, efficiently and with satisfaction, within the context of applying software testing methods.

These definitions will be used throughout this thesis to examine and evaluate black-box test method usability. They apply to novice and experienced testers, since both are affected by the ease of which a test method can be learnt, understood and used. Since a tester's own personal experience can impact on whether they find a test method 'usable' and whether they are capable of using it 'effectively', experience will be taken into account during experimental analysis[3].

1.2.1 Assessing Usability

To facilitate assessment of test method usability in an experimental context, quantitative and qualitative attributes need to be identified for measuring various aspects of usability, as follows.

- **Understandability** can be qualitatively evaluated by assessing whether a tester understands the conditions under which a test case design method should be applied. It can also be quantitatively assessed by examining their ability to apply the method correctly (measured by effectiveness, see

[2] A more traditional definition of test effectiveness as "failure-detection effectiveness" is provided in Section 1.2.2.
[3] Influences external to the test method can also affect a tester's ability to derive effective test cases (e.g. specification quality).

below). Although subjective, it can also be measured by a tester's self-rated understanding of a test method.

- **Learnability** can be quantitatively measured by the time it takes a tester to become competent in the use of a test method. It can be quantitatively and qualitatively assessed by examining the quantity and types of questions asked by a tester when they are learning to apply a test method.
- **Operability** can be quantitatively assessed by examining the proportion of correct test cases derived by a tester during application of a test case design method (measured by effectiveness, see below). Although subjective, it can also be quantitatively assessed by examining a tester's opinion of how easy the method is to use.
- **Effectiveness** consists of sub-attributes *completeness* and *accuracy*, as follows.
    - **Completeness** can be quantitatively measured as the proportion of all test cases that are derivable by an expert software tester when applying a particular test case design method (referred to as the 'total' number of test cases derivable).
    - **Accuracy** can be quantitatively and qualitatively assessed by the frequency and types of mistakes (i.e. errors) made by a tester during application of a test case design method.
- **Efficiency** can be quantitatively measured by examining the productivity of a tester; i.e. by the number of correct test cases that are derived by a tester during application of a test case design method over the total time taken.
- **Satisfaction** is subjective but can be qualitatively assessed by comparing a tester's preference for using one test case design method over another.

These will be used to evaluate the usability of black-box testing methods (see Chapters 5 and 6).

1.2.2 Failure-Detection Effectiveness

In the previous section, effectiveness was defined as "The accuracy and completeness with which testers achieve specified test case design goals during the application of a test case design method." A more traditional definition of effectiveness in test case design is the ability of a test method to detect program faults (defects) or failures ('activated' defects) (Reid et al. 1999). Since one fault can cause more than one failure, and one failure can be caused by more than one fault, the following definition, referred to as "failure-detection effectiveness" (to differentiate it from the definition of effectiveness given in the previous section) will also be used in this thesis when assessing test method effectiveness.

Failure-Detection Effectiveness. The ability of a test case design method to detect failures in software.

This can be calculated as the proportion of all program failures that are detectable by a test case design method, when the method is applied by an expert tester. This can be affected by the size of the program under test, the number and severity of faults it contains and the capability of the tester. This definition will

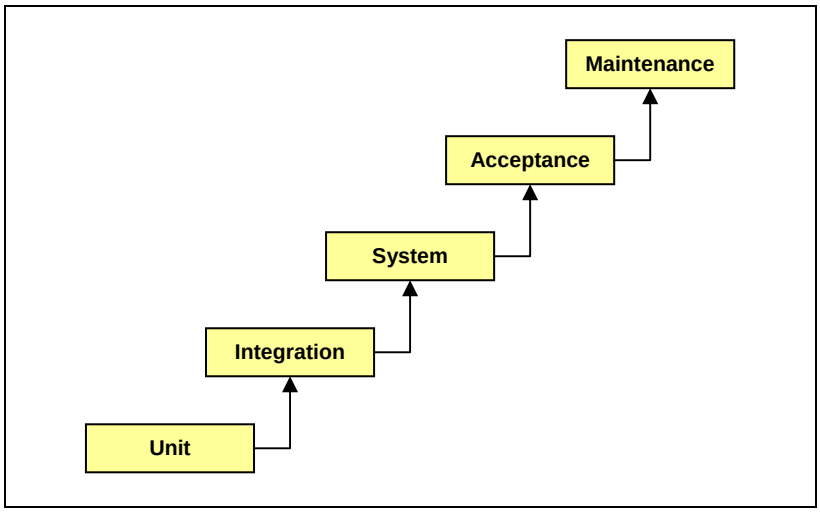
be used in Chapter 6 when evaluating the ability of black-box test case design methods to detect program failures.

1.3 Black-Box Testing Methods

Before exploring problems with existing black-box testing methods that affect their usability and failure-detection effectiveness, (see Section 1.4), it is important to consider the place of these methods within the broader context of software testing.

Black-box testing methods can be applied at any of the four 'levels' of testing that are commonly recognised in the software testing industry: Unit, Integration, System and Acceptance (Pfleeger 2001). More recently, a fifth level called Maintenance Testing was identified (Figure 1-1)[4] (ISO/IEC 29119-1). During *Unit Testing*, individual program modules are tested separately to check that they each meet their requirements. In *Integration Testing*, interactions between components are tested. During *System Testing*, the fully integrated system is tested to ensure it functions correctly in a test environment that is as close to the operational (i.e. production) environment as possible. *Acceptance Testing* is also used to determine whether the complete system meets its user requirements, and is often used as a means for obtaining 'sign off' from the customer, indicating that they agree that the system is ready for release. After systems have been accepted by customers, they often require enhancement and repair. Thus, during *Maintenance Testing* the system is tested after new code is implemented and re-tested or regression tested after existing code is changed, to ensure that it still meets user requirements. If changes are substantial then *Maintenance Testing* can consist of all four levels of testing (i.e. Unit, Integration, System and Acceptance Testing).

Figure 1-1: A five-level model for testing (ISO/IEC 29119-1).



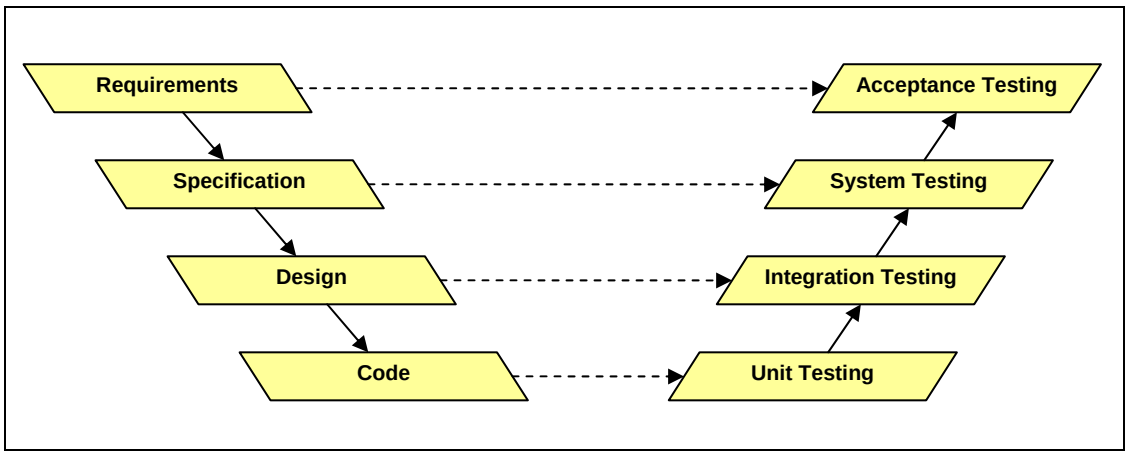
Independence between testing and development teams is often sought during certain levels of testing. Independent testers are often able to detect defects that developers miss because they think differently from them (Kaner et al. 2001) and since developers are can be subjective about the quality of their own source code. This avoids the conflict of interest between the necessity to find defects and the need to take

[4] This concept of Maintenance Testing has been defined in the new ISO/IEC 29119 Software Testing standard that is currently under development by ISO/IEC JTC1/SC7, Working Group 26 (Software Testing), for which Ms. Murnane is an editor.

responsibility for them (Pressman 1992) and the necessity to meet delivery schedules[5]. Nonetheless, Unit Testing is often carried out by program developers, since some methods used at that level (e.g. white-box testing methods) require an understanding of program source code. Integration and System Testing are usually performed by independent testers who are not directly involved with the development of the program source code, while Acceptance Testing is often carried out by the system's end-users, using a predefined test suite that is chosen by the customer.

The relationship between the four traditional levels of testing and the phases of the software development lifecycle (SDLC) can best be seen in the V-Model for software development, in which each development phase is supported by a level of testing and where the outputs of each development phase (i.e. requirements, specifications, designs, source code) can be used for test case design (Figure 1-2).

Figure 1-2: The V-Model (adapted from (Burnstein 2003) and (V-Modell® XT 2008)).



Across the SDLC, there are two main classes of testing: static and dynamic testing (Burnstein 2003). Static testing has also been referred to as 'verification testing' (Perry 2006) and 'static analysis' (Burnstein 2003, Naik & Tripathy 2008), while dynamic testing has also been called 'validation testing' (Perry 2006) and 'dynamic analysis' (Burnstein 2003, Naik & Tripathy 2008).

In *static testing*, program source code, requirements, specifications and other documentation (e.g. user and installation manuals) are manually 'tested' through the application of static testing methods, including reviews, walkthroughs, inspections, desk debugging, requirements testing (Perry 2006) and desk checking (Everett & McLeod 2007), each with varying levels of formality. The aim of static testing is to identify defects (e.g. ambiguities, inconsistencies, deviations from user requirements) in the program source code and program documentation as early in the SDLC as possible. As Burnstein observes (2003), some members of the software testing community do not consider static testing techniques like inspections, walkthroughs and reviews to be 'testing' techniques. Instead, they consider static testing to be a form of *software quality assurance*. Conversely, other members of the testing community (including (Burnstein 2003, Everett & McLeod 2007, Hetzel 1988) and this author) consider static testing to be a form of testing.

[5] Testers are also often trained in the use of prescriptive testing methods that can be used to systematically question the completeness and correctness of requirements, specifications and source code. For the same reason, it is also beneficial to seek independent testing of program documentation (e.g. requirements) (Everett & McLeod 2007).

Furthermore, the author follows the ISO/IEC standard definition of *quality assurance*, which is "a set of activities designed to evaluate the process by which products are developed or manufactured" (ISO/IEC 24765:2009), which is clearly a different activity to testing.

In *dynamic testing*, the program is executed to determine whether it meets its requirements and specifications (Perry 2006). Arguably the most common form of testing that is used within the software testing industry, dynamic testing encompasses any prescriptive or non-prescriptive test case design method that can be used to identify test cases, which can then be run against a program as a means of determining whether it meets its requirements. Therefore, this includes black-box testing methods.

Within both static and dynamic testing, there are two main classes of test case design methods: functional and non-functional (e.g. see (Everett & McLeod 2007, BS-NFT 2008)). *Functional testing methods* are used to identify test cases that determine whether a program satisfies its functional requirements, such as business rules or core functionality. *Non-functional testing methods* are used to derive test cases that check whether a program satisfies its so-called non-functional requirements, such as performance, reliability, usability and security.

Within *functional testing*, there are three main classes of test case design: white-box, black-box and grey-box (Figure 1-3).

White-box test case design is based on knowledge of the internal structure of the program source code. The aim is to design test cases that 'cover' the code in various ways, such as testing 'all paths' or 'all branches.' Statement testing, path testing, branch testing, condition testing and data-flow testing are all examples of white-box testing methods (Weiser et al. 1985, Pfleeger 2001). White-box testing can uncover faults that may be unlikely to be revealed through black-box testing, such as boundary faults on condition statements that are not specifically documented in program specifications. White-box testing methods are typically used by developers during Unit Testing, but can also be applied during Integration and System Testing when checking the correctness of program control flows and component interactions. White-box testing is also known as glass-box testing (Burnstein 2003), clear-box testing (Burnstein 2003), structural testing (Perry 2006) and logic-driven testing (Myers 1979).

Black-box test case design is based primarily on knowledge gained from program documentation, including requirements, specifications and user manuals. The aim is to design a set of test cases that cover a program's functionality and input/output domains in various ways. For example, Equivalence Partitioning was designed to provide guidance on the partitioning of program input and output domains into sets of equivalent data, and is intended to reduce the number of individual test data values that must be executed against the program in order to achieve adequate input/output domain coverage (Myers 1979). Boundary Value Analysis and Syntax Testing were designed to select test data values that target particular fault classes (Burnstein 2003). Use Case Testing guides in the design of test cases that can be used to check for correctness in program workflows and program/user interactions.

Rather than being based on *how* a program is written, black-box testing allows programs to be tested from the user's perspective. Black-box testing methods can be prescriptive or non-prescriptive (see Table 1-1) and can be applied at any level of testing, although their semantics may change depending on the level at which they are applied. For example, Boundary Value Analysis could be used during Unit Testing to test the boundaries of input fields or at System Testing to test the maximum number of users that can be simultaneously logged into a system. Although black-box testing methods were designed for dynamic testing, most can be utilised for static testing, such as using Boundary Value Analysis to question whether boundaries are defined in input/output data specifications (see Chapter 2).

Black-box testing is also known as functional testing (Perry 2006), specification testing (Burnstein 2003), specification-based testing (Pezzand & Young 2008), closed-box testing (Pfleeger 2001), input/output driven testing (Myers 1979), data-driven testing (Myers 1979) and behaviour testing (Everett & McLeod 2007).

A black-box testing method can also be utilised in a grey-box manner, if the tester uses knowledge of the source code and/or coding techniques during test case design. Similarly, during white-box test design, a tester will likely utilise knowledge of program inputs and outputs.

Figure 1-3: Black-box, white-box and grey-box testing methods (adapted from (Burnstein 2003)).

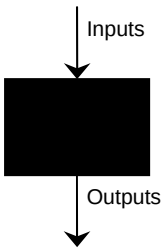
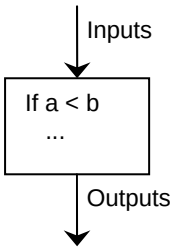
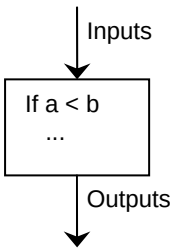
Test Method Class	Tester's View	Dominant Knowledge Sources	Test Methods
Black-box		Requirements documents Specifications Domain knowledge Manuals Defect analysis Data definitions	Boundary Value Analysis (BVA) Category Partition Method (CPM) Cause-Effect Graphing (CEG) Classification Trees (CT) Combinatorial Test Methods Decision Tables/Trees Equivalence Partitioning (EP) Error Guessing (EG) Exploratory Testing (ET) Random Testing (RT) State-Transition Testing Syntax Testing (ST) Test Catalogues Test Categories Test Matrices
White-box		Design Source code Control flow graphs Data flow graphs Cyclomatic complexity	Branch Condition Combination Testing Branch Condition Testing Branch/Decision Testing Data Flow Testing Linear Code Sequence and Jump Testing Modified Condition Decision Testing Mutation Testing Path Testing Statement Testing
Grey-box		Requirements documents Specifications Domain knowledge Manuals Defect analysis Data definitions Design Source code Control flow graphs Data flow graphs Cyclomatic complexity	Partition Testing

Table 1-1: Classification of black-box testing methods as prescriptive or non-prescriptive.

Black-Box Testing Method	Prescriptive	Non-Prescriptive
Boundary Value Analysis (BVA)	✓	
Category Partition Method (CPM)	✓	
Cause-Effect Graphing (CEG)	✓	
Classification Trees (CT)	✓	
Combinatorial Test Methods	✓	
Decision Tables/Trees	✓	
Equivalence Partitioning (EP)	✓	
Error Guessing (EG)		✓
Exploratory Testing (ET)		✓
Random Testing (RT)	✓	
State-Transition Testing	✓	
Syntax Testing (ST)	✓	
Test Matrices, Catalogues and Categories	✓	✓
Use Case Testing	✓	

Grey-box testing is a hybrid of white-box and black-box testing, in which test case design is based on knowledge of the program source code and specifications (e.g. see (Dustin 2003)). An example of grey-box testing is *Partition Analysis*, where test cases are chosen by locating input data partitions that execute the same program source code paths (Howden 1976, Richardson & Clarke 1981, Richardson & Clarke 1985) (Table 1-2). The precision of black-box testing can be improved through the use of grey-box information. Thus, although black-box testing methods were not specifically designed for grey-box testing, some can be applied using a combination of black-box and grey-box information (Table 1-2).

One approach to classifying the differences between black-box and white-box testing methods was proposed by Vegas, Juristo and Basili (2003), who developed an instantiated characterization schema to classify black-box and white-box testing methods, as well as data-flow and mutation testing methods. The schema was designed to support testers in the selection of the "best suited" methods for testing. While the schema enabled classification of various black-box testing methods, it did not address the key problems with existing black-box testing methods that are introduced in the next section (see Section 1.4).

Table 1-2: Classification of test case design methods as black-box, white-box or grey-box.

Functional Testing Methods	Test Method Classification ✓ denotes methods defined for the stated purpose, ~ denotes methods that can be used for the stated purpose but that were not originally designed for it		
	Black-Box	**White-Box**	**Grey-Box**
Boundary Value Analysis (BVA)	✓	~	~
Branch Condition Combination Testing		✓	
Branch Condition Testing		✓	
Branch/Decision Testing		✓	
Category Partition Method (CPM)	✓		
Cause-Effect Graphing (CEG)	✓		
Classification Trees (CT)	✓		
Combinatorial Test Methods	✓		
Data Flow Testing		✓	
Decision Tables/Trees	✓		
Equivalence Partitioning (EP)	✓		~
Error Guessing (EG)	✓	~	~
Exploratory Testing (ET)	✓	~	~
Linear Code Sequence and Jump Testing		✓	
Modified Condition Decision Testing		✓	
Mutation Testing		✓	
Partition Testing			✓
Path Testing		✓	~
Random Testing (RT)	✓		
Statement Testing		✓	
State-Transition Testing	✓		~
Syntax Testing (ST)	✓		
Test Matrices, Catalogues and Categories	✓		
Use Case Testing	✓		

1.4 Seven Problems with Existing Black-Box Testing Methods

While an extensive literature search indicated that the earliest definitions of test case design methods were white-box-based (Miller & Maloney 1963), one of the earliest references found that treated an algorithm as a "black-box" was in 1958 (Wolpe 1958). The earliest found examples of prescriptive black-box and grey-box testing methods were Sauder's (compiler) syntax testing method in 1962 (Sauder 1962), grey-box Partition Analysis in 1976 (Howden 1976, Richardson & Clarke 1981) and Myers' black-box Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) in 1979 (Myers 1979)[6]. The first non-prescriptive black-box testing approaches to be defined were Error Guessing in 1979 (Myers 1979) and Exploratory Testing in 1988 (Kaner 1988). Many of the earliest prescriptive black-box testing methods have been enhanced over the past thirty years and are still widely taught and used today. Despite this, a number of problems with their existing definitions need to be addressed.

[6] Myers' textbook 'The Art of Software Testing' (1979) is still one of the most oft-referenced textbooks on software testing today (e.g. see (Burnstein 2003, Jorgensen 1995, Kit 1995, Mosley 1993, Copeland 2004, Page et al. 2009, Mosley & Posey 2002, Parrington & Roper 1989)).

Ideally, any prescriptive testing method would be interpreted in the same way by different testers, such that when it is applied to a program specification, it results in an 'equivalent' test set, regardless of a tester's unique domain knowledge or experience. Further, each method would be complete and lead to the generation of all possible test cases that are derivable by the method for the given specification. In reality, inconsistencies, ambiguities and a lack of precision in existing black-box test methods definitions can lead to differing interpretations and thus varying test set quality. Some practitioners may argue that skilled black-box testers should be capable of deriving high-yield test sets using only their domain knowledge and experience; in their view, test methods like EP should only be used to supplement heuristic knowledge (Sommerville 2001). This raises the question of how testers obtain such skill in the first place. In practice however, prescriptive black-box testing methods are an essential part of the software testing process.

In this thesis, the following five problems that affect the usability and failure-detection effectiveness of existing black-box testing methods were initially identified and explored (Murnane, Hall & Reed 2005):

1. definition by exclusion;
2. multiple versions;
3. method overlap;
4. notational and terminological differences; and
5. reliance on domain knowledge.

Two additional problems, which were identified as the research into this topic progressed and which are also addressed in this thesis, are that existing black-box testing methods can be:

6. difficult to audit; and
7. difficult to automate.

Definition by exclusion relates to black-box test case design methods that involve 'partitioning.' During partitioning, program input and output domains are divided into classes of homogenous data, whose mapping involves executing (ideally) identical deterministic processes. As Ostrand and Balcer (1988) observe, "Various methods of creating a test partition are discussed in the literature. Despite the general agreement on this key technique of functional testing and its use since the earliest days of computing, the partitioning process lacks a systematic approach." For example, Myers (1797) provides the following guideline for EP:

> **"If an input condition specifies a 'must be' situation (e.g. 'first character of the identifier must be a letter'), identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter)."**

This describes the identification of two partitions of data: one containing valid test data values and the other containing all other (invalid) values that were not included in the valid set. Ideally, a member of every class of data would be included in the invalid partition, but novice testers may only be aware of a subset of

classes that are available. Thus, depending on a tester's interpretation of the test method, their assumptions about the program under test, their domain knowledge and their experience, each tester could produce a vastly dissimilar test set, resulting in black-box testing that is not repeatable or predictable. For example, for programs developed on 'western' keyboards, data in the invalid class could be identified through the ASCII (American Standard Code for Information Interchange) table (e.g. see (Oualline 2003)), which defines 94 printable characters that can be divided into classes of uppercase alpha, lowercase alpha, numeric and non-contiguous non-alphanumeric (i.e. special) characters (see Appendix G). In other programs, Unicode character sets (Aliprand et al. 2003) may need to be considered. For programs utilising EBCDIC (Extended Binary Coded Decimal Interchange Code), various classes of data could be defined for each non-contiguous set of alpha and non-alphanumeric characters that are defined in that encoding scheme. Thus, ***definition by exclusion*** assumes testers are familiar with the 'universe of discourse' of program inputs. In reality, program specifications often do not specify the valid and invalid data sets that should be considered by developers and testers, let alone the encoding schemes that are in use. In addition, test method definitions like those provided by Myers (1979) can be interpreted differently by each tester. As a consequence, the effectiveness of resulting test sets may be diminished and statements relating to program correctness and the nature of program faults detected may be meaningless. Ambiguous test method definitions may also lead to testers being unsure of how to apply the test method correctly. The lack of prescriptive guidelines for black-box testing also makes black-box test data generation ***difficult to automate***.

Multiple versions of each black-box testing method exist. For example, some definitions of BVA describe the selection of test data on, inside and outside field boundaries (BS 7925-2), while others do not include inside (Craig & Jaskiel 2002, Kaner 1988, Lewis 2000, Mosley 1993, Myers 1979) and outside (Jorgensen 1995, Mosley 1993) boundaries (see Section 2.3). Presently, no textbook, standard or paper describes the complete version (i.e. all test case design rules) from all black-box testing methods. Thus, a tester may not know how their chosen approach compares to all others, even within a method and, as a result, they may be unaware of how complete (or incomplete) their resulting test set will be.

Without a standard definition of each method, it can be ***difficult to audit*** the completeness of black-box testing. For example, the British Standard BS EN 50128 'highly recommends' the use of EP and BVA for certain testing classes of safety critical systems (BS 50128:2001). However, the completeness of black-box testing that is claimed to be conformant with that standard cannot be guaranteed, since it depends on which version of the methods are used and the level of precision that each version provides.

Notational and terminological differences in the definitions of these methods further impact on test method usability. As Jorgensen (1995) observed, "Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers." As a result, a new notation often must be understood for each new method learnt. For example, various names have been used to describe the process of partitioning program input and output domains into homogenous data sets. In the Category Partition Method (CPM) these sets are called 'choices' (Ostrand & Balcer 1988) while in EP they are called 'equivalence classes' and 'partitions' (Myers 1979). While EP and BVA are described as 'partitioning' approaches (e.g. see (Myers 1979)), ST is not described

in this way, despite the fact that partitions are implicitly created on each input and output field (see Section 2.2.3). These differences may make it difficult for novice and experienced testers to learn new test case design methods. This problem is compounded by black-box test method definitions that only provide examples that demonstrate method application without prescriptive definitions of the test case design rules underlying each method (e.g. see definitions of EP in (Kaner 1988, Tamres 2002)). If test case design rules cannot be easily identified by testers, it could result in incomplete test sets and ineffective testing.

Method overlap between black-box testing methods can result in duplicated test cases and inefficient testing. For example, boundary values are selected by at least two versions of ST (Beizer 1990, Marick 1995), while EP, BVA and ST all include test case design rules to select test data values that lie outside the boundaries of numeric fields (e.g. see (Myers 1979, Beizer 1990)). Although ST is described differently to EP and BVA, these overlaps suggest that all three methods could be defined using a common notation.

Non-prescriptive black-box testing approaches like Exploratory Testing and Error Guessing have a ***reliance on domain knowledge*** that is usually implicit and not specified (see Section 2.2.5). During non-prescriptive testing, novice testers may not be aware of the need to consider relevant domain knowledge and may also be unfamiliar with the "application domain[7]" (Reed 1990) of the program under test. As a result, they may produce incomplete and ineffective test sets. Also, there are currently no systematic procedures for capturing the definition of new test case design rules that are used during non-prescriptive testing. As a result, detected failures may not be reproducible and auditing issues may arise when the completeness or efficiency of non-prescriptive testing cannot be determined or proven to relevant stakeholders. The existence of prescriptive procedures (with consistent terminology) for documenting new test case design rules could facilitate improvement of the domain knowledge and skills of both novice and experienced testers as well as an improvement in test effectiveness.

A related issue is that when specifications of program input and output domains are unavailable, test case design is often based on the domain knowledge of expert testers who understand the program's application domain. Experienced testers may be capable of conducting efficient and effective black-box testing solely from their own unique domain knowledge and experience (see Section 2.6.2) and may do so without documenting their decisions. As a result, novice testers may not be able to learn from their more experienced peers if the test case design rules they use and the assumptions they make about the definitions of program input and output fields are not documented.

1.5 Aims and Contributions

The primary aim of this thesis is to explore how these seven problems affect the usability and failure-detection effectiveness of prescriptive and non-prescriptive approaches to black-box testing, and to present a new representation for describing black-box testing methods called the Atomic Rules approach (Murnane, Hall & Reed 2005), which aims to resolve these problems and improve test method usability and failure-

[7] Reed proposed a Knowledge Acquisition Based Approach to Software Project Planning (KABASPP) approach, which defines five knowledge areas from which domain knowledge can be obtained: application domain, application solution domain, development environment domain, run time environment domain and the managerial domain (Reed 1990). Each consists of knowledge concepts that are used in carrying out particular tasks on a software development projects.

detection effectiveness. In the Atomic Rules approach, black-box testing methods are decomposed into individual test case design rules called 'Atomic Rules.' Each Atomic Rule is defined in characterisation schema (Table 1-3), which ensures that each test case design rule is defined in a prescriptive and uniform notation. The following four-step test case design process allows each black-box testing method to be defined in a uniform notation (Figure 1-4):

1. Partition the input and output domains of the program into sets of equivalent data, by applying Data-Set Selection Rules (DSSRs).
2. Select test data values from each partition by applying Data-Item Selection Rules (DISRs).
3. Optionally manipulate[8] the test data values by applying Data-Item Manipulation Rules (DIMRs).
4. Construct test cases by creating combinations of test data values through the application of Test Case Construction Rules (TCCRs).

The aim of the four-step test case design process (Figure 1-4) and the Atomic Rules schema (Table 1-3) is to ensure that black-box testing methods are defined using a consistent, prescriptive notation.

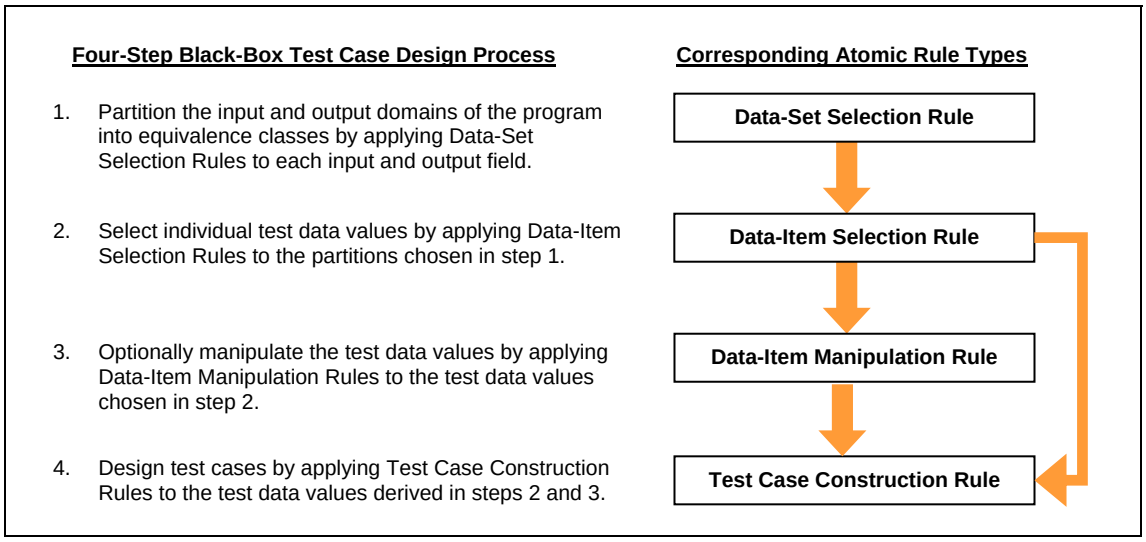
Table 1-3: Example of an 'Atomic Rule.'

Attribute	Values
Test Method	Equivalence Partitioning
Number	EP1
Identifier	LLBS
Name	Less than Lower Boundary Selection
Description	Select an equivalence class containing values below the lower boundary of a field
Source	(Myers 1979)
Rule Type	DSSR
Set Type	Range
Valid or Invalid	Invalid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric[9]
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	1
# Tests Derived	0

[8] The definition of Syntax Testing in (BS 7925-2) includes a test case design rule that 'mutates' test data values. In this thesis the word 'manipulate' (instead of 'mutate') is used to describe any test case design rule that derives an invalid test data value by altering a valid test data value (e.g. by removing a character from the end of a valid keyword).

[9] Atomic Rules *EP1* to *EP3* can be applied to any datatype that contains contiguous data. Alpha and non-alphanumeric can be considered contiguous if the ASCII table is used to identify values outside the valid boundaries.

Figure 1-4: The four-step black-box test case design process.



The Atomic Rules approach aims to resolve the seven problems with existing black-box testing methods as follows. The Atomic Rules schema and four-step test case design process were defined by identifying the common attributes of eleven different black-box testing methods that were described differently in fifteen different places (see Chapter 3, Section 3.2). This results in a uniform notation that eliminates ***notational and terminological differences*** between black-box testing methods and facilitates more prescriptive comparisons between methods. The approach allows ***multiple versions*** of each method to be combined to create one set of Atomic Rules. This also resolves ***method overlap*** by locating and eliminating redundant rules that appear in more than one method, or more than once within a method.

This also makes the methods ***easier to audit***; by providing one prescriptive definition of each black-box testing method, the Atomic Rules approach simplifies and disambiguates the process of auditing black-box test set completeness. Test cases that are submitted for audit can be checked for conformance against the Atomic Rules definition of specific black-box testing methods, by comparing the set of Atomic Rules from each method to those that were (or were not) applied to derive the test cases (see Chapter 3, Section 3.5). Standardisation organisations such as ISO could use the Atomic Rules definition of black-box testing methods to standardise their definitions.

Definition by exclusion and ***reliance on domain knowledge*** are resolved by defining explicit 'datatypes' (e.g. integer, real, alpha, non-alphanumeric) for use in the Atomic Rules characterisation schema, enabling definition of the 'universe of discourse' for program inputs. Individual Atomic Rules are then defined for EP, allowing the input domain to be partitioned by datatype, eliminating the need for testers to identify invalid partitions ad hoc. This also makes the methods ***easier to automate*** (see Chapter 4).

Atomic Rules from methods like EP, BVA and ST can also be used to support the use of other black-box testing methods, such as State Transition Testing, Use Case Testing and the Category Partition Method (see Chapter 3, Section 3.6).

Two approaches for supporting Atomic Rules are also presented in this thesis: *Systematic Method Tailoring* (SMT) and *Goal/Question/Answer/Specify/Verify* (GQASV) (Murnane, Reed & Hall 2006).

Systematic Method Tailoring facilitates the customisation of black-box testing (see Chapter 3, Section 3.10) through the creation of new Atomic Rules and new black-box testing methods. SMT also provides testers with a prescriptive procedure and notation for documenting non-prescriptive test case design rules, allowing them to be shared with other (novice or experienced) testers. This reduces the inherent ***reliance on domain knowledge*** of non-prescriptive testing approaches like Error Guessing and Exploratory Testing.

Goal/Question/Answer/Specify/Verify is a new specification elicitation procedure that guides testers in the resolution of undefined or poorly defined input and output fields, as well as facilitating the documentation of domain knowledge that is utilised during the specification process (see Chapter 3, Section 3.10). This reduces ***reliance on domain knowledge*** by ensuring specification-based domain knowledge is thoroughly documented and facilitates more effective black-box testing.

An additional contribution of this thesis is the definition of 'test method usability' and the identification of measurement and assessment approaches for evaluating test method usability (see Section 1.2).

1.6 Scope

This research was initially focussed on black-box testing methods that can be used to partition the input and output domains of a program, select data values from each partition and construct test cases (i.e. steps 1, 2 and 4 of the four-step test case design process, see Figure 1-4). As such, development of the Atomic Rules approach began with an investigation into possible representations for test case design rules from two of the most well-known black-box testing methods, Equivalence Partitioning and Boundary Value Analysis.

This was followed by the execution of two university-based experiments, which investigated whether the Atomic Rules representation of EP and BVA improved the usability of these methods (Section 1.7). The research was then extended to include Syntax Testing and Random Testing (RT). In a subsequent industrial experiment, the investigation was further extended to determine whether test case design rules utilised by experienced testers could be represented as Atomic Rules, which covered Error Guessing (EG) and Exploratory Testing (ET).

The investigation was then extended to determine whether combinatorial testing methods All Combinations, Each Choice, Base Choice, Orthogonal Array Testing and Specification-Based Mutation Testing could be represented as Atomic Rules. Whilst these methods typically consist of algorithms for test case construction that go beyond simple test data design, it is possible to represent them in a similar way to methods like EP. Thus, these methods are also within scope.

Test Matrices, Test Catalogues, Test Categories and the Category Partition Method (CPM) are also within scope, as they can support test case analysis and design by mapping Atomic Rules from EP, BVA and ST that have been applied to each program under test. Use Case Testing and State Transition Testing are within scope as they can be improved by utilising Atomic Rules from EP, BVA and ST.

Cause-Effect Graphing and Decision Tables (Myers 1979) are outside the scope of this research, as they test relationships between inputs and outputs and between combinations of inputs. Thus, they do not provide specific guidelines for input/output field partitioning or test data generation.

1.7 Evaluation

Three separate experiments were conducted to evaluate whether the Atomic Rules approach improves the usability and failure-detection effectiveness of existing black-box testing methods. The experiments formed the core evaluation of the Atomic Rules approach.

The first two were classroom experiments, which were carried out over two consecutive years with students from La Trobe University in Melbourne, Australia (see Chapter 5). The aim was to compare the usability of the Atomic Rules representation of EP and BVA to that of Myers' original definitions of these methods, to determine which enables novice testers to write more complete and correct black-box test sets.

An industrial experiment was then carried out to compare the usability and failure-detection effectiveness of black-box testing methods that are used by professional testers as part of their jobs, to that of the Atomic Rules representation of EP, BVA and ST (see Chapter 6). The participants were working for a large government organisation in Brisbane, Australia, and had software testing experience ranging from novice to expert.

A proof-of-concept evaluation of GQASV and SMT was also carried out against a real-world application (see Chapter 3). Automation of the Atomic Rules approach, GQASV and SMT was another interesting avenue for investigation. To this end, a prototype called the Atomic Rules Testing Tool (ARTT) has been developed (see Chapter 4). ARTT implements the Atomic Rules definition of EP, BVA and ST to enable automatic generation of black-box test data. ARTT demonstrates that the Atomic Rules approach defines black-box testing methods precisely enough to enable automation.

1.8 Thesis Structure

This thesis is structured as follows. A survey of relevant literature is provided in Chapter 2, including an extensive analysis of existing black-box test method definitions. The Atomic Rules approach is introduced in Chapter 3, including the Atomic Rules schema and four-step test case design process, as well as the supporting approaches GQASV and SMT. The Atomic Rules Testing Tool is presented in Chapter 4. This is followed by presentation of the two university-based experiments in Chapter 5 and the industrial experiment in Chapter 6. Conclusions and future work are discussed in Chapter 7.

Chapter 2

Black-Box Testing – History and Practice

"If I have seen further than you and Descartes it is by standing upon the shoulders of giants."
Sir Isaac Newton in a letter to Robert Hooke, 1675/1976.

2.1 Overview

ISO/IEC define the term *test case* as "a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to execute a particular program path or to verify compliance with a specific requirement" (ISO/IEC 24765:2008) (see Section 1.1). Black-box testing methods provide guidance on the design of test cases that can be used to verify that programs meet their requirements (i.e. that they *comply with a specific requirement*), whether those requirements are explicitly documented or implicitly known by individual testers on the testing team. This can include testing that a program accepts and processes 'valid' inputs correctly as specified and produces the correct output, or that it rejects 'invalid' inputs and displays appropriate error messages as required. Black-box test cases can be designed from business requirements, functional specifications, technical specifications and design specifications (Burnstein 2003) and can also be based on the unique domain knowledge and experience of each tester.

As was pointed out in Chapter 1, the usability and failure-detection effectiveness of black-box testing methods is currently affected by seven problems with existing definitions of these methods. In this chapter, a review of relevant literature is presented to explore these seven problems. This review includes the following black-box testing methods:

1. Equivalence Partitioning (Section 2.2.1);
2. Boundary Value Analysis (Section 2.2.2);
3. Syntax Testing (Section 2.2.3);
4. Random Testing (Section 2.2.4);
5. Non-prescriptive approaches to black-box testing (Section 2.2.5):
    - Error Guessing (Section 2.2.5.1); and
    - Exploratory Testing (Section 2.2.5.2);
6. Test Catalogues, Test Categories and Test Matrices (Section 2.2.6);
7. Combinatorial test methods (Section 2.2.7) including:

- All Combinations (Section 2.2.7.1);
- Each Choice (Section 2.2.7.2);
- Base Choice (Section 2.2.7.3);
- Orthogonal Array Testing (Section 2.2.7.4) and
- Specification-Based Mutation Testing (Section 2.2.7.5);

8. Category Partition Method (Section 2.2.8) and
9. Classification Trees (Section 2.2.9).

The remainder of this chapter is structured as follows. To support the literature review, terms that are used throughout the chapter are discussed in Section 2.1.1, followed by approaches to specifying program requirements and classes of program input and output fields in Section 2.1.2. A review of relevant literature begins with the definition of a common notation for reviewing the methods in Section 2.2, followed by a detailed review of each test method in Sections 2.2, 2.3 and 2.4. Approaches to selecting test methods are discussed in Section 2.5. Factors that can influence test effectiveness are discussed in Section 2.6. Automation of black-box testing is discussed in Section 2.7. A chapter summary is provided in Section 2.8.

2.1.1 Terminology

To support this literature review, definitions and explanations of the following terms are provided:

1. fault;
2. error;
3. failure;
4. test case;
5. test data value (also called ‘test input’);
6. input field;
7. expected result;
8. test case design rule;
9. test procedure;
10. test script; and
11. test condition.

ISO/IEC define the terms *error* as “a human action that produces an incorrect result, such as software containing a fault” and *fault* as “a manifestation of an error in software” (ISO/IEC 24765:2008). Thus, an error is a mistake a programmer makes when interpreting a requirement, resulting in the implementation of *faulty* program source code that does not meet end-user requirements. ISO/IEC define the term *failure* as

"an event in which a system or system component does not perform a required function within specified limits" (ISO/IEC 24765:2008). The execution of a black-box a test case against a program can cause one or more faults to manifest, thereby resulting in program failure.

As stated earlier, the term *test case* is defined by ISO/IEC as "a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to execute a particular program path or to verify compliance with a specific requirement" (ISO/IEC 24765:2008). Therefore, test case design requires the selection of test inputs, which are referred to as 'test data values' in this thesis to differentiate them from the 'input fields' of a program that they are designed to test. *Test data values* can be valid or invalid, according to what the program should accept or reject respectively. For example, an *input field* 'vehicle type' could be defined in Backus-Naur Form (BNF) (Knuth 1964) as *<vehicle_type> ::= [Car | Truck | Motorbike]*. A *valid test data value* 'Truck' could be selected to check that the program accepts valid input data, while an *invalid test data value* 'Bicycle' could be used to test that the program (correctly) rejects invalid input data.

Each test case includes one or more *test data values*, one for each input field covered by the test case. For example, if a test case covers a component that accepts two inputs, *<vehicle_type> ::= [Car | Truck | Motorbike]* and *<max_speed> ::= [120 – 240]*, a test case could consist of test data values 'Car' and '240'.

Test cases also include *expected results* that define the predicated behaviour of the program under test when it is executed with a particular test case (BS 7925-1). *Expected results* can be identified from program documentation or can be based on the domain knowledge and experience of program testers. For example, the expected result of a test case that consists of only valid test data values would explain what 'correct' behaviour the program should exhibit in response to the valid input, while a test case containing any invalid test data values would explain how the program should reject the input and (ideally) what error message should be displayed, describing why the input was rejected.

Test case design rules are the individual guidelines of black-box testing methods that can be used to design test cases. For example, Myers (1979) defines the following guideline for Equivalence Partitioning, which consists of a number of test case design rules, as follows:

> **"If an input condition specifies a range of values (e.g. "the item count can be from 1 to 999"), identify one valid equivalence class (1 ≤ item count ≤ 999) and two invalid equivalence classes (item count < 1 and item count > 999)."**

This guideline can be decomposed into three different *test case design rules*:

1. item count from lower boundary to upper boundary (1 ≤ item count ≤ 999);
2. item count less than lower boundary (item count < 1); and
3. item count greater than upper boundary (item count > 999).

The terms *test procedure* and *test script* are also used in software testing literature. A standard definition of a *test procedure* is "detailed instructions for the setup, execution, and evaluation of results for a given test case" (IEEE 1012:2004). *Test script* is a similar term that is "commonly used to refer to the automated test procedure used with a test harness" (BS 7925-1).

Some descriptions of black-box testing methods discuss the identification of *test conditions* (also called 'test objectives' (Craig & Jaskiel 2002)) prior to test case design, where each test case is specifically derived to cover at least one test condition. A number of definitions of the term *test condition* are cited in the literature, as follows.

- The International Software Testing Qualifications Board (ISTQB) define test condition as "an item or event of a component or system that could be verified by one or more test cases, e.g. a function, transition, feature, quality attribute, or structural element" (ISTQB 2005).
- Kent (2008) describes test conditions as refinements to requirements that define the expected behaviour of a system.
- Craig and Jaskiel (2002) consider test conditions to be categories of things that need to be tested, which are sets of high level and low level requirements.
- Both Fewster and Graham (2000) and Dustin (2003) consider equivalence classes and boundary values to be test conditions (identified by EP and BVA respectively);

In this thesis, *test conditions* are considered to be refinements of program requirements (as argued by Kent (2008)). Requirements and test conditions can be defined at high and low levels of detail. High-level requirements can also be converted into test conditions that define requirements in more detail. For example, the following high-level functional requirement could be specified in a 'business requirement specification,' which typically specifies program requirements at relatively high levels of detail:

> **Requirement 1:** the program must add two numbers together.

This defines the requirement at a very high level of detail[1]. This requirement could be converted directly into a high-level test condition, as follows:

> **Test Condition 1:** check that the program can add two numbers together correctly.

This requirement could be converted into a test condition that is specified at a low level of detail:

> **Test Condition 2:** check that the program can take two signed 16-bit integers in the range −32,768 to +32,767 as input, sum them together and output the result as a 32-bit integer to the screen, in a field that has a maximum range of −2,147,483,648 to +2,147,483,647. Any other input should be rejected, with a corresponding error message being displayed to the user.

[1] In Reed's KABASPP model (1990), this would be equivalent to defining a requirement in the program application domain.

Test condition 2 defines this requirement much more precisely[2]. On the other hand, test conditions can sometimes only be specified in this way if testers are aware of design decisions made by developers; for example, this could be identified by reviewing design specifications or by speaking with developers.

Further definitions of terms used throughout this thesis can be found in the glossary (see page xvi).

2.1.2 Classes of Input and Output Fields

Program input and output fields are typically expressed in the literature in two ways: as lists and ranges (these terms were adapted from similar concepts in (Jorgensen 1995, Lewis 2000, Mosley 1993, Myers 1979, Page et al. 2009)).

Lists are typically specified in the literature in one of two ways, as follows:

$$L ::= [v_1 \mid v_2 \mid \ldots \mid v_n] \tag{2.1}$$

$$L ::= [v_1, v_2, \ldots, v_n] \tag{2.2}$$

where *n* is the number of *v* values contained in list *L*.

Ranges are also commonly specified in the literature in one of two ways, as follows:

$$\{R : lb \leq R \leq ub\} \tag{2.3}$$

$$R ::= [lb - ub] \tag{2.4}$$

which defines a range of values from lower boundary *lb* to upper boundary *ub*.

Lists and ranges can also repeat and be optional or mandatory. *Repetition* can be specified by appending a superscript *lb – ub* to the specification of a list or a range, where *lb* and *ub* indicate the minimum and maximum times the field can repeat respectively. List repetition can be specified as:

$$L ::= [v_1 \mid v_2 \mid \ldots \mid v_n]^{lb-ub} \tag{2.5}$$

$$L ::= [v_1, v_2, \ldots, v_n]^{lb-ub} \tag{2.6}$$

Range repetition can be specified as:

$$\{R : lb \leq R \leq ub\}^{lb-ub} \tag{2.7}$$

$$R ::= [lb - ub]^{lb-ub} \tag{2.8}$$

where the superscript $^{lb-ub}$ indicates that the field that can repeat from *lb* (lower boundary) to *ub* (upper boundary) times.

[2] In Reed's KABASPP model, this defines a requirement within the program's run-time environment domain (Reed 1990).

The same notation can be used to indicate if a field is *optional* or *mandatory*. Using the notation above, *optional* ranges and lists can be specified as:

$$\{R : lb \leq R \leq ub\}^{0-1} \quad (2.9)$$

$$\mathrm{R} ::= [lb - ub]^{0-1} \quad (2.10)$$

$$L ::= [v_1 \mid v_2 \mid \ldots \mid v_n]^{0-1} \quad (2.11)$$

$$L ::= [v_1, v_2, \ldots, v_n]^{0-1} \quad (2.12)$$

where the superscript $^{0-1}$ indicates that the fields can appear zero or one times.

Similarly, a *mandatory* lists and ranges can be specified as follows:

$$\{R : lb \leq R \leq ub\}^{1} \quad (2.13)$$

$$\mathrm{R} ::= [lb - ub]^{1} \quad (2.14)$$

$$L ::= [v_1 \mid v_2 \mid \ldots \mid v_n]^{1} \quad (2.15)$$

$$L ::= [v_1, v_2, \ldots, v_n]^{1} \quad (2.16)$$

where the superscript ^{1} indicates that the field must appear exactly once.

In the absence of a superscript element (see expressions 2.1 to 2.4 above), it is assumed that the field is mandatory and must appear exactly once.

2.1.2.1 Examples

Consider the following examples. In Section 2.1.1, the following test condition was defined:

Test Condition 2: check that the program can take two signed 16-bit integers in the range −32,768 to +32,767 as input, sum them together and output the result as a 32-bit integer to the screen, in a field that has a maximum range of −2,147,483,648 to +2,147,483,647.

This test condition could give rise to the definition of two range-based fields, as follows:

$$\langle input\_integer \rangle ::= [-32{,}768 - 32{,}767]^{2}$$

$$\langle output\_integer \rangle ::= [-2{,}147{,}483{,}648 - 2{,}147{,}483{,}647]$$

This indicates that exactly two 16-bit integers must be input, while one 32-bit integer will be output.

Consider another example involving a list, in which the following test condition was initially defined:

Test Condition 3: check that the user can select between one and seven passenger vehicle types from the following list: boat, car, motorbike, plane, tractor, train, truck.

This test condition would give rise to one list-based field:

<vehicle> ::= [boat | car | motorbike | plane | tractor | train | truck][1-7]

2.2 Black-Box Testing Methods

Before reviewing relevant literature, it is first necessary to define the notation used when reviewing their common elements. As introduced in Section 1.4, black-box testing method typically focuses on just one of the following four key steps of black-box test case design:

1. Partition the input and output domains of the program into sets of equivalent data, by applying Data-Set Selection Rules (e.g. see Equivalence Partitioning in Section 2.2.1).
2. Select test data values from each partition by applying Data-Item Selection Rules (e.g. see Boundary Value Analysis in Section 2.2.2).
3. Optionally manipulate[3] the test data values by applying Data-Item Manipulation Rules (e.g. see Syntax Testing in Section 2.2.3).
4. Construct test cases by creating combinations of test data values via the application of Test Case Construction Rules (e.g. see Orthogonal Array Testing in Section 2.2.7.4).

These four steps are utilised throughout this chapter to describe the 'functionality' of the individual test case design rules of each black-box testing method.

Program specifications often do not provide sufficient information to allow a tester to partition the input and output domains of a program into valid and invalid equivalence classes (covered by step 1 above), in which case the tester is usually required to create adequately detailed field definitions prior to (or as a part of) the partitioning process. For example, in Section 2.1.1, *test condition 2* provides sufficient information to allow a tester to partition the input and output domains of the program. Conversely, *test condition 1* requires the tester to make assumptions about the minimum and maximum values that are allowable in each field, which may result in test cases that do not meet user requirements. Since black-box testing methods do not usually provide guidance on how to obtain and verify the correctness of detailed definitions of program input and output fields, a new approach called Goal/Question/Answer/Specify/Verify is introduced (see Chapter 3), which can be used to elicit and record such information.

2.2.1 Equivalence Partitioning (EP)

During Equivalence Partitioning, the input and output domains of the program under test are 'partitioned' into disjoint, mutually-exclusive sets of 'equivalent' data, in the sense that if one element in a set detects a program fault, then all elements in the set should find the same fault (Myers 1979) and execute the same program paths (Howden 1976). When a program is executed with any one test data value from an equivalence class, it is assumed that the program does not have to be tested against any other values from

[3] For example, the definition of Syntax Testing in (BS 7925-2) includes a test case design rule that 'mutates' test data values. In this thesis the word 'manipulate' is used to describe any test case design rule that derives an invalid test data value by altering a valid test data value (e.g. by removing a character from the end of a valid keyword).

that partition because its white-box behaviour is presumed to be identical for all values from that set (Myers 1979). Similarly, if any member of an equivalence class causes program failure then it is expected that all other elements from that class will cause the same failure (Hamlet & Taylor 1990). Such partitions are called "revealing" (Weyuker & Ostrand 1980) or "homogenous" (Hamlet & Taylor 1990). Partitioning is claimed to reduce the number of test cases that are required to achieve predefined test coverage goals by covering a large subset of all possible tests with the smallest number of test cases (Myers 1979).

EP has been called the "most basic" black-box testing method because it guides testers in the design of a compact test sets that achieve adequate coverage (Copeland 2004). EP's guidelines for partitioning the program input and output domains "prescribe" test case design. They can also act as a point of reference for measuring test coverage (Grindal, Offutt & Andler 2005). EP can be applied to high and low-level specifications (Richardson & Clarke 1985) and can be white-box, black-box or grey-box (Hamlet & Taylor 1990). Grey-box partitioning approaches were published as early as 1976 (Howden 1976), while the first purely black-box partitioning method was defined by Myers' EP in 1979 (Myers 1979)[4]. Many authors still cite Myers' definition (e.g. see (Burnstein 2003, Jorgensen 1995, Kit 1995, Mosley 1993, Copeland 2004, Page et al. 2009, Mosley & Posey 2002, Parrington & Roper 1989)).

Myers (1979) defined eight guidelines for EP (Figure 2-1).

[4] Kaner (1988) advocates Myers textbook as "the best in print" and, as further testament to its popularity, Myers' text was reprinted in 2004, twenty-five years after its first publication (Myers 2004).

Figure 2-1: Myers' (1979) guidelines for Equivalence Partitioning.

Myers' (1979) guidelines for Equivalence Partitioning[5]

Guidelines for equivalence class design

1. If an input condition specifies a range of values (e.g. "the item count can be from 1 to 999"), identify one valid equivalence class ($1 \leq$ item count ≤ 999) and two invalid equivalence classes (item count < 1 and item count > 999).
2. If an input condition specifies a number of values (e.g. "one through six owners can be listed for the automobile"), identify one valid class and two invalid equivalence classes (no owners and more than six owners).
3. If the input condition specifies a set of values and there is reason to believe that each is handled differently by the program (e.g. "type of vehicle must be BUS, TRUCK, TAXICAB, PASSENGER, or MOTORCYCLE"), identify a valid equivalence class for each one and one invalid equivalence class (e.g. "TRAILER").
4. If an input condition specifies a "must be" situation (e.g. "first character of the identifier must be a letter"), identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter).
5. If there is reason to believe that elements in an equivalence class are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes.

Guidelines for test case design

6. Assign a unique number to each equivalence class.
7. Until all valid equivalence classes have been covered by (incorporated into) test cases, write a new test case covering as many of the uncovered valid equivalence classes as possible.
8. Until all invalid equivalence classes have been covered by test cases, write a test case that covers one, and only one, of the uncovered equivalence classes.

The following interpretation and examples are provided to critique Myers' definition of EP.

Guideline 1. This defines three Data-Set Selection Rules for partitioning range-based fields. The field in this example can be restated as *<item_count> ::= [1 – 999]*. For this field, Myers proposes the use of three DSSRs, as follows.

1. Select a partition containing valid values; for the field in this example, this would select partition *<valid_item_count> ::= [1 – 999]*. This DSSR is defined prescriptively and does not require further clarification.
2. Select an equivalence class containing values that lie below the lower boundary of the field, which would select the set *item_count < 1*. One problem is that this DSSR does not state that a minimum boundary value must be chosen for the partition, which might only be known through grey-box information. For example, if the input was processed as a signed 32-bit integer, this partition could be defined as *<invalid_item_count_1> ::= [−32,768 – 1]*.
3. Select an equivalence class containing values that lie above the upper boundary of the field, selecting the set *item_count > 999*. Similar to rule 1, this DSSR does not indicate

[5] These guidelines are expressed in Myers' exact words.

that a maximum value must be chosen for this partition. For example, the partition could be defined as *<invalid_item_count_2> ::= [1000 – 32,767]*.

Guideline 2. This also defines three DSSRs for partitioning range-based fields, which are essentially the same as those described under guideline 1 and, as such, they do not provide any new guidance on partitioning. The field in this example can be restated as *<owner_count> ::= [1 – 6]*. The DSSRs applied to this field are as follows.

4. Select a partition containing only valid values, which can be specified as *<valid_owner_count> ::= [1 – 6]*. This could be covered by guideline 1, rule 1.

5. Select an equivalence class containing one value *owner_count = 0*. This could be derived more prescriptively by applying guideline 1, rule 2, selecting partition *<invalid_owner_count_1> ::= [−32,768 – 0]* (assuming 32-bit integers), followed by a Data-Item Selection Rule from BVA that selects the upper boundary of the partition, which would select the value *0*.

6. Select an equivalence class containing values in the set *owner_count > 6*, which could be covered by guideline 1, rule 3, resulting in partition *<invalid_owner_count_2> ::= [7 – 32,767]* (assuming 32-bit integers).

Guideline 3. This defines two DSSRs for partitioning list-based fields, where each valid input in the field is inserted into a separate partition (which is likely only known through grey-box information) and where identification of the invalid class requires domain knowledge. The field could be redefined as *<vehicle> ::= [BUS | TRUCK | TAXICAB | PASSENGER | MOTORCYCLE]*, while the guideline utilises two DSSRs, as follows.

7. Select a separate equivalence class for each vehicle type, resulting in partitions *<vehicle1> ::= [BUS]*, *<vehicle2> = [TRUCK]*, *<vehicle3> = [TAXICAB], <vehicle4> = [PASSENGER]*, *<vehicle5> = [MOTORCYCLE]*.

8. Select an equivalence class containing vehicle types not in the valid set. For this example, this could include values like *CYCLE*, *TAXI*, *TRAILER*, *TRICYCLE* and *SCOOTER*. More prescriptive testing could be achieved by defining DSSRs that select invalid partitions by datatype, including numeric and non-alphanumeric (i.e. special) characters at a minimum (e.g. see Chapter 3, Section 3.1.1.1).

Guideline 4. This defines two DSSRs that partition a field that could be treated as a list or range, depending on how the program processes inputs (which is grey-box information). For example, the input field in this example could be specified either as a list defined as *<letter_list> ::= [A | B | C | ... | Y | Z | a | b | c | ... | y | z]* or a range defined as *<letter_range> ::= [ascii(A) – ascii(Z) | ascii(a) – ascii(z)]* (assuming the ASCII table was used for partitioning). The guideline then recommends two DSSRs, as follows.

9. Select an equivalence class containing letters. Depending on the tester's assumption, this would either derive a list-based partition *<valid_letter_list> ::= [A | B | C | ... | Y | Z | a |*

b | c | ... | y | z] or a range-based partition *<valid_letter_range> ::= [ascii(65) – ascii(90) | ascii(97) – ascii(122)]*.

10. Select an equivalence class containing anything other than letters, which is ambiguous, but would ideally include numeric and non-alphanumeric characters at a minimum (e.g. see Chapter 3, Section 3.1.1.1). For example, if the ASCII table was used, depending on whether the field was treated as a list or range, this could result in a list-based partition *<invalid_letter_list> ::= [space | ! | " | ... | > | ? | @ | [| \ | \ | ^ | _ | ` | { | | | } | ~]* (where *space* represents the space character) or a range-based partition *<invalid_letter_range> ::= [space – @ | [ – ` | { – ~]*.

Guideline 5. This guideline recommends further division of any partition that contains a subset of characters that are suspected to be handled differently by the program. Thus, similar to guidelines three and four, it suggests partitioning based on grey-box information.

Guideline 6. This guideline recommends assignment of a unique identifier to each equivalence class, to enable traceability from partitions to test cases.

Guideline 7. This defines one Test Case Construction Rule (TCCR) that works by assigning as many input fields per test case with a valid value from a valid partition. This TCCR is repeatedly applied until all valid partitions have been 'covered' by at least one test case.

Guideline 8. This defines one TCCR that works by assigning one input field per test case with an invalid value from any invalid partition derived for that field, whilst assigning all other input fields a valid value. This TCCR is repeatedly applied until all invalid partitions have been 'covered' by at least one test case.

One problem with Myers' partitioning guidelines is that they lack precision (Ostrand & Balcer 1988) and have been referred to as "testing heuristics" (i.e. rules of thumb) (DeMillo et al. 1987), as the individual knowledge and experience of each tester can affect the completeness of resulting test sets. Other definitions of EP have improved on Myers' original definition (e.g. see (Tamres 2002, Kaner 1988, BS 7925-2, Craig & Jaskiel 2002), which are discussed below). Despite this, all publications of EP suffer from (at least) five problems: definition by exclusion, reliance on domain knowledge, multiple versions, difficult to audit and difficult to automate.

As introduced in Chapter 1, Myers' fourth guideline describes a "must-be" condition (also called a "Boolean condition" (Pressman 1992)) (see Figure 2-1) in which an invalid equivalence class is selected containing all inputs other than those in the valid class. This DSSR provides little guidance as to the contents of the invalid class. Ideally, a member from every class of data (e.g. integer, real, alpha, non-alphanumeric/special characters) would be represented in the invalid class, but a novice tester may only be aware of a subset of these. Thus, ***definition by exclusion*** assumes familiarity with the 'universe of discourse' with respect to program inputs, and reduces the learnability and operability of EP (Murnane, Reed & Hall 2006). As a result, different testers using this defining of EP may produce vastly dissimilar test sets from the same program specification (Patton 2006). Furthermore, the inherent ambiguity in this method

also makes it ***difficult to automate*** this method. On the other hand, Patton considers Myers' guidelines to be acceptable, as long as the coverage of each equivalence class is assessed through peer review, adding that EP is "science, but it's also art" (Patton 2006). Clearly, Myers' version of EP relies on the domain knowledge of the testers involved, reducing the repeatability and predictability of black-box test case design and resulting in test case design procedures that are not automatable. Myers' definition of EP cannot be considered complete, since it is not guaranteed to result in reproducible test sets.

Definition by exclusion is partially resolved by Tamres' (2002) definition of EP, in which the universe of discourse is defined through 'datatype' and 'data set' definitions (Abbott (1986) describes this as "data-oriented" testing) that apply to each Data-Set Selection Rule. Tamres explains the approach using a login screen consisting of input fields 'username' and 'password' and buttons 'OK' and 'Cancel.' For the input fields, Tamres described the use of DSSRs that define valid partitions by datatype, including datatypes *lowercase alpha*, *uppercase alpha*, *numeric* and *non-alphanumeric*, and a DSSR to select valid partitions including *[o, O]* and *[c, C]* as data that activates the OK and Cancel buttons respectively from the keyboard. Tamres also used DSSRs to identify invalid non-alphanumeric partitions *[-, _, $, !, &, ~]* for username and *[@, #, %, *, ^, ", (,), [,], {, }, /, ?, <, ?]* for password.

While Tamres used DSSRs to define the contents of valid and invalid equivalence classes by datatype and by test data values in significantly more detail than Myers, Tamres' definition of EP is incomplete. The *non-alphanumeric* class should ideally classify all non-alphanumeric character from the ASCII table as either valid or invalid; in Tamres' example, characters from the set *[+ = | : ; ' , < > .]* are not specifically included in the equivalence classes defined for username and password. Tamres' also did not include a guideline for selecting greater than the maximum number of characters for an input field, which is covered by Myers' first guideline (Figure 2-1). Furthermore, while Tamres' definition provides examples of the types of equivalence classes that could be selected for one example login screen, it did not include generic guidelines like those defined by Myers. Consequently, Tamres' definition of EP could cause omission of important equivalence classes, which would likely reduce the effectiveness of black-box testing.

Definition by exclusion for EP is also partially resolved by Kaner (1988), who utilises the ASCII table to identify invalid equivalence classes for a set of example fields. Kaner's definition of EP covers DSSRs that partition input fields consisting of uppercase and lowercase *alphas*, for which three invalid equivalence classes are defined as follows:

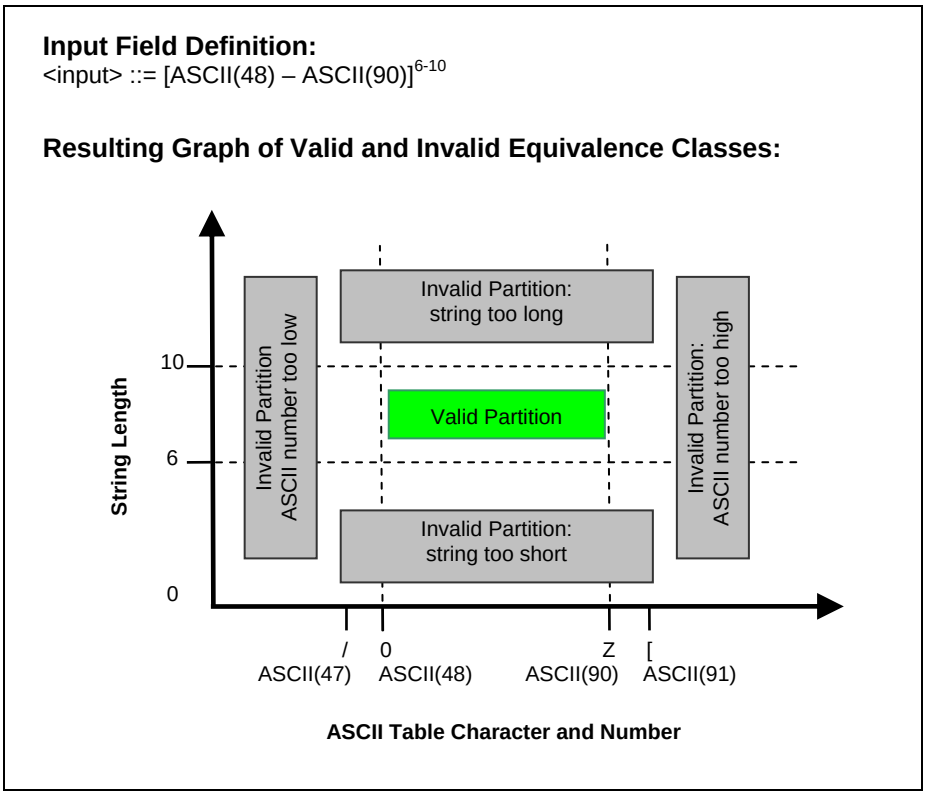
1. ASCII code below that for 'A';
2. ASCII code between the codes for 'Z' and 'a'; and
3. ASCII code greater than the code for 'z'.

Other than the *alpha* datatype, Kaner's definition of EP does not define DSSRs that cover other ASCII datatypes, such as *integer*, *special character* and *control character*. For some programs, particularly those with command line interfaces, test data values from each class could be treated differently. For example, *ctrl^Z* may cause program termination in UNIX environments, while very large integer inputs could cause

program failure if they are not properly handled by the program. The identification of invalid datatypes from character encoding sets becomes further complicated when the input domain is the Unicode table, which currently contains over 100,000 individual characters (Wikipedia Unicode 2008), whereas the ASCII table contains only 94 printable characters (Oualline 2003), which is a more manageable size. Also, like Tamres, Kaner demonstrates the selection of equivalence classes for example fields and as such, he does not provide generic guidelines that apply to any field type. Kaner's guidelines are also presented across nine sub-sections and two tables, which could make it difficult for a tester to identify whether or not they have derived all required equivalence classes.

Definition by exclusion in EP is also partially resolved by Black, who explains the method through illustrative examples that utilise various DSSRs, which partition the input domain by ASCII datatype, including *integer*, *real*, *character*, *string*, *date*, *time* and *currency* (Black 2007). While Black's definition of EP is no more prescriptive than Tamres or Kaner, it does introduce a unique approach to graphically representing partitions (Figure 2-2). In the example below (which was adapted from (Black 2007)), a valid input field is defined as any string of 6 to 10 ASCII characters situated between *ASCII(48)* (the number *0*) and *ASCII(90)* (the letter *Z*), which can be represented as $<input> ::= [ASCII(48) - ASCII(90)]^{6\text{-}10}$. From this, any character below *ASCII(48)* or above *ASCII(90)* is regarded as invalid, while any string less than 6 characters or greater than 10 characters in length is also considered invalid (Figure 2-2).

Figure 2-2: Example graphical representation of equivalence classes (adapted from (Black 2007)).



The British Standards Institute's (BSI's) Component Testing Standard (BS 7925-2) also partially solves ***definition by exclusion*** by defining DSSRs that select invalid partitions by datatype. For example, for the

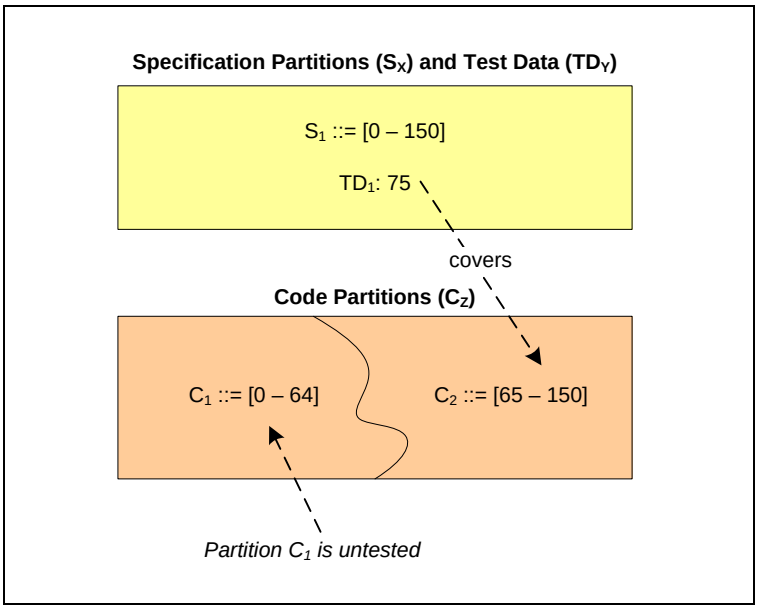
valid integer range *[0 – 75)*, four invalid classes are identified: *> 75*, *< 0*, *real number* and *alphabetic*. However, this definition is missing DSSRs that select *special characters* and *control characters*. The DSSRs in the standard are also not generic, as they are only explained for two example numeric fields. Since numeric data are the "easiest to deal with" (Abbott 1986), additional DSSRs for selecting other datatypes are required to ensure that EP is thorough and to allow novice testers to become competent in EP.

Hence, prescriptive DSSRs for selecting invalid equivalence classes by datatype are included in some definitions of EP, including (BS 7925-2, Kaner 1988, Tamres 2002). These definitions of the method enable greater coverage of valid and invalid input domains. On the other hand, Craig and Jaskiel (2002) argue that while special characters and decimals can be selected as invalid equivalence classes for numerical fields, they should be identified through non-prescriptive approaches like Error Guessing. Supporting this view, Andriole (1986) argues that there is "no direct, easily stated procedure" for selecting equivalence classes while Parrington and Roper (1989) claim that the only way to identify partitions is to analyse specifications for "keywords and phrases" and then identify valid and invalid classes for each one. This is particularly true when input and output fields are specified in natural language. Nonetheless, this author's view is that prescriptive definitions of DSSRs for EP would allow testers to define more complete and correct equivalence classes for any program under test (see Chapter 3 for a prescriptive approach to defining DSSRs for EP, called 'Atomic Rules').

Inadequate testing can also occur when the boundaries and contents of input and output fields are not explicitly specified, leading to a ***reliance on domain knowledge***. Consider an input field *<age> ::= [0 – 150]* in the context of a program that estimates life insurance premium cost. A developer who has experience in the domain of life insurance and may know that this field should be partitioned into two valid classes *<pensioner> ::= [65 – 150]* and *<non-pensioner> ::= [0 – 64]* and implement the program accordingly. On the other hand, if these partitions are not explicitly specified and the tester is unfamiliar with the domain of life insurance, then the program may be inadequately tested through EP (Figure 2-3)[6].

[6] This example uses domain knowledge that is so widely known that a tester should not make this exact mistake; however it illustrates the problem.

Figure 2-3: Inadequate specification of input fields, resulting in incomplete testing (Reed 1998).



Another example of ***reliance on domain knowledge*** and its effect on EP is when the 'set type' of input fields is not properly specified. Consider an input field that accepts valid Australian postcodes. The simplest definition for this field could be:

```
<postcode> ::=  <d><d><d> | <d><d><d><d>
       <d> ::=  [0 – 9]
```

However, this includes invalid postcodes, such as 0000, 1000 and 9999. An alternate definition is:

```
<postcode> ::=  [200 – 9729]
```

This still includes some invalid postcodes, such as 201 and 799. The most accurate definition would be a list every postcode that is currently recognised by Australia Post (Australia Post 2008]:

```
<postcode> ::=  [200 | 221 | 800 | 801 | 804 | 810 | 811 | ...
                | 1001 | 1002 | 1003 | 1004 | 1005 | ...
                | 2000 | 2001 | 2002 | 2004 | 2006 | ...
                | 9023 | 9464 | 9726 | 9728 | 9729]
```

Depending on how this field is specified and on the domain knowledge and experience of the programmer and tester, this field may be inadequately implemented and ineffectively tested.

Problems also arise when partitions are non-homogenous, resulting in program behaviour that is not consistent across an entire equivalence class (Jeng & Weyuker 1989), such as when a program behaves correctly when given a test data value from a partition but fails when given another value from the same partition (Hamet & Tailor 1990). In reality, this suggests that the specification did not allow the tester to identify the fact that the input field that was being partitioned that should have been divided into multiple equivalence classes, each which should have been tested separately. Also, specifications often do not

describe the 'expected output' for test data that is selected from invalid partitions (Jorgensen 1995), which can make the comparison of expected and actual testing results difficult.

There are also currently no easily automatable approaches for extracting partitions from specifications expressed in natural language. On the other hand, if program input and output fields are specified in a formal notation like BNF, and if DSSRs for EP are prescriptively defined, then this method would be amenable to automation, which would make the application of the method more repeatable and predictable (an automation approach for EP is presented in Chapter 4).

Another problem with EP, which was highlighted by the various versions of the method that were discussed throughout this section, is that ***multiple versions*** of this method have been published. This could make it difficult for both novice and experienced tester to determine how 'complete' their test sets are (e.g. how adequately each test set covers the input and output domains of each program under test, and whether all possible test case design rules from EP have been used). This also makes it ***difficult to audit*** the completeness of each black-box test set, since a number of test case design rules in each publication of EP are ambiguous and would need to be known to the auditor and interpreted in the same way by them.

2.2.2 Boundary Value Analysis (BVA)

Program faults often occur at the boundaries of data domains rather than at their centres (Pressman 1992). Boundary Value Analysis provides guidelines for selecting test data values that lie on, just above and just below the boundaries of input and output fields. Boundary values are typically selected by DISRs that select test data values from the edges of equivalence classes. The aim of BVA is to select a high yield test set that explores all program boundaries (Myers 1979). Although BVA increases the total number of test cases that are defined, it is also claimed to target the most error-prone values, increasing test effectiveness (Ould & Urwin 1986). BVA can be used to detect input and output errors as well as buffer overrun faults, which present a major security threat for online systems (Patton 2006). It is also believed that if a program can function correctly under extreme conditions then it will almost certainly operate well in ordinary scenarios (Patton 2006). BVA is considered to be more prescriptive than EP, since test data values in EP can be chosen from anywhere inside a partition, whereas in BVA the end points of the partitions are always selected (Graham 1994). It has been argued that the use of prescriptive guidelines for BVA increases the completeness of resulting test sets and the likelihood of detecting program faults (Pressman 1992). The facilitation of BVA is considered by some to be one of the greatest contributions of EP (Copeland 2004).

Myers (1979) was the first to define BVA as a purely black-box testing method and his treatment is widely cited (e.g. see (Pressman 1992, Kit 1995, Marick 1995, Andriole 1986, Tamres 2002, Parrington & Roper 1989, Mosley & Posey 2002, Copeland 2004, Rae et al. 1995, Sommerville 1994)). Myers defined six guidelines for BVA (Figure 2-4), covering the selection of boundary values from on and just outside the edges of equivalence classes. Other definitions of BVA include the selection of values that lie just inside equivalence classes (e.g. see (BS 7925-2, Copeland 2004, Graham 1994, Watkins 2001)).

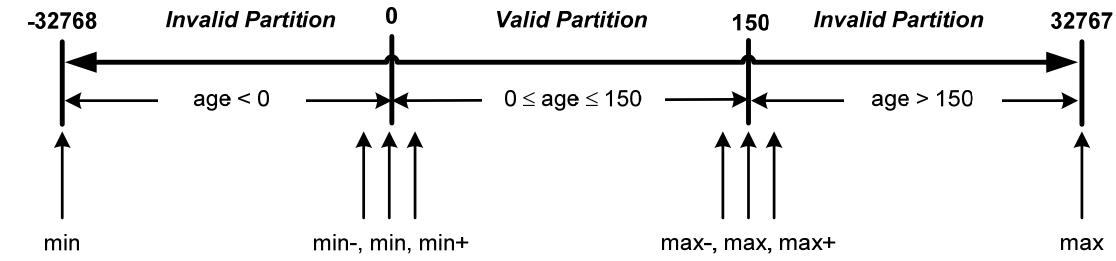
Figure 2-4: Myers' (1979) guidelines for Boundary Value Analysis.

Myers' (1979) guidelines for Boundary Value Analysis[7]

1. If an input condition specifies a range of values, write test cases for the edges of the range, and invalid-input test cases for situations just beyond the ends. For instance, if the valid domain of an input value is -1.0 - +1.0, write test cases for the situations -1.0, 1.0, -1.001 and 1.001.
2. If an input condition specifies a number of values, write test cases for the minimum and maximum number of values and one beneath and beyond these values. For instance, if an input file can contain 1 – 255 records, write test cases for 0, 1, 255 and 256 records.
3. Use guideline 1 for each output condition. For instance, if a program computes the monthly FICS deduction and if the minimum is $0.00 ad the maximum is $1165.25, write test cases that cause $0.00 and $1165.25 to be deducted. Also, see if it is possible to invent test cases that might causes a negative deduction or a deduction of more than $1165.25.
4. Use guideline 2 for each output condition. If an information retrieval system displays the most relevant abstracts based on an input request, but never more than four abstracts, write test cases such that the program displays zero, one and four abstracts, and write a test case that might cause the program to erroneously display five abstracts.
5. Of the input or output of a program is an ordered set (e.g. a sequential file, linear list, table), focus attention on the first and last elements of the set.
6. In addition, use your ingenuity to search for other boundary conditions.

For example, consider a range-based field *<age> ::= [0 – 150]*, for which one valid partition *[0 - 150]* is chosen (Figure 2-5). Six boundary values can be identified by applying six different DISRs: *-1*, *0*, *1*, *149*, *150* and *151*. These DISRs correspond to values that lie just below the lower boundary (min-), on the lower boundary (min), just above the lower boundary (min+), just below the upper boundary (max-), on the upper boundary (max) and just above the upper boundary (max)[8], where '+' and '-' refer to the smallest increment possible per datatype; for example, this could be +1 or -1 for integers and +0.01 or -0.01 for reals with two decimal places. Assuming this input was processed as an integer, for invalid classes *< 0* and *> 150*, min and max values that lie at the extreme edges of the integer range *-32768* and *32767* could be selected (BS 7925-2) (Figure 2-5).

Figure 2-5: Boundary values for a range-based field.



[7] These guidelines are expressed in Myers exact words.
[8] The terms min, min+, nom, max- and max originated from the BVA testing tool T (Jorgensen 2002).

Lists can also be tested through BVA. Consider a field that defines an ordered list of Australian capital cities: *<city> ::= [Adelaide | Brisbane | Canberra | Darwin | Hobart | Melbourne | Perth | Sydney]* (Figure 2-6). The first and last test data values 'Adelaide' and 'Sydney' could be chosen by DISRs that select on-boundary values, while the second and second-last values 'Brisbane' and 'Perth' could be selected by DISRs that test values just inside the boundaries of the list. If this field was implemented as a keyword-based input of a program with a command line interface (CLI), the 'outside' boundary values could not be tested (e.g. it does not make sense to subtract one from the keyword 'Adelaide'). On the other hand, if the field was implemented as a record list or a drop down list in a program with a Graphical User Interface (GUI), it would be sensible to use a BVA DISRs that tests 'just below' and 'just above' the field boundaries, to try to force the program to move off the end of the list array.

Figure 2-6: Boundary values for a list-based field.

As is the case with EP (and other black-box methods), ***multiple versions*** of BVA can be found in the literature. Not all definitions describe all boundary value selection rules (see Table 2-1), complicating auditing of test set completeness. While some define DISRs that select values on, inside and outside field boundaries (BS 7925-2, Copeland 2004, Graham 1994, Watkins 2001), others do not include inside (Craig & Jaskiel 2002, Kaner 1988, Lewis 2000, Mosley 1993, Myers 1979) and outside (Jorgensen 1995, Mosley 1993) boundaries. In addition, only two publications of BVA define DISRs that select boundary values from the extreme edges of integer ranges (Ould & Urwin 1986, BS 7925-2). These subtle but important differences can also make it ***difficult to audit*** the completeness of each boundary value test set, since the auditor would need to use the same definition of the method as the tester.

Craig and Jaskiel (2002) and Hutcheson (2003) argue that DISRs that select min+ (just above the lower boundary) and max- (just below the upper boundary) do not add much value as they are redundant when the 'nominal' value is selected, which is usually chosen during EP. Jorgensen (1995) maintains that the nominal value should be selected during BVA, though this violates the concept of selecting 'boundary' values; hence, min+ and max- should be considered during BVA. Jorgensen (1995) describes min+ and max- as belonging to a separate method called Robustness Testing and describes the Cartesian product of min, min-, nominal, max and max+ as Worst Case Testing. On the other hand, taking the Cartesian product of test data values is covered by combinatorial testing (see Section 2.2.7). At least one definition of BVA does not consider the selection of max+ characters to be feasible when testing GUI dialog boxes, as developers often limit the number of characters that can be input through the keyboard (Tamres 2002). As was demonstrated

in the capital city example above, this should still be tested, since field and array lengths may not be limited correctly by developers.

Thus, some versions of BVA fail to describe all boundary value selection rules. As a result, novice testers may not learn the complete set of BVA test selection rules and both novice and experienced testers may overlook certain high-yield boundary values during testing.

Table 2-1: Test selection rules for Boundary Value Analysis and their coverage in the literature.

		Referenced by																		
Data-Item Selection Rules for BVA	**Alternate Identifiers**	**(Copeland 2004)**	**(Craig & Jaskiel 2002)**	**(BS 7925-2)**	**(Graham 1994)**	**(Hutcheson 2003)**	**(Jorgensen 1995)**	**(Kaner 1988)**	**(Kaner et al. 2001)**	**(Kit 1995)**	**(Lewis 2000)**	**(Marick 1995)**	**(Mosley 1993)**	**(Myers 1979)**	**(Ould & Urwin 1986)**	**(Parrington & Roper 1989)**	**(Patton 2006)**	**(Sommerville 1994)**	**(Tamres 2002)**	**(Watkins 2001)**
Lower boundary –	min- LB -	✓	✓	✓	✓			✓	✓	✓	✓			✓	✓	✓	✓			✓
Lower boundary	Min LB	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓
Lower boundary +	min+ LB +	✓		✓	✓	✓	✓					✓					✓		✓	✓
Upper boundary –	max- UB -	✓		✓	✓	✓	✓					✓					✓		✓	✓
Upper boundary	Max UB	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓
Upper boundary +	max+ UB +	✓	✓	✓	✓				✓	✓	✓			✓	✓	✓	✓			✓

Another problem affecting BVA is that boundary conditions are often subtle (Myers 1979) and may not be obvious or explicitly specified (Patton 2006). Consequently, the effectiveness of BVA can ***rely on the domain knowledge*** of each tester (e.g. see Myers' sixth guideline in Figure 2-4). Examples of "subtle" boundaries include memory storage sizes, such as testing the edges of bytes and kilobytes (Patton 2006), minimum and maximum lengths of disc block transfers sizes (Graham 1994) and input fields that rely on ASCII table boundaries, such as testing with the 'at' symbol (@) that lies just below the lower boundary of uppercase alphabetical characters *[A – Z]* (Patton 2006, Kaner 1988). The use of this "application solution domain knowledge[9]" (Reed 1990), which could be obtained from programmers (Patton 2006), could cause BVA to be seen as a grey-box testing method, since it is based on knowledge of the program source code or the system hardware design. On the other hand, it could be argued that such information should be explicitly described in program specifications, as that would support developers in writing higher quality source code and would facilitate more thorough and effective testing. However, the type of critical thinking that is required to gain such domain knowledge could enable testers to improve the overall quality of their test suites (Jorgensen 1995). Nevertheless, it would be beneficial to have a requirements elicitation technique that testers could use to specify input and output fields to a level of detail that enables more

[9] Reed defines application solution domain knowledge as "the collection of machine executable descriptions (algorithms) which make it possible to realise the application as software" (Reed 1990).

effective black-box testing (as part of this research, Goal/Question/Answer/Specify/Verify has been developed to deal with this problem; see Chapter 3).

2.2.3 Syntax Testing (ST)

Syntax testing is a method for deriving test cases from input fields that are formally specified, often in a metalanguage like BNF (Beizer 1984). From the formal specification, an Abstract Syntax Tree (AST) is typically constructed, revealing hierarchical parent (non-terminal) and child (terminal) relationships (see Figure 2-7). Valid test cases are designed by systematically 'covering' the branches of the AST, while invalid tests can be designed by introducing faults into the terminal and non-terminal nodes of the AST.

Consider the following example, which involves a specification for a program that parses Australian street addresses (see Figure 2-7). A valid test case could be designed by selecting a valid value from each terminal node in the AST, which could result in the valid test case '*500 Main Road North Melbourne 3000*.' One approach to designing an invalid test case could be to insert an invalid alpha value in place of the <*house_number*> field, which could result in the invalid test case '*a Main Road North Melbourne 3000*.' Many of the test case design rules defined for ST overlap with rules from EP and BVA (see discussion below). Invalid test case design rules for ST can be based on likely programmer errors and program faults (Marick 1995). Test case design can be manual or automated through specification parsers and test case generators (see Section 2.7). The process of deriving test cases from formal specifications can assist in locating program and specification faults (Graham 1994).

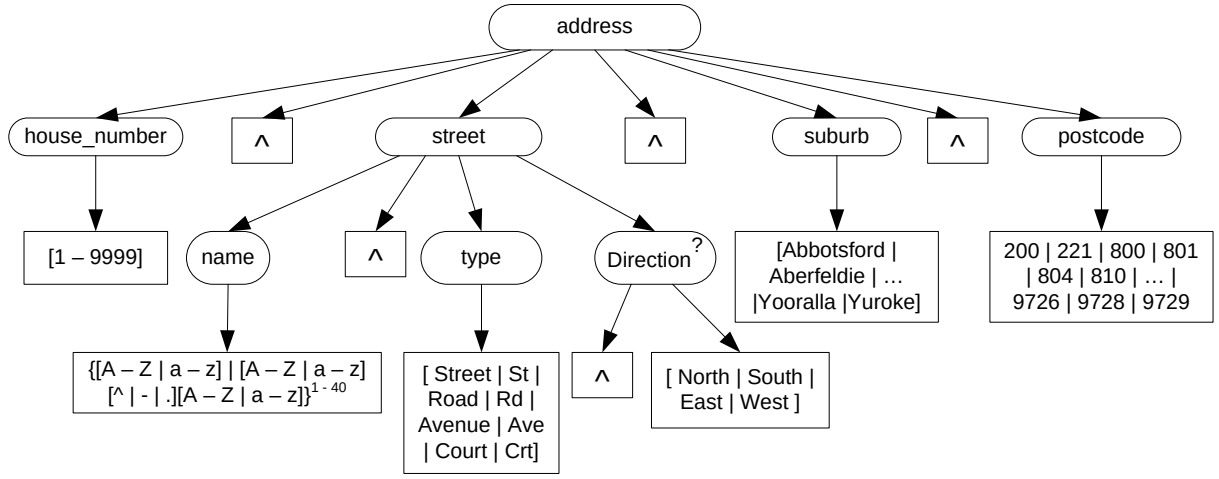
ST evolved from grammatical testing methods that were defined in the 1960's (Sauder 1962) and from compiler testing methods (e.g. see (Houssais 1977, Celentano et al. 1980, Duncan & Hutchison 1981, Bazzichi & Spadafora 1982, Homer & Schooler 1989, Marr & Lawlis 1991)). ST was used to automatically generate valid test programs for testing an Algol 68 compiler (Houssais 1977) and was researched extensively during the development and testing of FORTRAN and COBOL (DeMillo et al. 1987). ST can be used to derive invalid test programs, which provide programmers with examples of the types of input errors that their compilers may have to handle (DeMillo et al. 1987).

Figure 2-7: Simplified specification for the inputs to an Address Parser program[10].

Specification

```
      <address>  ::= <house_number> ∧ <street> ∧ <suburb> ∧ <postcode>
 <house_number>  ::= [1 – 9999]
       <street>  ::= <name>  ∧ <type>  {∧ <direction>}0-1
         <name>  ::= {[A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]}1 - 40
         <type>  ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]
    <direction>  ::= [North | South | East | West ]
       <suburb>  ::= [Abbotsford | Aberfeldie | … |Yooralla |Yuroke]
     <postcode>  ::= [200 | 221 | 800 | 801 | 804 | 810 | … | 9726 | 9728 | 9729]
              ∧  ::= [ ]  #i.e. one space
```

Abstract Syntax Tree



Optional nodes are denoted by a superscript "?"

Beizer (1995) identified various types of programs that can benefit from ST, including string recognisers, data validation code, command-driven programs, communication systems, database query languages, context-dependent menus and Macro languages that automate repetitive instructions such as the MS-DOS batch command language. Although Beizer (1990) did not recommend the use of ST for testing modern compilers, since the fact that they are automatically generated makes the types of faults detected not worth the effort required to generate the tests. Conversely, Marick (1995) argues that automatically generated parsers may still require ST to determine whether there are faults in the parser's syntax description. Kit (1995) considers ST to be less effective for testing programs that have explicit languages, such as compilers, but more effective against programs that have "hidden" languages, such as interactive commands to operating systems. Beizer (1990) argued that if program developers see enough defect reports citing program faults that were detected through ST, they may learn how to avoid making syntax-handling errors in the first place, reducing the effectiveness of the method but increasing the programmer's ability to build robust code. In this way, ST could be useful for teaching programmers how to produce higher quality programs in the first place.

[10] This specification is a simplified version of specifications given in assignments by Associate Professor Karl Reed at La Trobe University in 1998 and RMIT in 1981. The specification and corresponding program are used throughout this thesis to provide examples of various black-box testing methods and in the industry experiment discussed in Chapter 6.

Beizer (1995) claimed that the biggest payoff in ST is in the derivation of invalid test cases, though consideration should be given to the number of faults introduced per test case. Fault masking, where two faults cancel each other out to produce valid behaviour, can be avoided by introducing one fault per test case (Beizer 1995) and while double or triple defect tests may not increase test effectiveness (Beizer 1984), they may be useful for testing the diagnostic power of a system. Beizer (1995) further claimed that input errors introduced through ST can be syntactic or semantic, where syntactic faults are tested by incorrect input structure and semantic faults are tested by altering a field's input domain, such as changing the upper and lower boundaries of a numerical field, though Kit (1995) maintained that ST is not useful for semantics testing. It could be argued that altering a field's definition, such as its boundaries, leads to the selection of test data that lies outside the boundaries of the valid input domain, which is not the same as testing the semantics of a program and which is tested through BVA.

This highlights one of four problems with ST: method overlap, multiple versions, definition by exclusion and difficult to audit.

Many test case design rules from ST ***overlap*** with EP and BVA. In fact, 80% of test case design rules defined in five different and unique publications of ST overlap with other black-box testing methods (see Table 2-3, col. 3 'Is rule unique?'). For example, invalid datatypes and boundary values can be selected through ST (see Table 2-3, rules 2 to 5, 8, 10, 15 to 23, 27, 30), which overlaps with EP and BVA. This overlap could result in the design of additional and unnecessary test cases and inefficient testing. The only real differences between ST and EP/BVA are as follows.

1. Abstract Syntax Trees are not usually designed during EP or BVA, although they could be if these methods were applied to a formal specification.
2. ST is usually applied to formal specification, whereas this is not mandatory requirement for EP and BVA. However, the application of EP and BVA to formal specifications would likely improve the completeness of resulting test sets.
3. ST can produce invalid test data values through the use of Data-Item Manipulation Rules, which 'mutate' valid values (see "generic mutation" rules for ST in (BS 7925-2)), whereas typical definitions of EP and BVA (e.g. from (Myers 1979)) do not include DIMRs.

In the author's view, specifying input and output fields in a formal language like BNF also increases the precision of the specification, which can result in the production of higher-quality source code, as well as more thorough testing, regardless of the particular test methods used (see Section 2.6).

There is also ***method overlap*** within ST. For example, separate rules have been defined for repeating 'delimiter' and 'regular' fields (e.g. see Table 2-3, rule 13 for delimiters overlaps with rules 15 to 22 for regular fields). Since the repetition of a delimiter field is conceptually the same as repeating any other type of field, separate test case design rules are not required for both. Nonetheless, it can be useful to demonstrate the testing of delimiter fields to novice testers to ensure they understand how to test them.

As with EP and BVA, ***multiple versions*** of ST have been defined by various different authors (see Table 2-2, column 4 'Rule Defined By'). At present, no textbook, standard or paper describes every test case design rule for ST, which could result in the derivation of inadequate test sets and complicate the process of auditing test set completeness. This can also make it ***difficult to audit*** the completeness of each test set that is derived for ST. It would be beneficial to define one version of ST that encompasses all test case design rules from each version of the method, removes overlaps both within ST and with other black-box testing methods and makes test set comparison and auditing simpler.

Definition by exclusion exists in the test case design rules of ST. For example, the rules "introduce an invalid value for a field" and "introduce an invalid value for all fields" (see Table 2-2, rules 2 and 3) do not specify what type of invalid test data should be selected, which is similar to the ***definition by exclusion*** problem that is inherent in Myers' fourth guideline for EP (see Section 2.2.1).

Table 2-2: Test data and test case design rules for Syntax Testing.

Rule #	Error Class	Test Case Design Rule	Rule Type	Rule Defined By				
				(BS 7925-2)	(Beizer 1984)	(Beizer 1995)	(Hetzel 1988)	(Marick 1995)
1	High-level syntax errors	Introduce errors at highest level of AST (e.g. through invalid field combinations)	TCCR		✓			
2	Field value-related syntax errors	Introduce an invalid value for a field	DSSR	✓				
3		Introduce an invalid value for all fields	DSSR					✓
4		Choose invalid symbols for a field (e.g. subtraction instead of addition)	DSSR				✓	
5		Choose invalid datatypes (e.g. numbers or symbols instead of alphas)	DSSR	✓			✓	
6		Remove characters from the end of a field (e.g. “DI” instead of “DIR”)	DIMR				✓	✓
7		Add extra characters to the end of a field (e.g. “DIRR” instead of “DIR”)	DIMR			✓	✓	
8		Choose none of the legal alternatives for a field that contains alternatives	DSSR					✓
9		Choose all alternatives for one field in one test case in reverse order	TCCR					✓
10	Delimiter errors	Leave out a delimiter	DSSR		✓		✓	
11		Choose a delimiter that is valid at another syntax level but not at the current level	TCCR		✓			
12		Substitute another field for a delimiter	TCCR		✓			
13		Repeat a delimiter	DISR		✓			
14		Create errors in paired delimiters (e.g. add or remove delimiters)	DSSR		✓			
15	Repetition	One less than the minimum number of repetitions	DISR		✓			✓
16		Minimum number of repetitions	DISR		✓			✓
17		One more than the minimum number of repetitions	DISR		✓			
18		1 repetition	DISR					✓
19		One less than the maximum number of repetitions	DISR		✓			
20		Maximum number of repetitions	DISR		✓			✓
21		One more than the maximum number of repetitions	DISR		✓			
22		> 1 repetition	DISR					✓
23		Incorrect value in the last repetition of a field	DSSR					✓
24	Field-value errors (non-syntax errors)	Select invalid values for input fields	DSSR		✓			
25	Syntax-context errors (errors associated with field dependency and positioning)	Substitute a field that is correct at another level of syntax but not the current level	TCCR	✓	✓			
26		Substitute fields from same level of syntax, creating invalid order of valid fields	TCCR		✓			
27		Miss a field	DSSR	✓	✓		✓	✓
28		Add an extra field	TCCR	✓				
29		Repeat a field	TCCR	✓	✓			✓
30		Select values relating to database variable type input is stored in. e.g. if field is string 0 to 255 characters, try 0, 255 and 256	DISR		✓			
31	State-dependency errors	*No detail was provided for this rule*	N/A		✓			

Table 2-3: The overlap between Syntax Testing and other black-box testing methods.

Rule #	Syntax Testing Error Class	Syntax Testing Test Case Design Rule	Is rule unique?	Comments
1	High-level syntax errors	Introduce errors at highest level of AST; e.g. through invalid field combinations	No	Covered by combinatorial testing methods (Section 2.2.7)
2	Field value-related syntax errors	Introduce an invalid value for a field	No	Covered by Myers' 3rd and 4th EP guidelines for selecting invalid partitions (Section 2.2). Also subsumes all rules in this ST Error Class.
3		Introduce an invalid value for all fields	No	Covered by combinatorial methods (Section 2.2.7) and Myers' 3rd and 4th EP guidelines for selecting invalid partitions (Section 2.2)
4		Choose invalid symbols for a field (e.g. subtraction instead of addition sign)	No	Covered by Myers' 3rd EP guideline (Section 2.2)
5		Choose invalid datatypes (e.g. numbers or symbols instead of alphas)	No	Covered by Myers' 3rd and 4th EP guidelines for selecting invalid partitions (Section 2.2)
6		Remove characters from the end of a field (e.g. "DI" instead of "DIR")	Yes	Rule is unique
7		Add extra characters to the end of a field (e.g. "DIRR" instead of "DIR")	Yes	Rule is unique
8		Choose none of the legal alternatives for a field that contains alternatives	No	Covered by Myers's 2nd EP guideline for selecting zero alternatives (Section 2.2)
9		Choose all alternatives for one field in one test case in reverse order	Yes	Rule is unique
10	Delimiter errors	Leave out a delimiter	No	Covered by Myers's 2nd EP guideline for selecting zero alternatives (Section 2.2)
11		Choose a delimiter that is valid at another syntax level but not at the current level	No	Covered by combinatorial testing methods (Section 2.2.7)
12		Substitute another field for a delimiter	No	Covered by combinatorial testing methods (Section 2.2.7)
13		Repeat a delimiter	No	Covered by Rule # 21 in this table
14		Create errors in paired delimiters (e.g. add or remove delimiters)	No	Covered by Rule # 27 and 28 in this table
15	Repetition	One less than the minimum number of repetitions	No	Covered by BVA guidelines for selecting a value just below a lower boundary (Section 2.2.2)
16		Minimum number of repetitions	No	Covered by BVA guidelines for selecting a value on a lower boundary (Section 2.2.2)
17		One more than min number of repetitions	No	Covered by BVA guidelines for selecting a value just above a lower boundary (Section 2.2.2)
18		1 repetition	No	Covered by BVA guidelines for selecting boundary values (Section 2.2.2)
19		One less than max number of repetitions	No	Covered by BVA guidelines for selecting a value just above an upper boundary (Section 2.2.2)
20		Maximum number of repetitions	No	Covered by BVA guidelines for selecting a value on an upper boundary (Section 2.2.2)
21		One more than the maximum number of repetitions	No	Covered by BVA guidelines for selecting a value just above upper boundary (Section 2.2.2)
22		> 1 repetition	No	Covered by Myers' 1st guideline for selecting a partition above an upper boundary (Section 2.2)
23		Incorrect value in the last repetition of a field	No	Covered by combining a BVA rule to select the last boundary value (Section 2.2.2) with an EP rule that selects invalid partition (Section 2.2)
24	Field-value errors (non-syntax errors)	Select invalid values for input fields	No	Covered by Rule # 2 in this table
25	Syntax-context errors (errors associated with field dependency and positioning)	Substitute a field that is correct at another level of syntax but not the current level	No	Covered by combinatorial testing methods (Section 2.2.7)
26		Substitute fields from same level of syntax, creating invalid order of valid fields	No	Covered by combinatorial testing methods (Section 2.2.7)
27		Miss a field	No	Covered by Myers's 2nd EP guideline for selecting zero alternatives (Section 2.2)
28		Add an extra field	Yes	Rule is unique
29		Repeat a field	Yes	Rule is unique
30		Select values relating to database variable type input is stored in. e.g. if field is string 0 to 255 characters, try 0, 255 and 256	No	Covered by Myers' 2nd BVA guideline (Section 2.2)
31	State-dependency errors	*No detail or examples were provided for this rule*	No	Assumed covered by other EP, BVA and ST rules, where the expected outcome of the test case depends on system state

2.2.4 Random Testing (RT)

Random Testing is a black-box testing method in which test data values are chosen at random from the input domain of the program under test. As there is usually a pattern to the test data values that are chosen by human testers if they perform RT manually, automation is seen as a necessity (Kaner 1988). The first pseudo-random number generators were developed in the 1940's, such as those by von Neumann and Lehmer (Knuth 1973) (see (Merkel 2005) for a survey). RT is an integral part of IBM's "Cleanroom" software development methodology (Selby, Basili & Baker 1987). Craig and Jaskiel (2002) regard RT as a useful technique for "crash-proofing" a system, and though it may be effective for detecting defects that cannot be located through other black-box testing methods (Lewis 2000), it has also been argued that it should not be used in isolation from them (Watkins 2001). Four main approaches to RT are described in the literature: completely random generation (e.g. see (Kaner 1988)), random generation within equivalence classes (e.g. see (Craig & Jaskiel 2002)), random combination testing (e.g. see (Craig & Jaskiel 2002, McDermid 1991)) and statistical random testing (e.g. see (Younessi 2002)).

In *completely random input generation*, input strings are chosen entirely at random by applying DISRs that select random test data values for each program input field. For example, for the field *<name> ::= [A – Z, a – z, -]*$^{1\text{-}50}$, this approach could select a random string of alphabetical characters, numbers and non-alphanumeric characters, such as *fdhs8fd&^%^&G3F*. This approach is considered to be inefficient, since large amounts of test data are often required to cover all equivalence classes of the program (Kaner 1988).

Random generation within equivalence classes offers an improvement to RT, by first applying DSSRs from EP to partition the input domain, and then by applying DISRs that randomly select inputs from the (valid and invalid) partitions, facilitating adequate input domain coverage. In this approach, each partition should be covered by at least one test case (Craig & Jaskiel 2002). For example, for the *<name>* field, this approach could be used to randomly select strings of characters from the valid partition *[A – Z, a – z, -]*$^{1\text{-}50}$ and invalid strings that are too long (> 50 characters), too short (0 characters) or that contain invalid characters (e.g. integer, real, non-alphanumeric). Input domain coverage can be further enhanced by generating a different value for each partition every time testing is carried out.

In *random combinatorial testing* (also called "semi-random testing" (Craig & Jaskiel 2002)), TCCRs are applied to randomly select ordered pairs of values from valid input partitions to produce test cases (McDermid 1991). The disadvantage is that pairs of values may be reselected as test case design progresses.

In *statistical random testing*, test data generation is based on a probability distribution of the input domain (Younessi 2002). Common distributions are normal, negative exponential, Erlang, Poisson, Weibull, Student T (vonMayrhauser 1990) and uniform (Duran & Ntafos 1984). The distribution can also be based on the expected runtime distribution of the system, which is known as the "operational profile" (Thayer, Lipow & Nelson 1978, Musa 1993), which usually takes into account the frequencies at which particular inputs occur as well as likely sequences or combinations of inputs. This facilitates prediction of future reliability based on the reliability of the system when it is in use (Bertolino 2004). While this approach has been recommended for System Testing (Burnstein 2003) and for testing just prior to release

(Thayer, Lipow & Nelson 1978) (i.e. Acceptance Testing), its effectiveness relies on the accuracy of the chosen distribution, which may be unknown prior to release (Younessi 2002) and may change as the system matures (Weyuker & Jeng 1991).

There has been much debate on the effectiveness of RT. It was criticised by Myers (1979) as being "the poorest methodology" of all black-box methods, for having the least chance of any method to detect errors and to select an optimal test set. Craig and Jaskiel (2002) argue that randomly generated test cases may not be realistic and also claim there is no way of measuring their coverage or risk, although random generation within equivalence classes presents one approach for assessing coverage. A number of studies have compared Random Testing to grey-box Partition Testing (e.g. selecting test data values from the input domain that achieve various levels of source code branch or path coverage). Duran and Ntafos (1981, 1984) compared Random and Partition Testing by dividing the input domain of a simulated faulty program into twenty-five partitions that had randomly assigned failure rates. They found that Partition Testing had a higher probability of detecting at least one failure when the same numbers of tests were selected for both methods, but that it was more effective if it was used to select twice as many tests as Partition Testing.

Jeng and Weyuker analysed the theoretical conditions under which RT and Partition Testing would detect at least one failure (Jeng & Weyuker 1989, Weyuker & Jeng 1991). They argue that Partition Testing is "most successful" when partition selection is fault-based (e.g. testing for boundary errors) rather than control-flow or data-flow based. Chen and Yu (1994) extended Weyuker and Jeng's work, finding that Partition Testing can be as effective as RT, providing the number of test cases selected is proportional to the size of each partition (i.e. proportional partition testing). Ntafos (1988) developed a simulator to compare RT to proportional Partition Testing, finding that proportional Partition Testing is theoretically more effective when fewer test data values per partition are selected. This was supported by Chen and Yu (1996), who analytically compared RT and Partition Testing from the perspective of the expected number of failures detected, finding that Partition Testing was as effective as RT when partitions with failure rates that are greater than that of the entire input domain have higher sampling rates. Hamlet and Taylor (1988, 1990) reported a similar result from a theoretical comparison of these two techniques, finding that Partition Testing is only effective when partitions with a high failure rate are identified. The challenge with these latter approaches is that typically the failure intensity of each partition is not known in advance of testing.

2.2.5 Non-Prescriptive Approaches to Black-Box Testing

Non-prescriptive test case design is an unstructured and typically unscripted approach to testing that is often based on the unique domain knowledge and experience of each tester. In the most extreme case, it can be carried out ad hoc (also called 'ad-lib testing' (Beizer 1984)), without prior knowledge of the program under test and without any test case design prior to test execution. Non-prescriptive testing supplements the inherent incompleteness of prescriptive methods like EP and BVA (Mosley 1993) because it can be used to identify test cases that are not selectable through the use of prescriptive black-box testing methods.

Although ad hoc testing has been criticised by Copeland (2004) as being "sloppy, careless, unfocussed, random and unskilled", others regards the fact that it does not require special training, knowledge or

experience as a benefit (Patton 2006). Differences in such opinions may be caused by the varying degrees of ad hoc testing, from completely unstructured to semi-formal.

As Watkins (2001) argues, "some people have a natural flair for finding defects in software systems." During ad hoc testing, "pathological testers" (Reed 2007) may be using their own individual undocumented and yet systematic approaches to test case design (Craig & Jaskiel 2002). Although the most basic form of ad hoc testing is to "behave like a dumb user" (Patton 2006), approaches like Error Guessing (EG) and Exploratory Testing (ET) (discussed below) provide guidance to non-prescriptive testing. These approaches can be effective because they are based on the domain knowledge and expertise of experienced testers. Even when prescriptive testing is rigorous and ad hoc testing does not detect any major faults, this knowledge can improve confidence in a system (Beizer 1984). Although ad hoc testing can be enjoyable (Craig & Jaskiel 2002) it can also be frustrating for experienced testers to see novice (or "pathological") testers easily and quickly "crash" programs with this approach (Patton 2006).

A problem with all non-prescriptive testing approaches is that test cases are often not documented and, as a result, they cannot be reused during regression testing (Myers 1978) or for reproducing failures. A simple improvement (that is based on good testing practice) is to record the steps taken during testing and the expected and actual results of each test, as this facilitates analysis of whether the test section rules used during testing should become part of the prescriptive testing methods routinely used against the system under test (Craig & Jaskiel 2002). This allows testing to be tailored to the specific domain of each system under test. For example, EG rules identified through analysis of the "application solution domain" (Reed 1990) of mathematical software include testing for divide by zero errors and taking the square root of negative numbers (Mosley 1993), while rules from the "technology domain" (Reed 1990) of testing systems with relational databases include testing with escape characters that form part of the database query language. If effective test case design rules for each application and technology domain are identified and recorded, this facilitates more effective testing regardless of each tester's individual domain knowledge or experience (this is one of the aims of a new customisation approach called Systematic Method Tailoring; see Chapter 4).

Two popular non-prescriptive approaches to black-box testing, Error Guessing and Exploratory Testing, are discussed below.

2.2.5.1 Error Guessing (EG)

The first non-prescriptive black-box testing approach was Error Guessing, which was published by Myers in 1979 (Myers 1979). In this approach, testers identify a list of "error-prone situations" and derive test cases that are capable of detecting each potential fault (Myers 1979). Jorgensen argues that EG "is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform… There are no guidelines, other than to use 'best engineering judgement'. As a result, special value testing is very dependent on the abilities of the tester… Even though special value testing is highly subjective, it often results in a set of test cases which is more effective in revealing faults than the test sets

generated by the other methods… testimony to the craft of software testing"[11] (Jorgensen 1995). Error Guessing has also been referred to as "special values testing" (Jorgensen 1995, Perry 2000), "free-form testing" (Lewis 2000), "inspirational testing" (Black 2007) and (perhaps less commonly) "seat of the pants/skirt" testing (Jorgensen 1995).

EG has a ***reliance on domain knowledge*** that may originate from a tester's understanding of the design or implementation of the system under test or similar systems (Bertolino 2004, Watkins 2001), testing methods (Craig & Jaskiel 2002) and heuristics (Watkins 2001), the types of tests that previously detected faults (Watkins 2001), hardware (Mosley 1993), platforms (Bertolino 2004) and programmer assumptions (Myers 1979).

Craig and Jaskiel (2002) argue that the thought processes involved in EG may be similar to the procedures carried out during prescriptive black-box testing. In fact, many published descriptions of this ***method overlap*** significantly with prescriptive black-box testing methods (see Table 2-4). For example, Myers (1979), Andriole (1986) and Graham (1994) all include the selection of boundary values in their definitions of EG, which overlaps with BVA. Graham (1994) recommends selecting invalid datatypes through EG, which overlaps with EP. Interestingly, of the thirty different test case design rules that are defined for EG in five different textbooks on software testing ((Graham 1994, Jorgensen 1995, Mosley 1993, Mosley 1993, Myers 1979)), only six (20%) are unique (see Table 2-4). Thus, 80% of these rules overlap with other prescriptive black-box testing methods. ***Multiple versions*** of EG exist, probably because it is based on the domain knowledge of each author.

One unique aspect of EG is that it can be applied to test case design rules from other prescriptive black-box testing methods to facilitate selection of the most effective rules for testing (Watkins 2001); e.g. by identifying rules that previously detected faults (Kit 1995) or that focus on testing critically important aspects of the system (Craig & Jaskiel 2002) (i.e. risk-based testing). Black-box testing methods can be chosen for their ability to detect particular types of faults (Jorgensen 1995), such as by applying BVA to systems that suffer from boundary-related errors, which is similar to EG.

Abbott (1986) claims that the effectiveness of EG cannot be guaranteed since it is "ill-defined" with no universal approach. Although Andriole (1986) states that EG "carries no guarantee for success, but neither does it carry any penalty", in the author's view, it can result in wasted time and effort if different testers who are testing the same program overlap in their derived test sets. Nonetheless, EG is believed by some to be more efficient and effective than prescriptive black-box testing methods (Watkins 2001, Jorgensen 1995). Mosley and Posey (2002) argue that EG compensates for the "inherent incompleteness" of EP and BVA. Empirical data supporting this includes a study of failures detected in independently developed launch-intercept control software, where it was established that 83-90% of faults and 90-97% of failures were detected by special values (Wild, Chen & Eckhardt 1989). Nonetheless, EG is believed by some to be one of the most commonly used black-box testing methods in industry (Jorgensen 1995). For example, in a survey of software testing practices in Australia, which revealed that out of 65 organisations interviewed,

[11] Jorgensen (1995) did not quote any sources to support his view that Error Guessing is more effective than other testing methods.

just over one third (35.4%) choose to use ad hoc testing approaches over prescriptive black-box testing methods (Ng et al. 2004).

Given the apparent popularity and effectiveness of EG as an approach to defect detection, if (as mentioned in Section 2.2.5) the black-box test case design rules that are proven to be effective against particular 'error prone situations' are recorded, then this information can be shared with and taught to novice and experienced testers to facilitate more effective testing, regardless of each tester's existing domain knowledge and experience. Test Catalogues, Categories and Matrices (see Section 2.2.6) could be used to achieve this, as could Systematic Method Tailoring.

Table 2-4: Overlap between Error Guessing and prescriptive black-box testing methods.

Context	Error Guessing Rule	Rule Type	Is rule unique?	If rule is not unique, rule overlaps with…	Rule Defined By
Testing a sorting routine	Input list is empty	DISR	No	BVA min selection rule (Table 2-1)	(Myers 1979)
	Input list contains one entry	DISR	No	BVA min+ selection rule (Table 2-1)	(Myers 1979)
	All entries in list have same value	TCCR	Yes	-	(Myers 1979)
	Input list is already sorted	TCCR	Yes	-	(Myers 1979)
Function that grades multiple choice examination answers	Does program accept blank?	DISR	No	BVA min selection rule (Table 2-1)	(Myers 1979)
	Substitute student answer records for student information records	TCCR	No	Specification-Based Mutation Testing rule (Section 2.2.7.5)	(Myers 1979)
	Correct or student answer record are missing identification flags	DISR	No	BVA min- rule (i.e. selects a null length string) (Table 2-1)	(Myers 1979)
	Two students have same name or number	TCCR	No	ST duplication rules (Table 2-3)	(Myers 1979)
	Calculate median grade with odd and an even number of students	TCCR	Yes	-	(Myers 1979, Mosley 1993)
	Number of questions in the exam contains a negative number	DISR	No	EP less than lower boundary selection rule (Figure 2-1)	(Myers 1979)
Generic	Divide by zero	DISR	No	BVA min- rule (i.e. selects a null length string) (Table 2-1)	(Graham 1994, Mosley 1993)
	Empty file	DISR	No	BVA min selection rule (Table 2-1)	(Graham 1994)
	Empty record	DISR	No	BVA min selection rule (Table 2-1)	(Graham 1994)
	Empty field	DISR	No	BVA min selection rule (Table 2-1)	(Graham 1994)
	Negative number	DSSR	No	EP less than lower boundary selection rule (Figure 2-1)	(Graham 1994)
	Alpha character for numeric field	DSSR	No	EP “must be” rules (Figure 2-1)	(Graham 1994)
	Decimal point	DSSR	No	EP “must be” rule (Figure 2-1) or ST invalid datatype rule (see Table 2-3)	(Graham 1994)
	Embedded comma	DISR	No	Delimiter-based ST rule (Table 2-3)	(Graham 1994)
	Minimum size	DISR	No	BVA min selection rule (Table 2-1)	(Graham 1994)
	Square root of negative number	DSSR	No	Could be chosen by EP less than lower boundary rule (Figure 2-1)	(Mosley 1993)
	Maximum size	DISR	No	BVA max selection rule (Table 2-1)	(Graham 1994)
Function to calculate the next date	February 28	DISR	No	BVA max selection rule (Table 2-1)	(Jorgensen 1995)
	February 29	DISR	No	BVA max+ selection rule (Table 2-1)	(Jorgensen 1995)
	Leap years	DISR	Yes	-	(Jorgensen 1995)
Table length	Processing variable length tables	DSSR or DISR	No	Could be chosen through an EP or BVA rule that tests field count (Figure 2-1 & Table 2-1)	(Mosley 1993)
Cyclic mater file/database updates	Improper handling of duplicate keys	TCCR	No	ST field duplication rule (Table 2-3)	(Mosley 1993)
	Unmatched keys	DSSR	No	EP invalid value selection rule (Figure 2-1)	(Mosley 1993)
	Overlapping storage areas	TCCR	Yes	-	(Mosley 1993)
	Overwriting of buffers	DISR or DSSR	No	BVA max+ selection (see Table 2-1) or EP greater than upper boundary selection (Figure 2-1) for buffer length	(Mosley 1993)
	Forgetting to initialise buffer areas	TCCR	Yes	-	(Mosley 1993)

2.2.5.2 Exploratory Testing (ET)

The term Exploratory Testing was first used by Kaner in 1988 (Kaner 1988) to describe the process that experienced software testers follow when they design and execute non-prescriptive test cases against a program on the fly, while simultaneously learning about it (Craig & Jaskiel 2002). Rather than being a black-box testing *method* per se, it is a non-prescriptive testing approach that can constitute black or grey-box testing. Unlike EG, where lists of error prone situations are identified in advance of test execution, in ET each new test case is designed on the fly, based on knowledge gained during execution of previous test cases. ET was invented by professional testers and some of the largest organisations in the world use it. For example, Microsoft uses ET during testing of their Windows operating system (Microsoft 2003), including for the purposes of compatibility testing new versions of Windows (Page et al. 2009).

Jonathan and James Bach (2006) liken ET to a game of twenty questions, where one person thinks of an object (i.e. an animal, vegetable or mineral) and a guesser asks up to twenty questions with yes/no answers to try to deduce what the object is. They point out that this game would not work using a scripted testing approach, where all twenty questions were designed in advance and could not be adapted from the results of each answer. Therefore, ET is an iterative approach in which the identification of each test case depends on the results of previous tests (Dustin 2003) where each new test case has potential of being more effective than the last (Kaner et al. 2001). Agruss and Johnson (2000) argue that the numbers and severity of faults that can be detected through this approach "can be astounding."

ET encourages creativity (Itkonen & Rautiainen 2005) as it does not "disrupt the intellectual processes that make testers able to find important problems quickly" (Bach 2001). While there are no formal procedures defined for ET (Itkonen & Rautiainen 2005), the general approach is to choose an area of a program and design and execute a test case against it (often without recording them) and then use the actual results of testing to decide what to test next. ET can be performed "freestyle" without any guidelines or "session-based" where during timed, uninterrupted sessions (Copeland 2004). Sessions typically run from 45 minutes to 2 hours and are supported by "charters" (documents used to record testing goals), "session sheets" (documents that are used to record what occurred during testing) and "session debriefings" (meetings in which testers discuss the results of testing) (Bach 2000). Session-based ET ensures testers have enough time to perform ET effectively and allows them to remain focussed throughout testing (Copeland 2004). It facilitates planned, managed and controlled ET (Itkonen & Rautiainen 2005).

Similar to EG, ET can be used to select test cases that cannot be identified through scripted testing (Bach 2003). As Copeland (2004) claims, one of the skills experienced exploratory testers require is the ability to "choose appropriate test design techniques", suggesting that they are also more skilled at strategising the best approaches to testing systems in a general sense[12]. The knowledge and experience utilised during ET can originate from a tester's understanding of effective test case design rules from prescriptive testing methods (Craig & Jaskiel 2002). Thus, ET has a ***reliance on domain knowledge*** (Craig

[12] This raises the question of how test strategy development can be taught to inexperienced testers. One approach is to strategise and prioritise the choice of test methods based on the identification of testing and technology-related risks (K. J. Ross & Associates 2007).

& Jaskiel 2002) and the test case design rules utilised during ET can ***overlap*** with rules from prescriptive testing methods.

As Kaner et al. recognised (2001), sharing the testing heuristics utilised during ET with other testers can "improve the quality of their guesses." On the other hand, exploratory test cases are often not documented (Itkonen & Rautiainen 2005). Accordingly, ET could benefit from recording the actual test case design rules that are used during testing (Dustin 2003) as this would allow them to be reused and shared with other testers. Test Catalogues, Categories and Matrices or Systematic Method Tailoring could support the documentation of the test case design rules that are used during ET. As Everett and McLeod attest (2007), "Although some interesting results have been obtained by experienced testers using the "exploratory testing" approach, its premise… appears to contradict prudent testing practices for the inexperienced tester." Other challenges with ET, which were specifically reported by Microsoft, are that ET "generally doesn't scale well" for testing large-scale or mission-critical systems and that it is "not the best approach" for testing on long-term maintenance project (Page et al. 2009).

Nonetheless, many practitioners agree that ET can enable testing to be focussed on the most important or error prone areas of a system (Craig & Jaskiel 2002, Copeland 2004, Page et al. 2009), which is an example of risk-based testing. The operational profile (Thayer, Lipow & Nelson 1978, Musa 1993) of a system can be used in a similar way, to ensure ET is carried out on the areas of a system that will undergo the most usage (the operational profile could theoretically be used to support any prescriptive black-box testing method (e.g. EP, BVA, ST) to prioritise testing to the most important aspects of the system first). ET can be effective for uncovering additional information about previously detected defects (Copeland 2004) and can be used to provide rapid feedback to developers on the success of system changes (Itkonen & Rautiainen 2005). ET can be effective in prototyping environments, where it can be utilised by end-users to evaluate systems early in the SDLC (Rubin 1994). This is because it can be applied to systems that have not been properly specified, as the processes followed during ET allows testers to learn about the system under test and what constitutes valid and invalid input (Tamres 2002). It can also be useful prior to prescriptive testing (Patton 2006) as a means for locating error-prone areas of a system that require further exploration.

ET can be used to test a program without requirements; however, assumptions testers make may be different from developer's assumptions and neither may satisfy end-user requirements. It can be challenging to test complex aspects of a system effectively without requirements and this can result in untested requirements (Dustin 2003) and ineffective testing, particularly if testers do not have enough domain knowledge in the system (Patton 2006, Itkonen & Rautiainen 2005). Copeland (2004) argued that user manuals should be used during ET, as they allow testing to be carried out from the end-user's perspective, while Itkonen & Rautiainen (2005) argued that user manuals and even marketing material can improve ET effectiveness. It could equally be argued that user manuals can improve prescriptive black-box testing.

Dustin (2003) believes that "all test efforts" require ET, regardless of whether requirements are documented or not, but also notes that when specifications are ambiguous or incomplete, domain knowledge from developers and customers may need to be utilised in order to determine how to test the system effectively. It would be advantageous to define a prescriptive approach for recording the domain

knowledge utilised during ET, as this would enable it to be reused and for it to be shared with other novice and experienced testers (e.g. see Chapter 3 for a definition of a new approach called Goal/Question/Answer/Specify/Verify).

One final reported disadvantage of ET is that it cannot prevent defects (Copeland 2004). Since testing can only prove the presence of defects, never their absence (Dijkstra 1969), this is a moot point.

2.2.6 Test Catalogues, Test Categories and Test Matrices

Test Catalogues, Test Categories and Test Matrices are three similar approaches that can be used to map black-box test case design rules to input field types, in order to plan and trace black-box testing. They offer a simple solution to documenting the contexts in which particular test case design rules are applicable and effective, allowing this information to be stored and reused against the same or similar systems and for it to be shared with other testers. They also enable testing to be tailored to the specific testing needs of each program under test. The test case design rules included in the catalogues, categories and matrices can be selected from prescriptive black-box testing methods like EP, BVA and ST, from the domain knowledge of experienced testers and from grey-box information (e.g. knowledge of program source code).

Test Catalogues consist of lists of test case design rules that can be applied to specific types of input field (Table 2-5) (Marick 1995). Marick (1995) defined Catalogues for testing Unix-based programs and for various datatypes (e.g. numerical data) and program data structures like trees and lists.

Test Categories are comprised of questions that guide testing for specific types of input fields and for examples of expected results for testing with invalid inputs (Figure 2-8) (Tamres 2002). Questions included in Test Categories can be identified ad hoc by experienced testers or can be chosen from prescriptive black-box testing methods. For example, the Category “do it twice” (Figure 2-8) is similar to test case design rules for testing repetition in ST, the “expected system behaviour” for this category is similar to Myers’ BVA rules (1979) that attempt to force output fields to take on particular boundary values and questions for valid and invalid categories are similar to partitioning rules from EP. Tamres (2002) identified Test Categories for various black-box (Figure 2-8), white-box and non-functional scenarios.

The unique aspect of this approach that sets it apart from prescriptive black-box methods is that the questions within each Category prompt the tester to consider the classes of test data that can be chosen for various types of input fields, rather than defining the exact points of the input domain that must be tested. This encourages creativity and can provide a mechanism for recording test cases during Exploratory Testing (see Section 2.2.5.2). Test Categories are reminiscent of Error Guessing (see Section 2.2.5.1), in which a tester questions the types of faults may be inherent in the system under test and then uses that information to determine how to test the system. The questions within Test Categories could also be used to question the completeness of input data specifications, as they could be used to prompt deliberation on the types of inputs that should be accepted or rejected by the program and the expected behaviour for each one.

Table 2-5: Example of a Test Catalogue for testing a numerical field (Kaner et al. 2001) and the black-box testing methods that define the rules in the catalogue.

Test Case Design Rule (Kaner et al. 2001)	Corresponding Black-Box Testing Method & Rule	Rule Type
Nothing	BVA: lower boundary –	DISR
Empty field (clear the default value)	BVA: lower boundary –	DISR
Outside of upper bound (UB) # of digits or characters	BVA: upper boundary +	DISR
0	Error Guessing: select 0	DISR
Valid value	EP: select value from valid partition	DISR
At lower bound (LB) of value – 1	BVA: lower boundary –	DISR
At lower bound (LB) of value	BVA: lower boundary	DISR
At upper bound (UB) of value	BVA: upper boundary	DISR
At upper bound (UB) of value + 1	BVA: upper boundary +	DISR
Far below the LB of value	EP/BVA: select lower boundary of partition that lies below lower boundary of a valid field	DSSR/DISR
Far above the UB of value	EP/BVA: select upper boundary of partition that lies above the upper boundary of a valid field	DSSR/DISR
At LB number of digits or characters	BVA: lower boundary	DISR
At LB – 1 number of digits or characters	BVA: lower boundary –	DISR
At UB number of digits or characters	BVA: upper boundary	DISR
At UB + 1 number of digits or characters	BVA: upper boundary +	DISR
Far more than UB number of digits or characters	EP/BVA: select upper boundary of partition of digits or characters that are longer than the max field length	DSSR/DISR
Negative	Error Guessing: select a negative number or EP: select value from partition that lies below lower boundary of the valid field	DSSR
Non-digits, especially / (ASCII 47) and : (ASCII 58)	Error Guessing: select special values	DSSR
Wrong datatype (e.g. decimal into integer)	EP: select invalid datatype	DSSR
Expressions	Error Guessing: select special values	DSSR
Leading spaces	ST: add extra characters to start of a field	DIMR
Many leading spaces	ST: add extra characters to start of a field	DIMR
Leading zero	ST: add extra characters to start of a field	DIMR
Many leading zeros	ST: add extra characters to start of a field	DIMR
Leading + sign	ST: add extra characters to start of a field	DIMR
Many leading + signs	ST: add extra characters to start of a field	DIMR
Nonprinting characters (e.g. Ctrl+char)	Error Guessing: select special values	DSSR
Operating system filename reserved chars (e.g. "*.:")	Error Guessing: select special values	DSSR
Language reserved characters	Error Guessing: select special values	DSSR
Upper ASCII characters (128-254)	Error Guessing: select special values	DSSR
ASCII 255 (often interpreted as end of file)	Error Guessing: select special values	DSSR
Uppercase characters	EP: select invalid datatype (alphabetic characters)	DSSR
Lowercase characters	EP: select invalid datatype (alphabetic characters)	DSSR
Modifiers (e.g. Ctrl, Alt, Shift-Ctrl)	Error Guessing: select special values	DSSR
Function keys (e.g. F2, F3, F4)	Error Guessing: select special values	DSSR
Enter nothing but wait for a long time before pressing the Enter or Tab key, clicking OK, or doing something equivalent that takes you out of a field	Error Guessing: select special values	DSSR/DISR/ TCCR
Enter one digit but wait for a long time before entering another digit or digits and then press the Enter key	Error Guessing: select special values	DSSR/DISR/ TCCR
Enter digits and edit them using the backspace and arrow keys	Error Guessing: select special values	DSSR/DISR/ TCCR
Enter digits while the system is reacting to interrupts of different kinds (e.g. printer activity, clock events, mouse movement)	Error Guessing: select special values	DSSR/DISR/ TCCR
Enter digit, shift focus to another application, return to the application to see where focus is	Error Guessing: select special values	DSSR/DISR/ TCCR

Figure 2-8: Test Categories for black-box testing (Tamres 2002).

Test Category: no data provided

Possible questions:

- How can the system be starved?
- What happens if the system is not provided with any data?
- What does it mean to withhold data?
- What are the default values or states?

Expected system behaviour can include:

- post an error message;
- provide a default value;
- reuse the prior value or state;
- prompt the user for missing data;
- void the transaction;
- abort execution and enter a message in the log file.

Test Category: do it twice

Possible questions:

- What happens if you provide the same data or input twice in succession?

Expected system behaviour can include:

- post an error message;
- overwrite the previous value or state;
- prompt user to approve overwriting prior value;
- ignore the second incident;
- process the second request as a separate independent path.

Test Category: valid data

Possible questions:

- What are valid instances of this data?
- What is the valid input range?
- What are the boundary data?
- What is the format of a valid packet?
- What information is provided in a valid transaction?

Test Category: invalid data

Possible questions:

- What does it mean to exceed the bounds?
- What are the consequences?
- What constitutes bad data for the application under test?

Examples of invalid data for numerical fields:

- values out of range;
- negative numbers;
- decimals;
- leading zeros or spaces;
- alphabetic characters.

Examples of invalid data for alphanumeric fields:

- leading spaces;
- non-alphanumeric characters;
- special keystrokes, such as CONTROL-SHIFT combinations.

Examples of invalid data for signal driven input include:

- bad timing specifications;
- timeout;
- bad signal;
- missing acknowledge response;
- bad checksum;
- noise.

Possible system behaviour for invalid conditions include:

- post an error message;
- prompt user for correct data;
- reuse a prior valid value or state;
- void the transaction;
- abort execution and enter a message in the log file;
- ignore incident and try to process request as given.

Test Matrices can be used to map black-box test case design rules to types of program input/output fields (e.g. integers, rational numbers, filenames, dates) or to program actions (e.g. create, read, update, delete, replace, append or overwrite files) (Table 2-6) (Kaner et al. 2001). The top row of the matrix consists of test case design rules, the left column comprises input and output fields and the cells of the matrix can be used to track which rules can be applied to each field and whether the program has passed or failed each test. Although test case design rules can be selected from prescriptive black-box testing methods like EP, BVA and ST, brainstorming has also been suggested as an effective approach for rule identification (Kaner et al. 2001) (which is essentially an ad hoc approach to testing).

Table 2-6: Example of a Test Matrix (adapted from (Kaner et al. 2001)).

	Test Case Design Rules																									
Field Type	**Nothing**	**Empty (clear default)**	**0**	**LB –**	**LB**	**UB**	**UB +**	**Far below LB**	**Far above UB**	**UB number of chars**	**UB + 1 chars**	**Far beyond UB chars**	**Negative**	**Non-digit (/ ASCII 47)**	**Non-digit (: ASCII 58)**	**Wrong data type**	**Expressions**	**Leading Spaces**	**Non-printing char**	**O/S file name**	**Upper ASCII**	**Upper case**	**Lower case**	**Modifiers**	**Function keys**
Numeric																									
Alpha																									
Alphanumeric																									

One challenge is that standard guidelines for *creating* new Catalogues, Categories or Matrices have not been published. These approaches ***rely on the domain knowledge*** of each individual tester and as such, they could be enhanced through the definition of systematic approaches for identifying the types of input and output fields that can be mapped to particular types of input and output fields. This could be facilitated by Systematic Method Tailoring. A further enhancement could be to define test case design rules in a more prescriptive format, as this would allow the rules described in Catalogues, Categories and Matrices to be interpreted in the same way by different testers. This could be supported by the Atomic Rules approach.

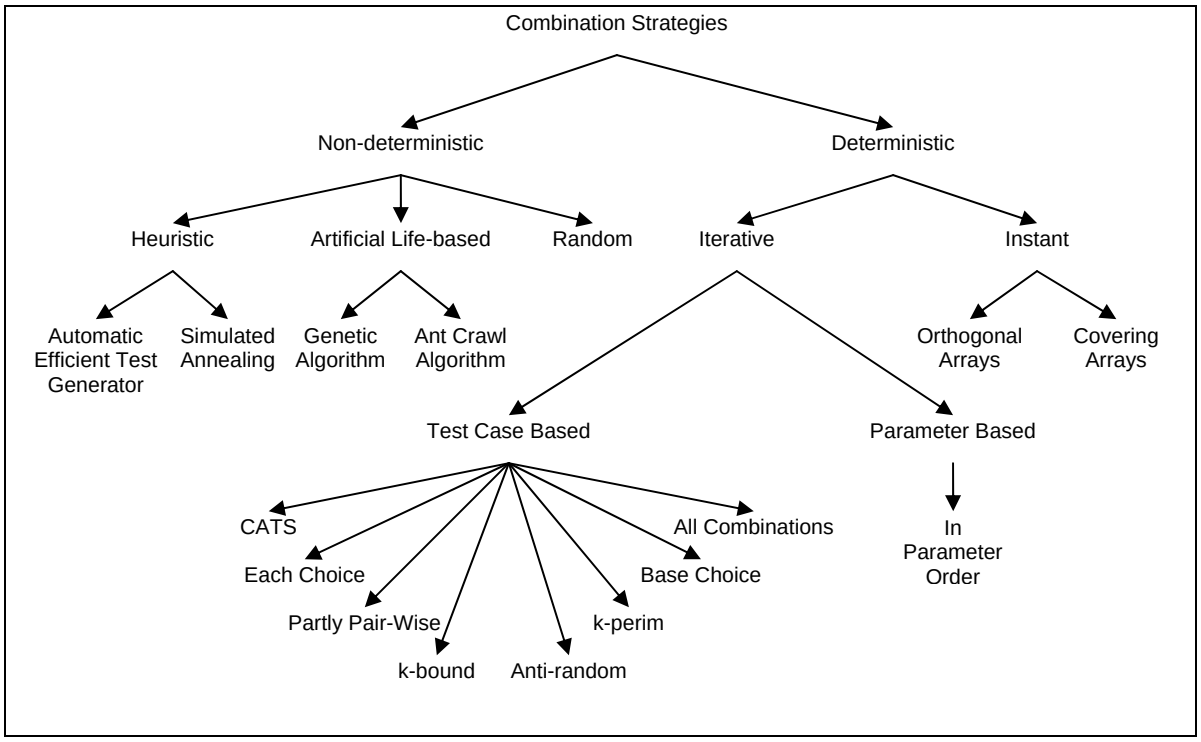
2.2.7 Combinatorial Test Methods

In black-box testing methods like EP and BVA, test cases are usually derived by manually choosing and then combining valid and invalid test data values (e.g. see the "one-to-one" and "many-to-one" test case construction rules defined in (BS 7925-2)). Combinatorial test methods enable the automatic generation of black-box test cases via the application of combinatorial algorithms (i.e. TCCRs) to test data values that are derived during the application of methods like EP, BVA and ST. Combinatorial testing methods such as pair-wise testing (see Section 2.2.7.4) can be used to reduce the number of test cases that are generated by other black-box testing methods (Watkins 2001). Many of the concepts underlying combinatorial testing originate from mathematics (e.g. see Section 2.2.7.1.1).

Grindal, Offutt & Andler (2005) distinguish between deterministic and non-deterministic combinatorial strategies (Figure 2-9). *Deterministic strategies* always produce the same result given any set of test data values. Deterministic strategies can be further divided into *instant strategies* that produce a complete test set at once and *iterative strategies* that derive test sets step by step. *Non-deterministic strategies* utilise randomisation at some point in their algorithms.

Although combinatorial test methods are not the main focus of this thesis, they can be supported by the new approaches presented in Chapter 3. Thus, in the following sections, this family of testing approaches are explained and explored.

Figure 2-9: Classification scheme for combinatorial test methods (Grindal et al. 2005).



2.2.7.1 All Combinations (AC)

One of the most popular algorithms in testing is All Combinations, where test cases are chosen by taking the n-ary Cartesian product of a test data set, resulting in a set of ordered tuples. This is also known as the direct product of sets and the cross product. It results in a test set that achieves *n-wise* coverage, where all possible combinations of data values from *n* fields are covered by at least one test case (Grindal et al. 2005).

The Cartesian product of two sets ***A*** and ***B*** is called the binary Cartesian product, denoted $\boldsymbol{A \times B}$, which is the set of all ordered pairs of data values from ***A*** and ***B***:

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}.$$

Likewise, the Cartesian product of ***n*** sets, denoted $\boldsymbol{A_1 \times \ldots \times A_n}$, is the set of all ordered tuples:

$$A_1 \times \ldots \times A_n = \{(a_1, \ldots, a_n) \mid a_1 \in A_1 \wedge \ldots \wedge a_n \in A_n\}.$$

For ***n*** fields, where field $\boldsymbol{P_i}$ has $\boldsymbol{V_i}$ test data values, the number of test cases selected is (Grindal, Offutt & Andler 2004):

$$\prod_{i=1}^{n} V_i$$

Consider the fields 'name' and 'city' defined as <*name*> = *[Adrian* | *Joanna* | *John* | *Mary* | *Nicole* | *Steve]* and <*city*> = *[Adelaide* | *Brisbane* | *Canberra* | *Darwin* | *Hobart* | *Melbourne* | *Perth* | *Sydney]*, for which test data values <*name_values*> = *{John, Mary, Steve}* and <*city_values*> = *{Melbourne, Sydney}* are chosen. All Combinations can be used to selects 3 x 2 = 6 test cases: (John, Melbourne), (John, Sydney),

(Mary, Melbourne), (Mary, Sydney), (Steve, Melbourne), (Steve, Sydney). The order of elements is retained, ensuring that each field is only assigned values that have been chosen from its own partition. In any real-world example, this approach is likely to cause a combinatorial explosion that can make it prohibitively expensive.

Jorgensen (1995) used this approach to define two new test methods: Worst Case Testing (Cartesian product of valid boundary values) and Robust Worst Case Testing (Cartesian product of all boundary values). These approaches make this algorithm more economically feasible due to the relatively limited size of the test data domain.

2.2.7.1.1 Permutations and Combinations in Mathematics

All Combinations in testing is different to the concept of Combinations in a branch of mathematics called Combinatorics, which relates to *field placement* rather than test data converge and is a function of the number of fields ***n*** chosen from ***r*** fields that are included in a test case. Consider three input fields *ABC*. There is ${}^{3}C_{3} = 1$ test case, *ABC*, which could be selected if all three fields are included and not repeated. If two fields are included (e.g. if the third is null) then this results in ${}^{3}C_{2} = 3$ test cases, *AB*, *AC* and *BC*. Thus, in the Combinations algorithm, the fields retain their order.

A simular concept is Permutations, which is also a function of ***n*** and ***r***. For three fields *ABC* there are a maximum of ${}^{3}P_{3} = 6$ permutations: *ABC*, *ACB*, *CAB*, *BCA*, *BAC* and *CBA*. Thus, in this algorithm the fields do not necessarily retain their order, though no fields are repeated.

2.2.7.2 Each Choice (EC)

The Each Choice algorithm ensures that all test data values from a set are included in at least one test case, achieving *1-wise* coverage (Grindal, Offutt & Andler 2004). This has been used in EP, BVA (Jorgensen 1995) and the Base-Choice approach (Ammann & Offutt 1994) (see Section 2.2.7.3).

For ***n*** input fields, where field P_i has V_i values, the number of test cases generated by this algorithm is (Grindal et al. 2004):

$$\underset{i=1}{\overset{n}{Max}} V_i$$

This generates ***1*** *to* ***n*** test cases, depending on how many 'uncovered' test data values are included in each new test case. For example, consider three input fields *A*, *B* and *C*, for which test data values *A = [1, 2, 3]*, *B = [1, 2]* and *C = [1]* have been chosen. A minimum of three test cases are required to cover every test data value at least once, resulting in test cases (1, 1, 1), (2, 2, 1) and (3, 1, 1). Additional test cases could also be selected, such as (1, 2, 1) or (3, 2, 1).

2.2.7.3 Base Choice (BC)

The Base-Choice algorithm was proposed by Ammann and Offutt (1994) as an approach to selecting test cases for the Category Partition Method (see Section 2.2.7.6). In this approach, a base test case is

chosen first by assigning the most 'typical' test data values to it, which could be default values, values chosen from equivalence classes or from the anticipated operational profile of the system under test. Test cases are then added to the test set by alternating the value of one field at a time with all other test data values chosen for that field. The size of the resulting test set is a function of $\boldsymbol{n}$ fields, where the $\boldsymbol{i^{th}}$ field has $\boldsymbol{j_k}$ data values (Ammann & Offutt 1994):

$$(\sum_{i=1}^{n} j_k) - n + 1$$

Consider three fields *A*, *B* and *C* with a chosen collection of test data values *A = [1, 2, 3]*, *B = [1, 2]* and *C = [1]*. If 'typical' values resulted in a base-choice test case of (1, 2, 3), then two additional tests would be required by alternating one parameter at a time: (1, 1, 3) and (1, 2, 1). This satisfies *1-wise* coverage and *single error* coverage (Grindal et al. 2004) and provides a formal definition for the typical approach to test case design used in prescriptive methods like EP and BVA, where each test case covers exactly one value from one partition at a time. This results in fewer test cases than All Combinations and ensures that the most important test data values are included in the final test set.

2.2.7.4 Orthogonal Array (OA) (Pair-Wise) Testing

Orthogonal Arrays (OAs) were first used in testing by Mandl in 1985 (Mandl 1985). They originate from the mathematical concept of Latin Squares. An OA is a two-dimensional array in which any two columns contain all combinations of pairs of values and if any pair occurs multiple times then each will appear exactly the same number of times (Copeland 2004). Each row is a tuple of test data values that form one test case, with the entire set satisfying *pair-wise* coverage (Grindal et al. 2004). The standard notation for representing an OA is $\boldsymbol{L_r(N^c)}$, where $\boldsymbol{r}$ is the number of rows, $\boldsymbol{c}$ the number of columns (i.e. fields under test) and $\boldsymbol{N}$ is the maximum number of values that can be chosen for each field (Copeland 2004). For a program with $\boldsymbol{n}$ fields, where field $\boldsymbol{P_i}$ has $\boldsymbol{V_i}$ values, an OA will result in $\boldsymbol{V_i^2}$ test cases, which is calculated as (Grindal et al. 2004):

$$V_i^2 = \left(\underset{j=1}{\overset{n}{Max}} V_j \right)^2$$

OAs can be used to generate test cases from test data values chosen through EP, BVA and ST. They can also be used to test (valid and/or invalid) combinations of conditions for Compatibility Testing (see Glossary for definition). For example, an OA could be designed for testing the compatibility of web-based application with various browsers (e.g. Safari, Internet Explorer, Opera), plug-ins (e.g. Real Player, Media Player), servers (e.g. Microsoft IIS, Apache, Netscape Enterprise) and operating systems (e.g. Windows, Macintosh OSx, Linux)[13]. An OA that achieves pair-wise coverage for this particular set of configuration values is $\boldsymbol{L_9(3^4)}$, which results in nine test cases (Table 2-7). Any pair of columns in the array contains every combination of pairs of values for each of the four fields.

[13] This list of browsers, plug-ins, servers and operating systems is exemplar only; it is not intended to be exhaustive. For example, the OA could be extended to consider versions of each system and other systems (e.g. see (Craig & Jaskiel 2002) and (Cohen et al. 2003)).

Table 2-7: Orthogonal Array for testing an Internet-based application.

Test Case	Browser	Plug-In	Application Server	Operating System
1	Safari	None	IIS	Windows
2	Safari	Real Player	Apache	Macintosh OSx
3	Safari	Media Player	Netscape Enterprise	Linux
4	Internet Explorer	None	IIS	Windows
5	Internet Explorer	Real Player	Apache	Macintosh OSx
6	Internet Explorer	Media Player	Netscape Enterprise	Linux
7	Opera	None	IIS	Windows
8	Opera	Real Player	Apache	Macintosh OSx
9	Opera	Media Player	Netscape Enterprise	Linux

In the OA above (Table 2-7), each column has the same 'range' (i.e. three test data values each). An OA can also have columns with different 'ranges' (Copeland 2004). For example, an $L_{18}(2^1 3^7)$ array contains one column of two values (2^1) and seven columns of three values (3^7), resulting in eighteen test cases (L_{18}). It can sometimes be difficult to choose an OA with a size that fits the number of columns and range of values per column precisely (Copeland 2004). Consider an OA for five fields, each with eight, three, six, three and three data values respectively. A perfectly sized OA would be $L_r(8^1 6^1 3^3)$, but an OA of this size does not exist (Copeland 2004). The next largest size $L_{64}(8^2 4^3)$ is chosen and any field that is not assigned a value after all pairs have been covered can be assigned a random or specific value (Copeland 2004).

2.2.7.5 Specification-Based Mutation Testing

The fundamental concept of mutation testing is that an artefact under test is modified by applying a 'mutation operator' that introduces a specific type of fault. Mutation can be program-based, which can be used to check the ability of a test set to locate certain types of faults, or specification-based to systematically generate black-box test cases.

Program-based mutation was originally proposed by DeMillo, Lipton and Sayward in 1978 (DeMillo et al. 1978). In this approach, mutation operators are used to introduce various types of faults into program source code. Each type of fault is based on the types of mistakes programmers commonly make during development, including replacing a relational operator with an invalid relational operator (e.g. replacing a < sign with > in a conditional statement). Mutants are said to be 'killed' when test cases executed against the mutant and original programs produce different output. Tests that kill mutants are considered to be "effective with respect to mutation" and killed mutants are not executed again against any other test case (Voas & Offutt 1996). If a test case does not locate a mutant (i.e. the mutant and original programs produce the same output) then the test is discarded and new tests are designed to try to detect the mutant code.

The 'mutation score' is a ratio of the number of killed mutants to mutants that are not equivalent to the original program, which is a measure of test set adequacy. A test set is 'mutation-adequate' if the mutation score is 100% (Offutt & Lee 1994). Typically, scores over 90% are difficult to reach and those over 95% are extremely difficult to achieve (Offutt & Liu 1999). Although program mutation showed promise, it was never widely adopted by industry (Ng et al. 2004), possibly because it suffers a combinatorial explosion

that can result in high testing cost. Also, since program mutation is an approach to testing test quality, it is often seen by industry professionals as an additional and unnecessary expense. The number of mutants that can be generated for any given program is $O(N^2)$, where N is the number of variable references in the program (Acree 1980). In one experiment that used the Mothra automated mutation testing system, 951 mutants were generated for a simple 30 line triangle classification program (Offutt & Lee 1994).

Specification-based mutation was originally proposed by Budd and Gopal in 1985 (Budd & Gopal 1984). In this approach test cases are designed by mutating program specifications. Depending on the language used in the specification, specification mutants can look similar to program mutants, such as the replacement of a relational operator with an invalid operator. Specification mutation has been applied to various specifications languages, including predicate calculus, state charts, model checkers, Boolean algebra, Extensible Markup Language (XML) and BNF (see Table 2-8).

Budd and Gopal (1984) defined five mutation operators for predicate calculus specifications. Fabbri, Maldonado, Sugeta and Masiero (1999) experimented with specification mutation to validate statechart specifications, defining a mutation operator set that was taken as a fault model. Their goal was to investigate methods of selecting useful test sets and test methods for ensuring that a specification and program are thoroughly tested. Ammann, Black and Majurski (1998) developed a model checker called Symbolic Model Verifier (SMV) to automatically generate specification mutants they referred to as 'complete' in the sense that they included inputs and expected results. Ammann and Black (1999) found that in order to make mutation with of a model checker possible, specifications need to be decomposed to lower language levels. They investigated an approach to reducing large-scale state machines using a technique called "finite focus," which allowed tests to be automatically derived for very large software systems. They proved that finite focus was a sound reduction technique, producing smaller mutation-adequate test sets.

Woodward (1993) defined nine mutation operators for Boolean algebra specifications by examining errors made in the assignments of 59 third year and 20 postgraduate university students in a course on software engineering. Lee and Offutt (2001) applied specification mutation to test the semantic correctness of XML messages communicated between web components. Syntactic errors were not considered as XML parsers that are freely available can be used to test for this. Since XML allows the definition of unique languages for each Document Type Definition (DTD), they could not define universal mutation operators that could be applied to all DTDs. Instead, they defined a generic class of operators and instantiated the class to create DTD-specific operators that were applied to the constraints of the language to produce mutants. Two example mutation operators were defined, although future work included the aim of identifying additional mutation operators.

Table 2-8: Specification-based mutation operators for various specification languages.

Mutation operators for predicate calculus specifications (Budd & Gopal 1984):

1. Relational operator replacement
2. Arithmetic operator replacement
3. Operand increment/decrement
4. Operand substitution
5. True/false replacement*
6. Logical operator replacement*
7. Quantifier changes*

* Operators 6, 7 and 8 were discarded by Budd and Gopal as they were either too expensive, subsumed by other operators or were not useful (Budd & Gopal 1984).

Mutation operators for Boolean algebra specifications (Woodward 1993):

1. OpsAttrDel – delete operator attribute
2. OpsAttrIns – insert operator attribute
3. OpsAttrRpl – replace an operator attribute
4. EqnsOpRplOp – replace non-constant operator with non-constant operator
5. EqnsOpRplCon – replace non-constant operator with constant operator
6. EqnsOpsRplVar – replace non-constant operator with VAR
7. EqnsConRplOp – replace constant operator with non-constant operator
8. EqnsConRplCon – replace constant operator with constant operator
9. EqnsConRplVar – replace constant operator with VAR
10. EqnsVarRplOp – replace VAR with non-constant operator
11. EqnsVarRplCon – replace VAR with constant operator
12. EqnsVarRplVar – replace VAR with VAR
13. EqnsDel – delete an equation
14. EqnsIfDel – delete conditional part of equation
15. EqnsOrd – reorder an equation

Mutation operators for BNF specifications (Murnane 1999, Murnane & Reed 2001):

1. substitute one terminal for another terminal
2. substitute n - 1 terminals for n other terminals
3. substitution one non-terminal for another non-terminal

Mutation operators for finite state machines (Fabbri et al. 1999):

1. wrong-start-state
2. arc-missing
3. event-missing
4. event-extra
5. event-exchanged
6. destination-exchanged
7. output-missing
8. output- exchanged
9. state-missing

Mutation operators for extended finite state machines (Fabbri et al. 1999):

1. expression deletion
2. boolean expression negation
3. term associativity shift
4. arithmetic operator by arithmetic operator
5. relational operator by relational operator
6. logical operator by logical operator
7. logical negation
8. variable by variable replacement
9. variable by constant replacement
10. constant by required constant replacement
11. constant by scalar variable replacement

Mutation operators for statechart features (Fabbri et al. 1999):

1. transition's history deletion
2. transition with history by transition replacement
3. history-missing
4. h by h* replacement
5. h* by h replacement
6. h-extra
7. h*-extra
8. in(s) condition-missing
9. in(s) condition state replacement
10. not-yet(e) condition-missing
11. not-yet(e) condition event replacement
12. exit(s) event-missing
13. exit(s) event state replacement
14. entered(s) event-missing
15. entered(s) event state replacement
16. broadcasting origin transition replacement
17. broadcasting destination transition replacement

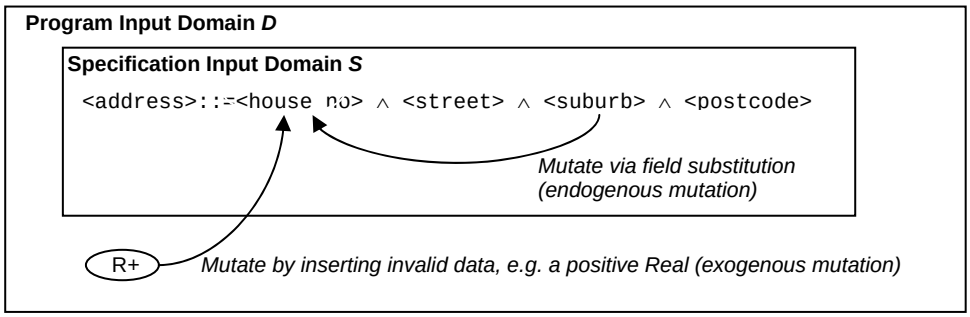
Specification mutation has also been applied to BNF specifications (Murnane 1999, Murnane & Reed 2001) (Figure 2-10, Figure 2-11). In this approach, mutation operators are applied to a specification to construct test cases, where each (terminal) input field is substituted for every other field, one substitution per test (see Figure 2-11, 'endogenous' mutation). When fields are substituted for each other but not for themselves, the number of mutants is ***N(N-1)***, which is $O(N^2)$, where ***N*** is the number of input fields in the test case. Double-defect mutants can be selected by substituting two fields per test. Although this may result in a more rigorous test set, the number of resulting tests is ***N!***, which is prohibitively large. Specification

mutation can be performed by substituting non-terminal fields or invalid datatypes into the fields under test, which corresponds to the selection of invalid datatypes in EP (Figure 2-11, see 'exogenous' mutation).

Figure 2-10: A simplified version of an address specification expressed in BNF.

```
    <address>  ::= <house_number> ∧ <street> ∧ <suburb> ∧ <postcode>
<house_number> ::= [1 – 999]
     <street>  ::= {Annensen Court, Aaran Close, …, Zelda Court}
     <suburb>  ::= {Abbotsford, Aberfeldie, …, Yooralla, Yuroke}
   <postcode>  ::= [200 | 221 | 800 | 801 | 804 | 810 | … | 9726 | 9728 | 9729]
            ∧  ::= [ ]  #i.e. one spaces
```

Figure 2-11: Example of specification-based mutation for the address parser.



2.2.7.6 Commonality within Combinatorial Test Methods

Most combinatorial test methods have two attributes in common: they consist of algorithms for test case design and the expected number of test cases can usually be calculated, or at least estimated. Nonetheless, there is no standard notation for describing them in the literature. ***Notational and terminological differences*** between the methods could be resolved by defining a common notation for all black-box testing methods (see Chapter 3 for one such approach).

2.2.8 The Category Partition Method (CPM)

Specifications that are written in natural language can be "wordy and unstructured," which can make test case design difficult (Ostrand & Balcer 1988). The Category Partition Method (CPM) was developed by Ostrand and Balcer (1988) to formalise the documentation of black-box test cases in a language they named the 'Test Specification Language' (TSL). CPM comprises six steps, as follows:

1. Decompose the specification into functional units that can each be tested separately.
2. Identify 'categories' for each functional unit, which are essentially input fields and environmental conditions whose state can affect functional unit behaviour. Each category is then partitioned into disjoint equivalence classes called 'choices.'
3. Identify the expected result of combinations of choices and constraints, which limit how the occurrence of a choice in one category can restrict the choices in another.

4. Document categories and choices in TSL (see Figure 2-12 and Figure 2-13), facilitating automatic generation of test cases through the *TSL processor* tool, by defining the relationships between inputs and outputs. Two types of specifications can be produced: unrestricted and restricted. *Unrestricted specifications* contain the Cartesian-product of categories. In *restricted specifications*, choices are annotated with constraints describing relationships between choices and expected outputs, to limit the ways in which 'test frames' (Figure 2-14) are constructed by making the values that can be selected from one category dependant on the value of another. Restricted specifications result in fewer test frames. Test frames (Figure 2-14) describe the structure of each test case.
5. Analyse the test frames to ensure that no 'impossible' combinations of choices have been defined (see Figure 2-14) and to ensure that they result in an acceptable number of tests. To achieve this, steps four and five can be repeated until the test frames are appropriately refined.
6. Create test cases by selecting one value from the choices of each category within each test frame. Test procedures[14] can be constructed by joining together sequences of related test cases.

Appendix A provides a complete example from (Ostrand & Balcer 1988), which illustrates these six steps being applied to an example specification for a 'find' command.

Balcer, Hasling and Ostrand consider TSL and the TSL processor to be two of the most beneficial aspects of the CPM; translating specifications into TSL facilitates identification of ambiguous and inconsistent requirements, allowing specification faults to be identified prior to test case design, while the TSL processor makes test design and maintenance more efficient and precise (Ostrand & Balcer 1988, Balcer, Hasling & Ostrand 1989). Employing the use of TSL early in the software development lifecycle could allow testers to become involved in the program specification and review process, which could result in clearer and more testable requirements, and would also allow testers to learn about system under test earlier, allowing them to be better-prepared for testing.

[14] Test procedures are test cases that are joined together to test sequences of functionality within a program. They include procedural or environmental requirements that must be met in order for the tests to be executed. For example, individual test cases could be written for testing the components of an online banking application, which separately test the login screen, funds transfer and logout; these three tests could be joined together into a test procedure that tests a scenario in which a user logs in, performs a funds transfer and logs out. In (Ostrand & Balcer 1988) and (Balcer, Hasling & Ostrand 1989), Test Procedures are referred to as Test Scripts, whereas in this thesis, Test Scripts are considered to be automated test procedures (e.g. see definition of Test Script in the Glossary).

Figure 2-12: Structure of a test specification expressed in the Test Specification Language (TSL) (Balcer, Hasling & Ostrand 1989).

```
TEST <test-name>
       [<description-string>]
       [SETUP {<string>}]
       FORM {<string>}
       CLEANUP {<string>}]
PARAMETER <param-name>
       [<description-string>]
       [<setup-cleanup>]
              * <choice-1>
                     [<value-list>]
                     [<setup-cleanup>]
              …
              * <choice-n>
                     [<value-list>]
                     [<setup-cleanup>]
```

```
ENVIRONMENT <environment-name>
       [<description-string>]
       [<setup-cleanup>]
              * <choice-1>
                     [<setup-cleanup>]
              …
              * <choice-n>
                     [<setup-cleanup>]
RESULT <result-name>
       [<description-string>]
       [<setup-cleanup>]
       [VERIFY <verification-code>]
       IF <result-expression-1>
              …
       IF <result-expression-n>
```

Figure 2-13: Specification for a 'find' command and the corresponding Test Specification Language specification (Ostrand & Balcer 1988).

Specification in Natural Language*	Corresponding TSL Specification
Command find **Syntax** find <pattern> <file> **Function** The find command is used to locate one or more instances of a given pattern in a text file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern in written only once, regardless of the number of times the pattern occurs in it. The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes (“). To include a quotation mark in the pattern, two quotes in a row (“”) must be used. **Examples** find john myfile displays lines in the file **myfile** which contain **john** find “john smith” myfile displays lines in the file **myfile** which contain **john smith** find “john”” smith” myfile displays lines in the file **myfile** which contain **john” smith** ** Note: this specification is written exactly as it appears in (Ostrand & Balcer 1988).*	**Parameters** **Pattern size** empty single character many characters longer than any line in the file **Quoting** pattern is quoted pattern is not quoted pattern is improperly quoted **Embedded white spaces** no embedded white spaces one embedded white space several embedded white spaces **Embedded quotes** no embedded quotes one embedded quote several embedded quotes **File name** good file name no file with this name omitted **Environments** **Number of occurrences of pattern in the file** none exactly one more than one **Pattern occurrences on target line** # assumes line contains the pattern one more than one

Figure 2-14: Example of a contradictory test frame (Ostrand & Balcer 1988).

Pattern size: empty*
Quoting: pattern is quoted*
Embedded blanks: several embedded white spaces
Embedded quotes: no embedded quotes
File name: good file name
Number of occurrences of pattern in file: none
Pattern occurrences on the target line: one

**** contradiction**: pattern cannot be empty if it is quoted*

Ostrand and Balcer (1988) compared CPM to Goodenough and Gerhart's Condition Table method (1975), Elmendorf's Cause-Effect Graphing (1974), Weyuker and Ostrand's Revealing Subdomains (1980) and Richardson and Clarke's Partition Analysis (1981). Although the concepts of partitioning categories into choices and selecting data values from each choice to construct test cases are essentially the same process as partitioning input fields and selecting test data values during EP and BVA, CPM has not been compared to or integrated with these methods. For example, in Figure 2-13 the category 'pattern size' is partitioned into choices *empty*, *single character*, *many characters*, *longer than any line in the file*, which could be selected more prescriptively through EP and BVA, since these methods were designed to guide testers specifically in the selection of this type of test data. As CPM does not provide prescriptive guidelines for identifying choices or test data values from each choice, this approach could be improved by combining TSL with the test case design rules from EP, BVA and ST.

2.2.9 Classification Trees

Classification Trees were originally proposed by Grochtmann et al. in 1993 as an improvement to CPM (Grochtmann & Grimm 1993, Grochtmann, Grimm & Wegener 1993). In this approach, categories and choices are represented in a tree, with categories as root nodes, choices as leaf nodes and intermediate nodes representing the decomposition of categories into sub-categories (Figure 2-15). Test cases are constructed in the same way as CPM (i.e. by taking the Cartesian product of choices and removing impossible choice combinations), although other combinatorial test methods could be used to define test cases (see Section 2.2.7). Test cases are recorded as horizontal lines beneath the tree that intersect with choices included in each test case, with black dots at the line's intersection marking the selection of a test data value for inclusion in a test case. Some automated support for Classification Trees is provided through the Classification Tree Editor (see Section 2.7).

Classification Trees have been improved by a number of researchers. Chen, Poon and Tse (1999) developed an algorithm for removing duplicate tree nodes. For example, for resolving the duplicate node in Figure 2-15, in which 'Price of Ticket' appears under the 'Class of Seat' and 'Total Mileage' branches. Chen and Poon (1996) developed classification-hierarchy tables for capturing and documenting the hierarchy of classification trees. Singh, Conrad and Sadeghipour (1997) built Classification Trees from formal Z specifications. As with CPM, the Classification Tree approach could be improved by combining prescriptive test case design rules from EP, BVA and ST with the visual representation provided by the Classification Tree

Figure 2-15: Classification Tree for an airline bonus points programme (Chen, Poon & Tse 1999).

*Specification**

The software under test is the program *bonus* being developed for Number-One Airline. It calculates the bonus points earned by passengers from their trips. Passengers can then claim various benefits such as free accommodation in leading hotels using the bonus points awarded. The program calculates the bonus points according to the following specification.

(1) Classes of Seats

There are three classes of seats, namely first, business, and economy.

(2) Upgrading of Classes

Passengers holding an economy-class ticket are eligible for upgrading their tickets to a business class free of charge, provided that:

(a) there are vacancies in the business class,
(b) the passengers are holding a frequent-flyer card, and
(c) the total mileage for the trip is less than 1000.

Under no circumstances can an economy-class or business-class ticket be upgraded to the first class.

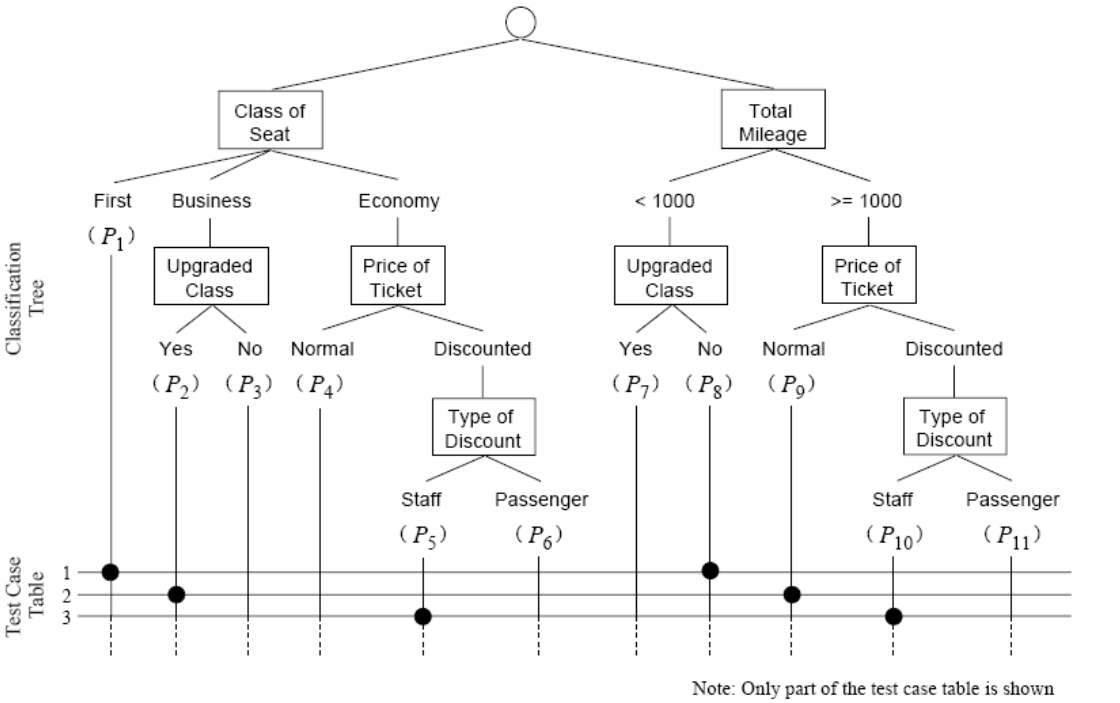
(3) Discounts

Discounts are only available to:

- economy-class tickets, and
- the total mileage for the trip is not less than 1000. There are two types of discounts, namely staff discount and passenger discount.

For (2*c*) and (3*b*), any distance less than one mile will not be counted. The number of bonus points earned will be calculated from the combination above.

Classification Tree



** This specification and corresponding Classification Tree are shown here exactly as they appear in (Chen et al. 1999).*

2.3 Summary of Black-Box Test Case Design Steps and Methods

As outlined in Section 2.2, each black-box testing method typically focuses on one of the following four steps of test case design:

1. Partition the input and output domains of the program into sets of equivalent data, by applying Data-Set Selection Rules.
2. Select test data values from each partition by applying Data-Item Selection Rules.
3. Optionally manipulate[15] the test data values by applying Data-Item Manipulation Rules.
4. Construct test cases by creating combinations of test data values via the application of Test Case Construction Rules.

Although some methods touch on more than one of these steps (e.g. EP provides guidance on partitioning, test data selection and test case design), a method's strength usually lies in just one of these steps, as outlined below (Table 2-9).

Table 2-9: Test case design steps covered by black-box testing methods.

	Four Steps of Test Case Design			
Black-Box Testing Method	**Step 1.** Partition the input and output domains of the program by applying DSSRs to each field	**Step 2.** Select test data values from each partition by applying DISRs to each partition	**Step 3.** Optionally manipulate test data values by applying DIMRs to each test data value	**Step 4.** Construct test cases by combining test data values by applying of TCCRs
All Combinations				✓
Base Choice				✓
Boundary Value Analysis	Often uses DSSRs from EP	✓		Often uses TCCRs from EP
Category Partition Method				✓
Classification Trees	Non-systematic	Non-systematic		✓
Each Choice				✓
Equivalence Partitioning	✓	✓		✓
Error Guessing	Non-systematic	Non-systematic	Non-systematic	Non-systematic
Exploratory Testing	Non-systematic	Non-systematic	Non-systematic	Non-systematic
Orthogonal Array Testing				✓
Random Testing	Often uses DSSRs from EP	✓		
Specification-Based Mutation Testing				✓
Syntax Testing	Implicitly performed	✓	✓	✓
Test Matrices, Catalogues & Categories	Can provide guidance	Can provide guidance	Can provide guidance	

[15] For example, the definition of Syntax Testing in (BS 7925-2) includes a test case design rule that 'mutates' test data values. In this thesis the word 'manipulate' is used to describe any test case design rule that derives an invalid test data value by altering a valid test data value (e.g. by removing a character from the end of a valid keyword).

2.4 Combining Testing Methods

A number of novel black-box methods have been proposed by combining other black-box testing methods. Howden developed a new method called Functional Testing, which combines test case design rules from BVA, EP, EG with All Combinations (Howden 1980). Grindal et al. (2004) produced two new combinatorial methods by combining Base Choice with Orthogonal Arrays and with Heuristic Pair-Wise Testing. Jorgensen (1995) combined BVA with All Combinations to produce Worst Case Testing and with Robustness Testing to produce the Worst Case Robustness Testing (see Section 2.2.2). Jorgensen also extended Myers' (1979) and Mosley's (1993) definitions of EP by combining EP with combinatorial methods to produce Traditional, Strong and Weak EP (Jorgensen 1995). Thus, creating new black-box methods from the 'atomic elements' of existing methods is something that has been achieved in the past (see Chapter 3 for the Systematic Method Tailoring approach to customisation).

2.5 Approaches to Test Method Selection

As Beizer observed (1990), the application of "unsuitable" test methods can result in the design of inappropriate test cases. Effective test method selection approaches are an essential part of effective black-box testing. In the subsections below, various approaches to test method selection are described, which base selection on:

1. the steps of the test case design process targeted by the method (Section 2.5.1);
2. the classes of error detected by the method (e.g. see (Jorgensen 1995)) (Section 2.5.2);
3. a characterisation schema that differentiates between the 'functionality' of each method (Vegas et al. 2003) (Section 2.5.3);
4. a decision table that classifies the conditions under which each black-box test method should be selected (Jorgensen 1995) (Section 2.5.4); and
5. test effectiveness (Section 2.5.5).

2.5.1 Test Method Selection by Test Design Step

Each of the black-box methods discussed in this chapter targets one or more of the following four steps of black-box test case design (e.g. see Section 2.3, Table 2-9):

1. partitioning the input and output domains (i.e. using Data-Set Selection Rules);
2. selecting test data values from each partition (i.e. using Data-Item Selection Rules);
3. manipulating the chosen test data values (i.e. using Data-Item Manipulation Rules); and
4. constructing test cases (i.e. using Test Case Construction Rules).

Black-box testing methods could be chosen for their ability to target one ore more of these steps. On the other hand, current descriptions of black-box testing methods do not clearly identify which steps of the test

case design process are targeted by each method. Mapping test case design rules to specific types of input fields has been achieved by Test Catalogues, Categories and Matrices. However, these approaches do not describe the black-box testing methods that each test case design rule belongs to. Consequently, testers who are unfamiliar with the specific mechanics of each black-box testing method may find it difficult to map between the rules in a Test Catalogue, Category or Matrix with the black-box testing methods they were derived from. The Atomic Rules approach and Systematic Method Tailoring are possible solutions to these problems.

2.5.2 Test Method Selection by Error Class

Black-box testing methods can be chosen for their ability to detect certain classes of error (Jorgensen 1995). For instance, BVA targets the mishandling of boundary values and EP and ST can be used to test a program's input parsing capabilities. Test Catalogues, Categories and Matrices can facilitate this mapping. On the other hand, they do not include guidelines to assist testers in creating new mappings of input fields to test case design rules for each program under test.

2.5.3 Test Method Selection via Vegas et al.'s Characterisation Schema

Vegas et al. (2003, 2004) developed a characterisation schema for classifying black-box and white-box testing methods, which utilises information about the methods, the system under test, the test environment and knowledge of tester abilities to facilitate selection of the "best suited" methods for testing (see Table 2-10, Table 2-11). The schema has been instantiated for four types of methods:

1. black-box testing methods: BVA and RT (see Sections 2.2.2 and 2.2.4);
2. white-box testing methods: sentence, decision, path and thread coverage (e.g. see (Myers 1979, Pressman 1992));
3. data-flow methods: all-c-uses, all-p-uses, all-uses, all-du-paths and all-possible-rendezvous (e.g. see (Pfleeger 2001)); and
4. mutation testing: standard and selective program mutation (e.g. see (Bottachi & Mresa 1999)).

The characterisation schema clearly distinguishes between these four types of methods, which can assist with identifying the conditions under which a black-box testing method should be used over a white-box method. Yet it does not clearly identify the conditions under which one specific black-box testing method should be used over another. For example, there is very little difference between the definitions of BVA and RT (see Table 2-11). This schema could be enhanced by additional attributes that clearly differentiate between the individual test case design rules that are included in each black-box testing method. Since the schema has not been instantiated for methods like EP and ST, it could also be improved by deriving schema instances for these methods.

Table 2-10: Characterisation schema for assist with test method selection (Vegas et al. 2003).

Level	Element	Attribute	Value
Tactical	Objective	Purpose	Type of evaluation and quality attribute to be tested in the system
		Defect type	Defect types detected in the system
		Effectiveness	What capability the set of cases should have to detect defects
	Scope	Element	Elements of the system on which the test acts
		Aspect	Functionality of the system to be tested
Operational	Agents	Knowledge	Knowledge required to be able to apply the technique
		Experience	Experience required to be able to apply the technique
	Tools	Identifier	Name of the tool and the manufacturer
		Automation	Part of the technique automated by the tool
		Cost	Cost of tool purchase and maintenance
		Environment	Platform (software and hardware) and programming language with which the tool operates
		Support	Support provided by the tool manufacturer
	Technique	Comprehensibility	Whether or not the technique is easy to understand
		Cost of application	How much effort it takes to apply the technique
		Inputs	Inputs required to apply the technique
		Adequacy criterion	Test case generation and stopping rule
		Test data cost	Cost of identifying the test data
		Dependencies	Relationships of one technique with another
		Repeatability	Whether two people generate the same test cases
		Sources of Information	Where to find information about the technique
	Test Cases	Completeness	Coverage provided by the set of cases
		Precision	How many repeated test cases the technique generates
		Number of generated cases	Number of cases generated per software size unit
	Object	Software type	Type of software that can be tested using the technique
		Software architecture	Development paradigm to which it is linked
		Programming language	Programming language with which it can be used
		Development method	Development method or life cycle to which it is linked
		Size	Size that the software should have to be able to use the technique
Use	Project	Reference projects	Earlier projects in which the technique has been used
		Tools used	Tools used in earlier projects
		Personnel	Personnel who worked on earlier projects
	Satisfaction	Opinion	General opinion about the technique after having used it
		Benefits	Benefits of using the technique
		Problems	Problems with using the technique

Table 2-11: Characterisation schemas instantiated for Boundary Value Analysis and Random Testing (Vegas et al. 2003).

Level	Element	Attribute	Boundary Value Analysis	Random Testing
Tactical	Objective	Purpose	Find defects	Find defects
		Defect type	Any	Any
		Effectiveness	Finds 55% of defects	42% probability of finding a fault
	Scope	Element	Any	Units (functions), complete systems
		Aspect	Any	Any
Operational	Agents	Knowledge	None	None
		Experience	None	Errors people usually make
	Tools	Identifier	-	-
		Automation	-	-
		Cost	-	-
		Environment	-	-
		Support	-	-
	Technique	Comprehensibility	High	High
		Cost of application	Low	Low
		Inputs	Code specification	Code specification
		Adequacy criterion	Functional: Boundary Value Analysis	Functional: Random testing
		Test data cost	Low	Low
		Dependencies	When applied with black-box effectiveness may rise to 75%	Might (and should) be completed with other technique
		Repeatability	No	No
		Sources of Information	(Beizer 1995, Sommerville 2001)	(Beizer 1995, Myers 1979, Sommerville 2001, Pfleeger 2001)
	Test Cases	Completeness	-	-
		Precision	-	-
		Number of generated cases	Depends on the complexity of the input domain	As many as wanted
	Object	Software type	Any	Any
		Software architecture	Any	Any
		Programming language	Any	Any
		Development method	Any	Any
		Size	Any	Any
Use	Project	Reference projects	-	-
		Tools used	-	-
		Personnel	-	-
	Satisfaction	Opinion	-	Fine for complementing other methods or acceptance testing
		Benefits	-	It is very easy to apply
		Problems	-	- Although mean effectiveness is high, variance is also high - Maximum benefits obtained with people with experience

2.5.4 Test Method Selection via Jorgensen's Decision Table

As an initial step towards developing an "expert system" for black-box test method selection, Jorgensen (1995) developed a decision table based on the goals of testing and characteristics of the system under test, which could be used for informing test method selection (Table 2-12). While the decision table provides insight into the high-level differences between some black-box methods, it does not cover prescriptive methods like ST and RT or non-prescriptive approaches like EG and ET. It also does not explain the differentiation between Physical and Logical variables.

Table 2-12: Decision table for selecting black-box test methods (Jorgensen 1995).

Conditions	**Rules**									
c1. variables are	Physical					Logical				
c2. independent variables?	Y				N	Y				N
c3. fault assumption is	Single		Multiple		-	Single		Multiple		-
c4. exception handling?	Y	N	Y	N	-	Y	N	Y	N	-
Actions										
a1. boundary value analysis		Y								
a2. robustness testing	Y									
a3. worst case testing				Y						
a4. robust worst case			Y							
a5. traditional equivalence class	Y		Y			Y		Y		
a6. weak equivalence class	Y	Y	Y			Y	Y			
a7. strong equivalence class				Y	Y			Y	Y	Y
a8. decision tables					Y					Y

2.5.5 Test Method Selection by Effectiveness

Test methods can, in principle, be selected by their effectiveness (e.g. failure-detection effectiveness, defined in Chapter 1), which can be determined through empirical study and theoretic analysis. Yet, not enough empirical study has been carried out into test method effectiveness for this to be considered a reliable approach for decision making (Vegas et al. 2003). While many researchers have experimentally compared the effectiveness of white-box and black-box testing methods (e.g. see (Myers 1978, Basili & Selby 1987, Kamsities & Lott 1995, Wood et al. 1997)) and random testing to white-box and grey-box partition testing (e.g. see (Duran & Ntafos 1984, Jeng & Weyuker 1989, Weyuker & Jeng 1991, Hamlet & Taylor 1990, Tsoukalas et al. 1993, Chen & Yu 1994, Ntafos 1998, Gutjahr 1999)), there has been less research into the effectiveness of purely black-box methods like EP, BVA and ST. Also, most studies focus on test set quality and not on aspects that affect novice testers, such as ease of adoption (i.e. test method learnability). Instead, they focus on a variety of quantitative metrics for assessing effectiveness, such as:

- fault detection effectiveness (i.e. number faults detected / total number of known faults) (Reid et al. 1999);
- faults detected per severity level (e.g. critical versus cosmetic faults) (Itkonen et al. 2007);

- test efficiency (i.e. number faults detected / total time spent in testing) (Reid et al. 1999); and
- source code coverage (Seo & Choi 2006).

Ideally, these metrics could form a basis for test method selection[16]. For example, if the aim of testing is to detect more failures, then theoretically, a method that has been proven through experimentation to have a higher failure detection ratio could be selected for use. On the other hand, since each experiment typically uses a different set of metrics, it can be difficult to make accurate comparisons between them and to draw reliable conclusions. More experimentation into test method effectiveness is required before this can be used as an accurate approach for test method selection.

Another issue is that while non-prescriptive approaches like EG are believed by some to be among the most popular in industry (Jorgensen 1995), few studies have compared the effectiveness of prescriptive black-box testing methods (e.g. as they are described in textbooks) to those used by professional testers in industry. Bach (2001) argues that "computer scientists are not qualified to study [the use of non-prescriptive approaches like] ET, because to study ET is to study how people think: cognitive psychology", suggesting that it could be challenging to assess how professional testers actually perform testing. This view is supported by Kaner et al. (2001) who states that the type of thinking that is required during ET is similar to that which is required in fields like sociology, and that textbooks on psychology (e.g. (Koslowski 1996)) explain why testing is "more than simply looking at external behaviour and checking it against simple expectations" (Kaner et al. 2001). Despite these caveats, empirical evaluation of black-box testing methods is necessary to provide evidence-based decision making. As a result, relevant experiments from both industry and academia are discussed below.

Reid (1997) is one of the only researchers to empirically compare pure black-box EP, BVA and RT, where effectiveness was based on the probability that a test case would detect a fault that was previously found in a system already in use. Existing defect reports were used to identify known faults in the seventeen system modules, where each module contained one fault (i.e. seventeen faults in total). Although all test cases were derived by the primary researcher, the approach used was to derive all possible inputs that satisfied the test method (e.g. derive all possible equivalence classes for all program modules). The hypotheses were that BVA is more effective than EP and that both are more effective than RT. Reid did find BVA to be far more effective than EP, but it required more than three times as many test cases (Table 2-13). Only eight random test cases per module were required for RT to be as effective as EP. On the other hand, 50,000 random tests had to be selected for it to be as effective as BVA. For test case construction, no significant difference was found between minimised and one-to-one BVA, while minimized EP was slightly less effective than one-one-one EP.

[16] Ntafos (1988) used the level of automated support available to determine effectiveness; however, this is qualitative.

Table 2-13: Results of an empirical comparison of EP, BVA and RT (from (Reid 1997)).

Measure	EP (1 to 1)	EP (Minimised)	BVA (1 to 1)	BVA (Minimised)	Random Testing
Probability of fault detection	33%	31%	73%	79%	12%
Average number faults detected	5.7	5.3	12.4	13.4	2
Average number test cases required to detect one fault	7.6	4.9	25.1	13.6	*Not stated*

Ostrand and Balcer (1988) used CPM to derive black-box test cases for ninety-one high level functions of a configuration management system, which was implemented in 35,000 lines of Ada. One tester produced a TSL specification for all ninety-one requirements, while four others reviewed them for inconsistencies. The TSL processor produced 1,022 test cases. Writing the TSL specifications took around three weeks, during which test script writing took the most time (although the researchers did not quantify this observation). Test execution took around two weeks, during which modifications to test scripts and specifications were required. In total, thirty-nine program faults were detected, showing that CPM is effective for defect detection, though it could also be argued that it is costly in terms of the time taken to produce test suites. Ostrand and Balcer (1988) did not compare this outcome with other experiments.

Yu et al. (2003) conducted an experiment with 104 final-year computer science students, who had at least one year of full-time work experience. The aims were to identify test methods the students initially chose to use for testing a program they coded themselves and to compare their opinions of those methods to the Classification Trees approach, which they were taught after the initial coding and testing phases. The advantages that were reported by the students include that the Classification Tree approach is "systematic" (63% of the group) and that the visual representation provided by the trees gives the approach an advantage over other white-box and black-box methods, as they are easy to read and understand and they illustrate relationships between test cases (54% of the group). One of the reported disadvantages was that Classification Trees can become too large and complicated if specifications are not properly decomposed. Other disadvantages were that each tester may produce different test sets from the approach and that test set quality depends on specification quality, though these problems apply to all prescriptive black-box methods.

A number of experiments have identified possible relationships between test effectiveness and tester experience. These include experiments by Lauterbach and Randall (1989) and Itkonen et al. (2007). Lauterbach and Randall (1989) carried out a case study with four professional testers who used three static methods (code reviews, error and anomaly detection and structure analysis) and three dynamic methods (white-box branch testing, black-box testing and RT), although the paper did not name which specific black-box methods were used. The metrics utilised were defect detection effectiveness (i.e. the percentage of known defects detected by a method) and effort required to conduct testing. While they found that black-box testing resulted in lower levels of code coverage, they also found that the choice of tester had a greater impact on test effectiveness than did the choice of test method. This suggested that tester experience can have a significant impact on testing effectiveness, which is an issue that can be masked by using students in experimentation.

Itkonen et al. (2007) compared the effectiveness of ET to EP, BVA and combinatorial testing in an experiment involving seventy-nine advanced software engineering students. Although they found no significant difference between numbers of faults detected by the non-prescriptive and prescriptive approaches, they found that the testers were able to detect more obvious and more obscure faults through ET, as well as more user interface and usability problems. On the other hand, they found more technical faults of a less severe nature through prescriptive testing. It could be argued that the group's inexperience with testing reduced their effectiveness during ET and if more experienced "pathological" testers (Reed 2007) were used they may have been able to produce more effective test cases. This view is supported by the findings of the Lauterbach/Randall study (1989).

Itkonen and Rautiainen (2005) conducted an industry-based case study with six software testers from three software development companies, which were code-named Mercury (1 participant), Neptune (4 participants) and Vulcan (2 participants) who were already using ET. Although the findings reported by Itkonen and Rautiainen (2005) are insightful and support a number of advantages and disadvantages to ET reported here, as only six participants were used in the case study, the results can only be considered to be indicative. The participants did not derive test cases during the study. Instead, data was collected on the types of defects and defect counts identified with ET in the past. The average number of defects detected per hour was higher for Neptune and Mercury, which may have been due to them using session-based ET as this approach allows testers to remain uninterrupted and focussed throughout testing. On the other hand, the participants from Vulcan who were using traditional ET did not feel that being interrupted altered their test effectiveness. Itkonen and Rautiainen could not confirm whether Vulcan's metrics were accurate. The metrics from Mercury may have been affected by the maturity of the system under test, as their product was relatively new and was likely to contain more defects. Nonetheless, Itkonen and Rautiainen concluded that ET did improve test productivity, particularly when testing complicated aspects of a system. They considered ET to be effective for defect detection. For example, in this study, 4.8 and 8.7 defects per hour were detected at the two companies using session-based ET, compared to less than 3 defects per hour in a study of Use Case Testing (Anderson et al. 2003) and 2.47 defects per hour in a case study of functional testing (Wood et al. 1997). Also, 15% of faults detected at Mercury were considered to be "serious."

Interestingly, none of the participants from the Itkonen/Rautiainen case study had any prior training in software testing. Although they reported setting goals for ET, none claimed to use any prescriptive black-box testing methods, despite the fact that they reported testing with combinations of inputs and boundary values, suggesting that the test case design rules they used overlap with prescriptive black-box methods. Interviews of 40 to 70 minutes were conducted with the participants using a standard questionnaire (Table 2-14). One of the challenges was identifying testers with enough domain knowledge that enabled them to use the system under test like a professional user. Itkonen and Rautiainen questioned what effect that domain knowledge, testing experience and testing training has on the defect detection ability of the tester, as they found that each participant tested the software differently. In addition, all three companies reported that assessing test coverage was a problem with ET. Recording the test case design rules used and the domain knowledge utilised during ET could be one solution, as it would enable this information to be shared with novice testers (see Chapter 3 for relevant approaches).

Ideally, metrics like failure-detection effectiveness could be used as a basis for test method selection. Realistically, more experimentation is required before this can be used for decision making.

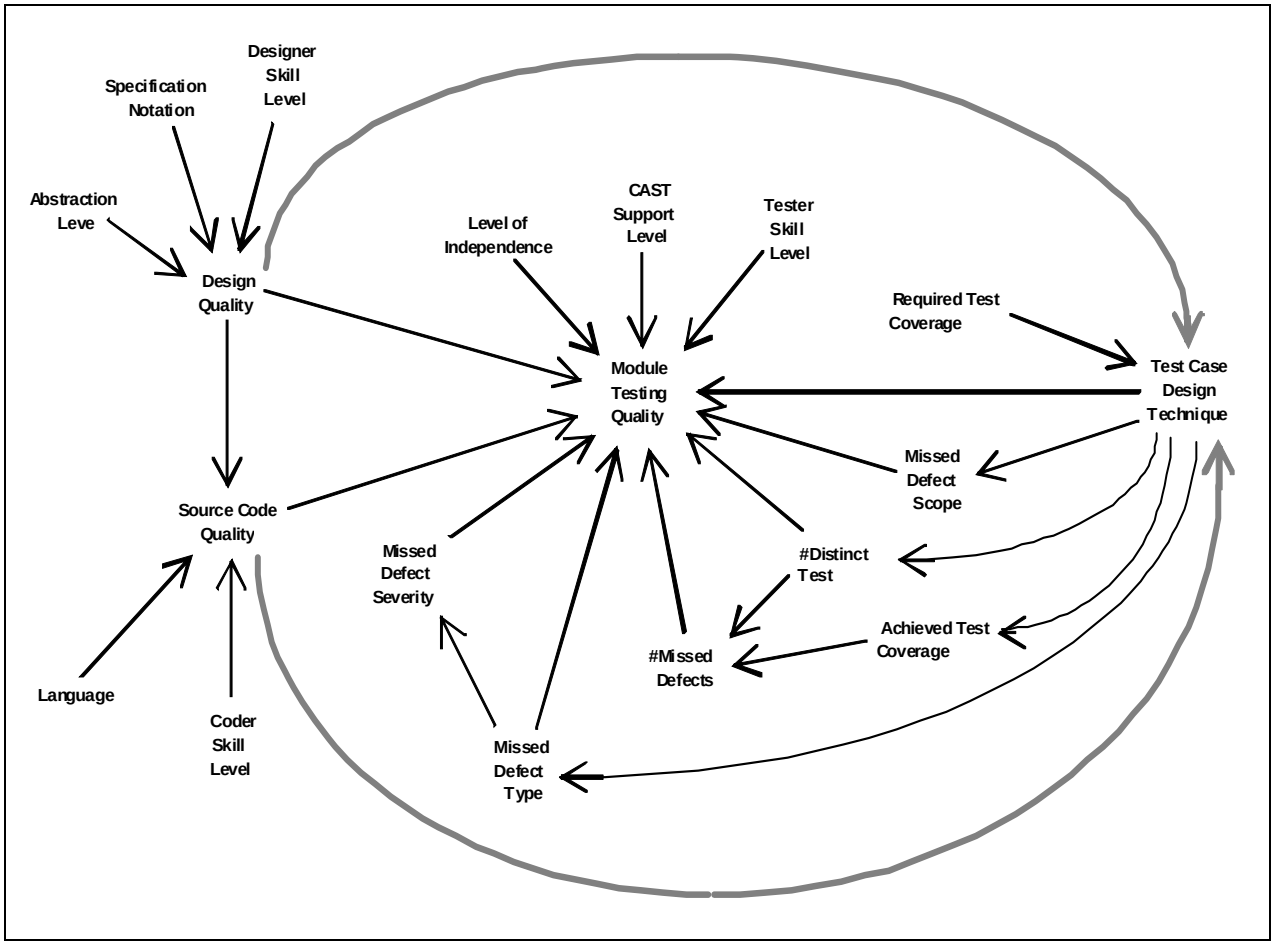
Table 2-14: Advantages of Exploratory Testing reported at Mercury, Neptune and Vulcan (Itkonen & Rautiainen 2005) (✓ indicates that all participants agree with the statement).

Advantages of Exploratory Testing	**Mercury** 1 participant	**Neptune** 4 participants	**Vulcan** 2 participants
Software can be used in many ways and there are many combinations between features, thus writing detailed test cases for everything is difficult, laborious and even impossible; thus, ET is a "natural choice"	✓	✓	✓
ET is well suited to testing from a user's perspective	✓	✓	✓
ET emphasises utilisation of tester knowledge, experience and creativity to find defects	✓	✓	✓
ET enables quick feedback on features from testers to developers	✓	✓	✓
ET adapts well in situations in which requirements and software frequently change and in which specifications are often ambiguous or incomplete	✓	✓	✓
ET enables learning about a system and the knowledge that is gained can be utilised during future work, including training and customer support.		✓	✓
When user manuals are used to guide ET, it also enables them to be evaluated for effectiveness and correctness		✓	
ET enables testing of the features of a system as a whole, allowing issues to be detected that would otherwise go undetected during scripted testing			✓
ET provides more versatile testing that delves deeper into tested features	✓	✓	✓
Each time the system is tested it is done so in a different way and enables exploration for new defects		✓	✓
ET enables testing aspects of the software that would not be included in test plans or test cases	Five out of seven interviewees (paper did not mention which companies)		
ET was high in efficiency and effectiveness	✓	✓	✓
ET enables defects to be detected in a short period of time			✓
ET enables more defects to be detected during system testing, possibly due to testers deriving more destructive input			✓
ET allows testers to quickly obtain an overall picture of system quality		✓	✓
Disadvantages of Exploratory Testing	**Mercury**	**Neptune**	**Vulcan**
It is difficult to determine the efficiency of ET in the long term, as test coverage is difficult to estimate, which possibly leaves many features untested		1 person	
A lack of test documentation makes it difficult to determine test coverage and what should be tested next	✓		
ET is less efficient and effective when performed by less experienced testers who have less domain knowledge			✓
Relying on the expertise of testers makes ET more prone to human error		✓	✓
Using ET to test complex systems is very time consuming		1 person	
It is impossible to find testers with enough experience to act as professional users		1 person	
As individual testers have different backgrounds an experience, they all perform ET from different viewpoints; however, this was also seen as an advantage in the versatility of the testing		✓	✓
Defects are not easily reproducible when using ET (however, this was not a problem at Mercury as testers kept detailed logs during ET)		✓	

2.6 External Influences on Test Set Quality

Reid (1994) identified a number of factors that can affect test set quality and, ultimately, the chance that a test case will detect a defect (Figure 2-16). This included specification notation, tester skill level, the level of independence between testers and developers, the quality of program source code and the required level of test coverage and the impact of this on the test methods that are chosen for testing a program. The elements of Reid's diagram can be used for considering external factors that could affect test set effectiveness during experimentation (e.g. see Chapters 5 and 6).

Figure 2-16: Factors affecting the quality of testing (Reid 1994).



Something that Reid did not consider was whether a tester's domain knowledge of the system under test can have an impact on their test effectiveness. This factor, along with specification notation (i.e. specification language) are discussed in more detail in the following two sections.

2.6.1 Effect of Specification Language on Black-Box Testing

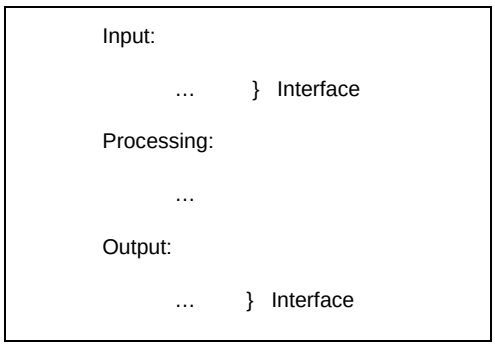
As Parrington and Roper (1989) state, "the purpose of a specification is to provide a clear, precise and unequivocal statement of the function to be implemented." This is not often achieved in practice, since specifications are usually written in natural language and consist of ambiguous vocabulary and undefined terms (Parrington & Roper 1989). Specifications are considered to be the greatest source of error in

software development (Patton 2006). For example, a survey of software testing practices in Australian found that out of 65 organisations, 34% and 25% respectively reported that 40 to 59% and 20 to 39% of program faults were caused by specification defects (Ng et al. 2004). Although specification languages are not the main topic of this thesis, they can affect a tester's ability to apply prescriptive black-box testing methods effectively. Thus, a number of issues with specification languages are discussed.

Fuchs and Schwitter (1996) argue that specifications are usually written in natural language because they need to be readable by all stakeholders, while Abbott (1986) observed that specifications that define the characteristics of input and output fields can be difficult for end-users to understand. Nonetheless, specifications that do not define the nature of valid and invalid program inputs (Jorgensen 1995) (e.g. boundary values) can result in inadequate testing (Marick 1995, Abbott 1986). The professional experience of the author of this thesis[17] is that that the majority of specifications produced in industry require further clarification and refinement before specification-based test case design is possible. For example, the syntax (e.g. boundaries, valid datatypes, valid data sets) of input and output fields are rarely defined, while the expected behaviour of the program for invalid inputs is seldom specified. Although this information can theoretically be obtained from software developers or by analysing program source code, this does not support independent black-box testing and it can still result in inadequate testing if developers do not have a complete or correct understanding of user requirements. Abbott (1986) suggested that two specifications should be produced, one for testers and one for end users. Parrington and Roper (1989) recommend that all user requirements should be rewritten to remove ambiguity and ensure test cases can be selected from them.

Consequently, Parrington and Roper (1989) proposed a specification structure that defines the inputs, outputs and functions of each component under test (Figure 2-17). Although this ensures that each input and output is defined, it does not include a language for defining the syntax of the inputs and outputs. The Test Specification Language (TSL) proposed by Ostrand and Balcer (1988) (see Section 2.2.7.6) supports the systematic documentation of equivalence classes for each input field, but did not provide a means for producing detailed syntax definitions for each input and output field. The use of formal specification languages could enhance Parrington and Roper's specification structure and Ostrand and Balcer's TSL.

Figure 2-17: Specification structure proposed by Parrington and Roper (Parrington & Roper 1989).



[17] This observation is based on three years of industry-based programming experience and almost four years of experience of working as a senior test consultant with a software testing consultancy in Australia.

Formal languages allow specifications to be expressed in "unambiguous language" that enables specification defects and ambiguities to be more easily detected than in specifications expressed in natural language (Liskov & Zilles 1975). On the other hand, formal specifications can be difficult to read without training (Fuchs & Schwitter 1996) and few industrial organisations use them (Ostrand 2002). One formal language that is "readily learned, easily understood, and widely accepted" (Lee & Dorocak 1973) is Backus-Naur Form (BNF) (Backus 1958, Naur 1960, Knuth 1964). BNF is based on context-free grammars, which consist of production rules that define the terminal and non-terminal elements of program input and output fields precisely (Paakki 1995)[18]. BNF has proven to be an effective for specifying the syntax of input fields for the purposes of applying EP, BVA, specification-based mutation testing (Murnane & Reed 2001) and ST (Beizer 1995). It enables precise definition of the minimum and maximum boundary values of range-based fields and individual data elements of list-based fields (see Figure 2-18). It facilitates the construction of Abstract Syntax Trees (see Section 2.2.3), which can be used to illustrate the relationships between each input field and which facilitate automatic test case generation (e.g. see (Kaksonen, Laakso & Takanen 2008)). Thus, specifying input and output fields in BNF can enable more effective black-box testing, regardless of which particular test method is applied.

Figure 2-18: Example of a BNF specification for the street name of an address.

```
    <street>  ::= <name>  ∧ <type>  {∧ <direction>}
      <name>  ::= [A – Z | a – z | - ]1 - 40
      <type>  ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt | …]
 <direction>  ::= [North | South | East | West ]
          ∧   ::= [ ]  #i.e. one space
```

2.6.2 Effects of Domain Knowledge on Black-Box Testing

Interestingly, most publications of non-prescriptive testing approaches like EG and ET (see Section 2.2.5) suggest that there is no system to the seemingly 'intuitive' process that takes place during test case design and execution. For example, Jorgensen (1995) refers to EG as the "most intuitive" black-box method. Kaner (1988) maintained that "in complex situations, your intuition will often point you toward a tactic that was successful (you found bugs with it) under similar circumstances. Sometimes you won't be aware of this comparison. You might not even consciously remember the previous situations. This is the stuff of expertise." Agruss and Johnson (2000) argued that "much of what experienced software testers do is highly intuitive, rather than strictly logical."

The Oxford English Dictionary (1970) defines intuition as "the immediate apprehension of an object by the mind without the intervention of any reasoning process" and "immediate apprehension by the intellect alone." Regardless of whether the tester is consciously aware of the process they follow when using non-prescriptive testing approaches, there may still be a pattern to the types of test case design rules they use. As

[18] Wikipedia (Context Free 2008) provides a general definition of context-free grammars.

Kaner et al. (2001) observe, a tester's skill with ET can increase as they become more familiar with a system, including the market it was developed for, the risks associated with developing it and the failures previously detected in it. As Barber (2007) claimed, "the more we know about what a system or application is supposed to do, the more intuitive we believe it is." These views suggest that the intuition a tester uses during ET could be logical and procedural domain knowledge they have gained over time, which could be based (among other things) on their knowledge of effective test case design rules from prescriptive black-box testing methods (Craig & Jaskiel 2002). There may also be information about the domain of a system that gives experienced "pathological" testers clues on how to test it effectively. It is also possible that individual testers have their own unique collection of test case design rules that they routinely use when using non-prescriptive approaches to testing.

In fact, many prescriptive black-box testing methods are based on domain knowledge of program faults and failures. As Wild et al. argue (1992), program faults are often the result of programmers misunderstanding the problem domain of a program. Many prescriptive methods are based on specific types of program faults. For example, BVA is based on the view that programmers often make 'off-by-one' errors, which can be identified through "application solution" domain knowledge (Reed 1990). It is also realistic to assume that there are other high-yield black-box test case design rules that are commonly used by experienced testers that have not yet been published in software testing literature. If those rules could be explicitly defined, this knowledge could be used to enhance black-box testing methods and to improve the defect detection skills of both novice and experienced testers (see Chapter 3 for GQASV and SMT, which can support this). This knowledge could also be used to classify black-box methods on the extent to which they rely on domain knowledge.

2.7 Automation of Black-Box Testing Methods

Since testing can involve the design of thousands of test cases and specification changes can cause changes to many of those tests, automation can be necessary (Bauer & Finger 1979). In 1983, Perry argued that "testing has been primarily a manual operation and often an inefficient function in many organisations since it can suffer from human error and is often very time consuming." (Perry 1983). Today there are many automated tools available to support black-box testing and to analyse and improve test coverage and effectiveness. For example, model based testing tools provide support at the system testing level, while the 'xUnit' family of tools support automated unit testing (Bertolino 2007). Saley, Hoffman and Strooper (2002) developed a white-box tool that automatically generates boundary values for testing Java classes. JCover can support analysis of code coverage achieved when testing Java programs (Codework 2009).

The tools that are of most interest in this thesis are those that can be used for the generation of black-box test data values and test cases. This includes tools for Random Testing (Section 2.7.1), Syntax Testing (Section 2.7.2), black-box testing (Section 2.7.3), Classification Trees (Section 2.7.4) and a new prototype testing tool called the Atomic Rules Testing Tool (ARTT), which automates Atomic Rule definitions of EP, BVA and ST (see Chapter 3 and 4).

2.7.1 Automated Random Testing

In terms of automation, Random Testing is probably the most well supported black-box testing method. Pseudo-random number generators are provided with most standard programming languages and are available in simple-to-use functions like *randbetween* in Microsoft Excel. Combinatorial test methods are also well supported, as numerous algorithms are available (see Section 2.2.7).

2.7.2 Automated Syntax Testing

Automated Syntax Testing dates back to as early as the 1960's and 1970's. In 1962, Sauder implemented a tool that parsed data declarations in COBOL programs to determine the nature of valid inputs in order to generate test data (Sauder 1962). In 1977, Houssais conducted an experiment to measure the detection of faults in an Algol 68 compiler using a test data generation tool that automatically produced syntactically and semantically correct test programs (Houssais 1977). Although Houssais did not provide an example of the language in which the specifications were written, it did state that the grammar was a subset of affix grammars for Algol 68. Examples of the types of test cases that were generated were testing loops with 1, 2, 3 and 'many' iterations. While Houssais maintained that the method could be used to generate invalid test programs, this was not included in his research or experimentation, nor was an explanation of how this could be achieved. Thus, the main shortcomings of this paper were that it did not explain how the test programs were derived and did not describe the test generation algorithm utilised.

In 1982, Fultyn described an automated approach for constructing valid inputs from BNF specifications (Fultyn 1982), which could be considered to be a form of ST. The approach involved random selection of terminals and randomly made decisions about whether to include optional constructs in test cases. One advantage was that program features that would not usually be executed by users were tested. On the other hand, neither the testing strategy nor the automated tool was verified through experimentation or analysis.

A more recent tool that automates Syntax Testing is JSynTest (JSynTest). This tool takes a specification expressed in a variant of BNF as input, generates an Abstract Syntax Tree (referred to as an "AND-OR graph") and outputs a Java program that can either be used to generate test cases or become part of the input validation code of the system under test. Beizer (1995) observed that automation of Syntax Testing is readily achieved through LEX and YACC and commercial tools like T, which has also been used to automate BVA (Jorgensen 2002). Bouquet, Dadeau and Legeard (2006) developed a grey-box testing tool that automatically selects boundary values from "built in" datatypes such as integers and characters from Java classes expressed in the Java Modelling Language (JML).

2.7.3 Automated Black-Box Testing

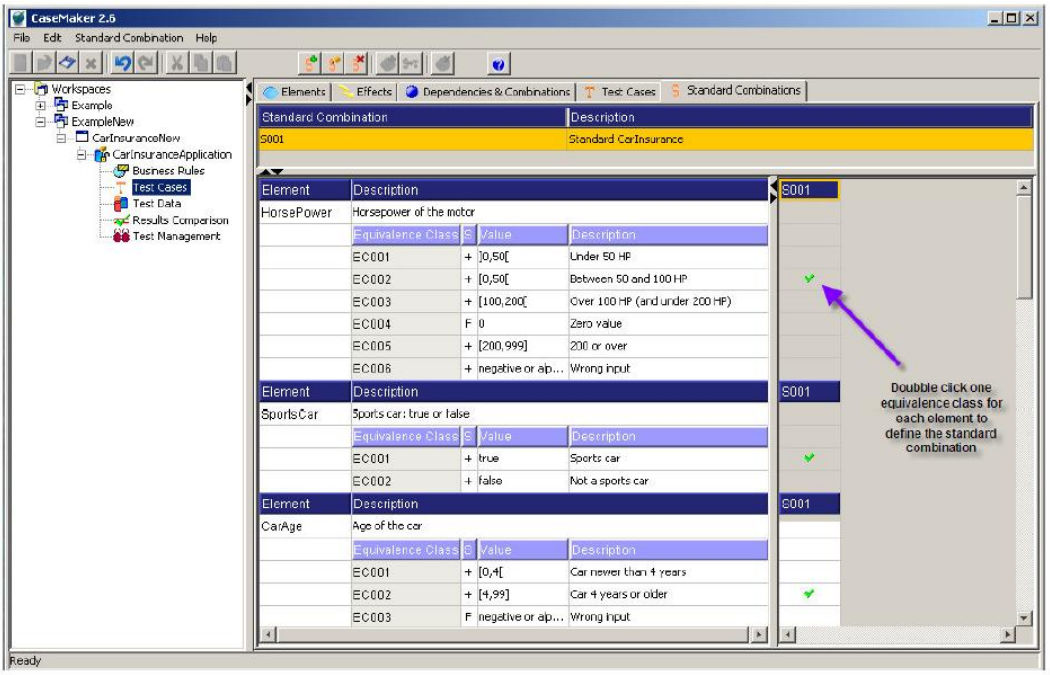
CaseMaker is a commercial testing tool that can be used to automatically generate test data values and test cases for EP, BVA, EG, Decision Tables and Pair-wise testing (Figure 2-19) (Díaz & Hilterscheid). CaseMaker can generate test cases from input data specifications produced in Microsoft Word or as UML statecharts and business rules expressed in a formal notation. Equivalence classes are automatically generated for range-based fields by identifying partitions below the lower boundary (invalid partition),

above the upper boundary (invalid partition), between the lower and upper boundaries (valid partition) and by selecting 'any other value' (invalid partition *defined by exclusion*). For lists, partitions are generated by selecting the valid items in the input list (valid partition) and 'any other value' (invalid partition *defined by exclusion*). Test data values are automatically generated from each equivalence class, which can include the nominal value, boundary values, random values and all values. Expected results called "effects" can be added to each test data value and dependencies between values can be specified (Case Maker Part 3 2007). Test cases are then automatically generated using a Pair-wise algorithm. Test cases can be output in comma-separated format (csv), HTML, XLS and Microsoft Word. Comma-separated files can be input into Microfocus' (formally Compuware's) automation tool called Test Partner and can then be used to design automated test scripts.

Interestingly, CaseMaker does not generate test data values that lie just outside and just inside the boundaries of range-based fields. Although ST is not directly supported by the tool, 'functions' can be applied to the test data values that are selected by the tool, to manipulate (i.e. mutate) them in a similar fashion to ST (Case Maker Part 4 2007). Manipulation functions include date and time conversion functions, mathematical (e.g. absolute value, maximum and maximum values, round, square root), text conversion (e.g. convert from lower case to upper case and visa versa, concatenate), trigonometric (e.g. sine, cosine, tangent) and constants (e.g. speed of light, gravity). CaseMaker also enables allocation of prefixes and suffixes to test data values (e.g. % and $).

A case study that assessed CaseMaker against Comverse's Mobile Internet Solutions system revealed that the tool improved test coverage and reduced testing by two person weeks (from a total duration of two person months) and in one component, it reduced the duration of manual test data derivation by three to four times (Tsubery 2007).

Figure 2-19: Equivalence class generation in CaseMaker (Díaz & Hilterscheid).



2.7.4 Classification Trees

An automated tool called the Classification-Tree Editor (CTE) has been developed for supporting the construction of Classification Trees (Lehmann & Wegener 2000). While this tool supports the documentation of Classification Trees and the generation of test cases, it does not automate the identification of equivalence classes, boundary values or syntax testing values for input fields.

2.8 Summary

In this chapter, the features, advantages, disadvantages, similarities and differences of a wide variety of black-box testing methods were presented. This literature survey included a detailed examination of at least seven problems that affect the usability and failure-detection effectiveness of existing black-box testing methods: definition by exclusion, multiple versions, method overlap, notational and terminological differences, reliance on domain knowledge, difficult to audit and difficult to automate.

Common terminology used in black-box test case design was introduced first (Section 2.1.1). This was followed by a discussion of the four key steps of black-box test case design (Section 2.2). Individual black-box testing methods were presented and the seven problems with black-box testing methods mentioned above were discussed (Sections 2.2.1 to 2.2.9). A summary of how each of the black-box testing methods explored touches on the four steps of black-box test case design was then presented (Section 2.3). Approaches to combining (Section 2.4) and selecting test methods (Section 2.5) were then presented. This included a review of experimental comparisons of black-box testing methods (Section 2.5.5). External influences on test case quality were explored (Section 2.6), including specification language and domain knowledge. Approaches to black-box test method automation were also reviewed (Section 2.7)

The seven problems with existing descriptions of black-box testing methods that were explored in this chapter indicate a need for a prescriptive, uniform notation for representing these methods that resolves these problems and that ultimately improves the usability and failure-detection effectiveness of these methods. Improved approaches to specifying program input and output fields would also enable more effective testing, as would approaches for guiding testers in the creation of customised black-box testing methods.

In the next chapter, three new approaches to supporting more effective black-box testing are introduced, in an attempt to resolve these issues. They are the Atomic Rules approach, Systematic Method Tailoring and Goal/Question/Answer/Specify/Verify.

Chapter 3

A Generalised Representation for Black-Box Testing Methods

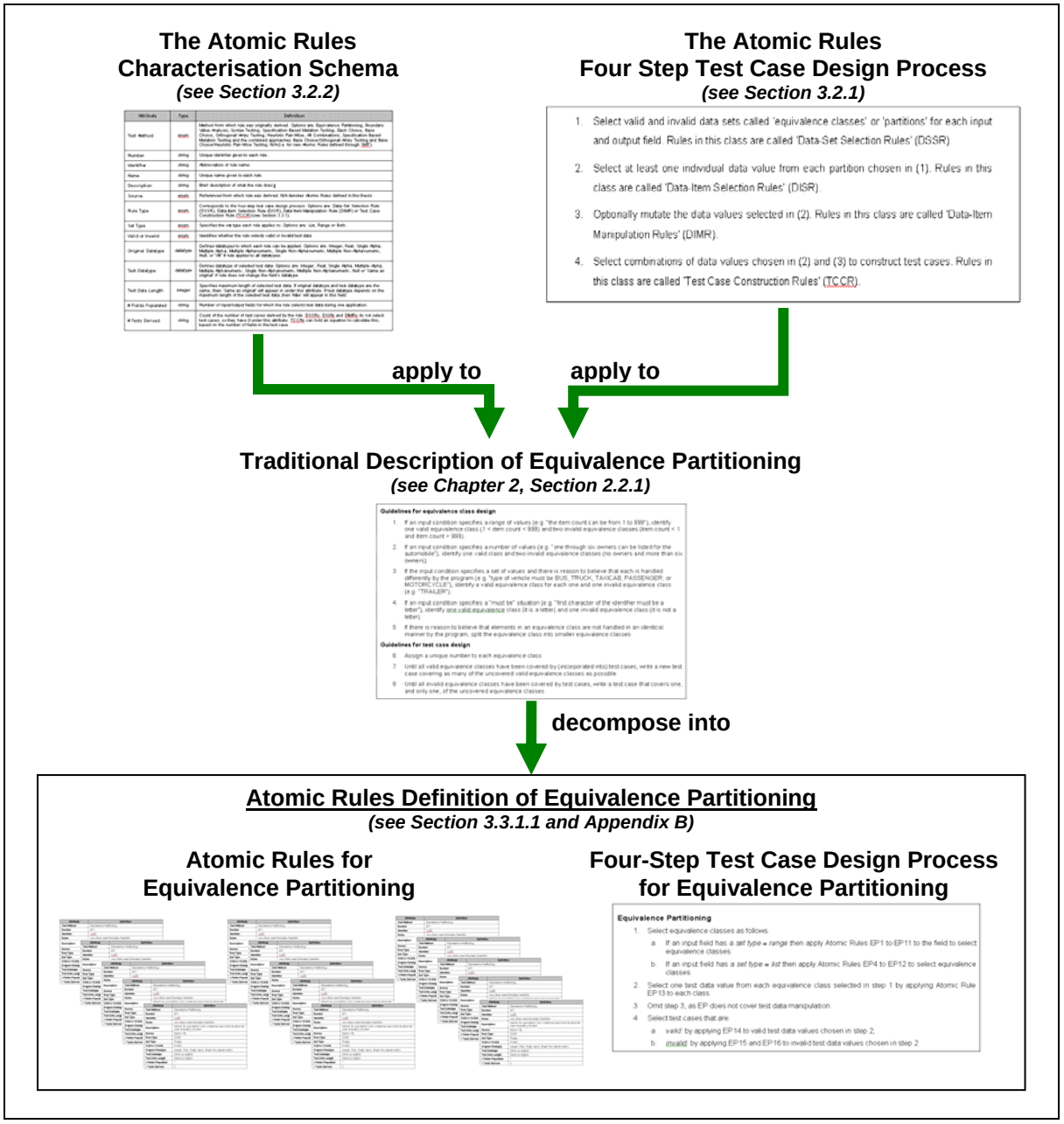
"As a matter of cosmic history, it has always been easier to destroy than to create."
Mr. Spock, Star Trek II: The Wrath of Khan, 1982

3.1 Overview

As stated in the introduction, the goal of this thesis is to investigate and resolve seven problems with existing descriptions of black-box testing methods that affect the usability and failure-detection effectiveness of the methods; these were: *definition by exclusion*, *multiple versions*, *method overlap*, *notational and terminological differences*, *reliance on domain knowledge*, *difficult to audit* and *difficult to automate*. In this chapter, a new representation for describing black-box testing methods called the Atomic Rules approach is introduced, in an attempt to resolve these problems. Two supporting approaches are also introduced, Systematic Method Tailoring and Goal/Question/Answer/Specify/Verify.

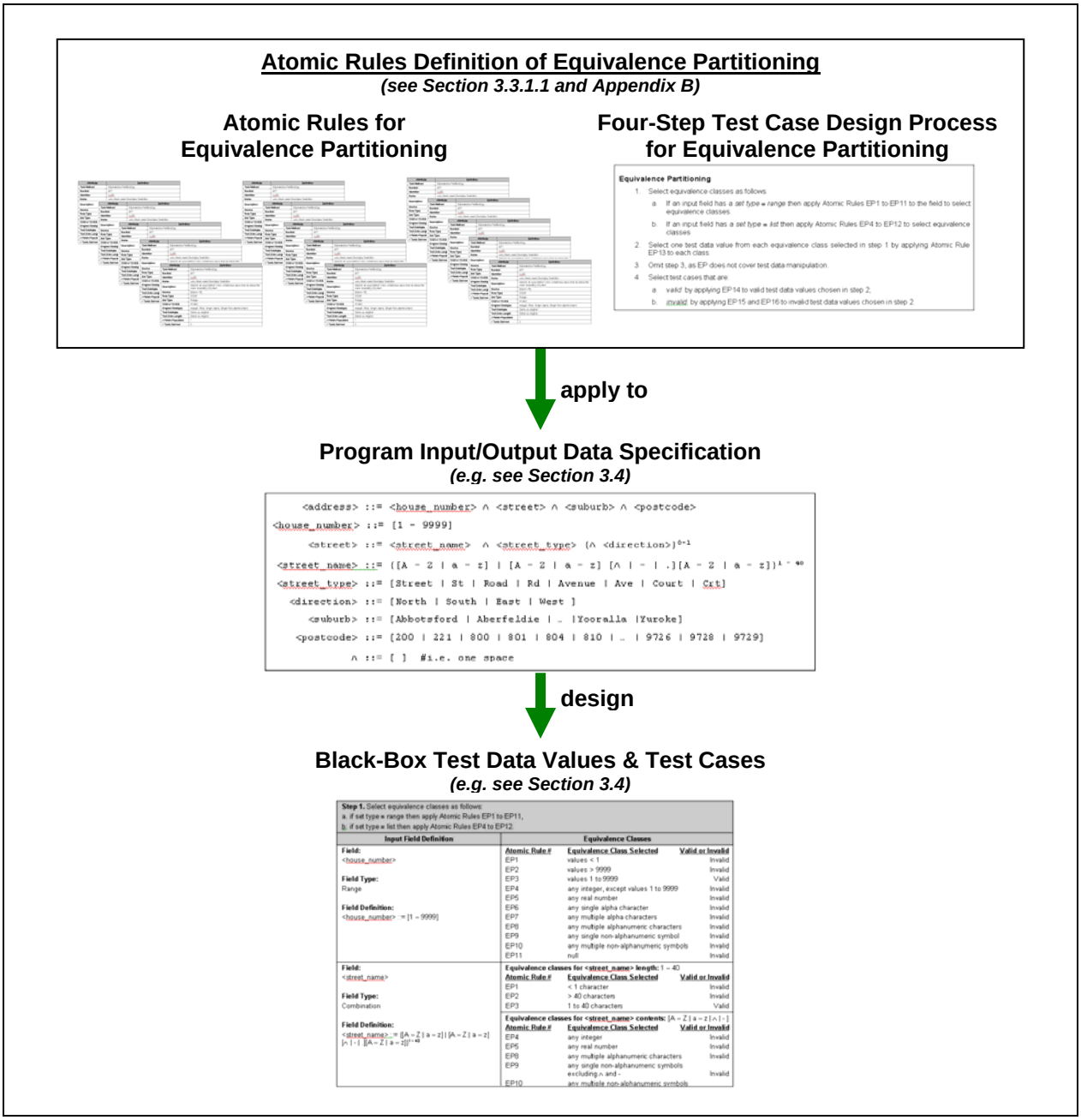
In the Atomic Rules approach, black-box testing methods are decomposed into individual test case design rules called 'Atomic Rules.' Each Atomic Rule is defined in an instance of a characterisation schema called the 'Atomic Rules schema,' which defines the characteristics of each individual test case design rule in a uniform notation. The Atomic Rules are then utilised within the four-step black-box test case design process (which was introduced in Chapter 1), allowing each black-box testing method to be defined in a uniform notation. For example, the illustration below demonstrates the decomposition of Myers' original definition of Equivalence Partitioning into Atomic Rules and the redefinition of this test method in the four-step test case design process (Figure 3-1).

Figure 3-1: Example of developing an Atomic Rules definition of Myers' (Myers 1979) definition of Equivalence Partitioning.



Once a black-box testing method has been described in the Atomic Rules approach, it can be applied to the specification of a program's inputs and outputs to generate black-box test cases in a far more repeatable and predictable fashion that would be possible using its original formulation (Figure 3-2). Eleven black-box testing methods have been represented in the Atomic Rules approach, including Equivalence Partitioning, Boundary Value Analysis, Syntax Testing and combinatorial methods Each Choice, Base Choice, Orthogonal Array Testing, Heuristic Pair-Wise, All Combinations, Specification-Based Mutation Testing and the combined approaches Base Choice/Orthogonal Array Testing and Base Choice/Heuristic Pair-Wise Testing (see Appendix B).

Figure 3-2: Illustration of applying the Atomic Rules definition of Equivalence Partitioning to a specification to design black-box test data and test cases.



In summary, the Atomic Rules approach resolves the seven problems with traditional black-box testing methods in the follow ways.

The uniformity of the Atomic Rules characterisation schema and four-step test case design process resolves ***notational and terminological differences*** between methods by providing them with a common vocabulary and process model. ***Method overlap*** is resolved during method decomposition by identifying Atomic Rules that overlap both within and between methods. ***Multiple versions*** of the same method are resolved by creating one set of Atomic Rules that covers the test case design rules of all published versions of that method. This also makes the methods ***easier to audit***, since checks for test set completeness can be

based on the one prescriptive definition of each black-box testing method. ***Definition by exclusion*** is resolved by defining a series of Atomic Rules for Equivalence Partitioning that each select equivalence classes for one explicit datatype (e.g. integer, real, alpha, non-alphanumeric), thereby expressing the universe of discourse for program inputs. This also reduces ***reliance on domain knowledge***, as it provides testers what one definition of the universe of discourse with respect to program inputs and ensures that each Atomic Rule is defined to a level of detail that facilitates the design of effective and predictable test cases, regardless of a tester's own unique domain knowledge and experience. The prescriptive notation and process used in the Atomic Rules approach also the methods ***easier to automate***. In Chapter 4, a prototype testing tool called the Atomic Rules Testing Tool is presented. ARTT automates the application of Atomic Rules from EP, BVA and ST to specifications input by the user and can also be used to record domain knowledge captured during GQASV and to define new Atomic Rules through SMT.

Reliance on domain knowledge is also reduced via two new approaches called SMT and GQASV. *SMT* enables the creation of new Atomic Rules that are based on each tester's own unique domain knowledge, allowing that knowledge to be specified, shared and reused. GQASV supports the definition of precise input data specifications to support more effective black-box testing (when such specifications are not readily available) and also supports the capture of domain knowledge that is utilised during the specification process, allowing the knowledge gained during the specification process to be shared and reused.

An additional benefit is that individual Atomic Rules from EP, BVA and ST can be used to aid test data selection for methods like State Transition Testing, Use Case Testing and the Category Partition Method.

The current chapter begins by describing the Atomic Rules' four-step test case design process (Section 3.2.1) and characterisation schema (Section 3.2.2). Decomposition of black-box testing methods into Atomic Rules is demonstrated for EP, BVA, ST and one combinatorial testing method (Section 3.3.1.4). This is followed by worked examples that demonstrate the application of the Atomic Rules definition of EP, BVA and ST to an example specification for an Address Parser (Section 3.4). Additional benefits of the Atomic Rules approach are then discussed, including how it simplifies the checking the completeness of black-box test sets (Section 3.5) and how it can be used to support State Transition Testing (Section 3.6.1), Use Case Testing (Section 3.6.2) and the Category Partition Method (Section 3.6.3). Then, improvements (Section 3.7) and limitations (Section 3.8) of the Atomic Rules approach are presented. This is followed by research related to the development of characterisation schemas and method decomposition (Section 3.9). Goal/Question/Answer/Specify/Verify and Systematic Method Tailoring are then introduced (Section 3.10) and demonstrated against a real-world online foreign currency calculator (Section 3.10.3).

3.2 The Atomic Rules Approach

The Atomic Rules approach was developed by analysing the common elements of eleven different black-box testing methods, including Equivalence Partitioning (EP), Boundary Value Analysis (BVA), Syntax Testing (ST) and combinatorial methods Each Choice, Base Choice, Orthogonal Array Testing, Heuristic Pair-Wise, All Combinations, Specification-Based Mutation Testing (SBMT) and the combined approaches Base Choice/Orthogonal Array Testing and Base Choice/Heuristic Pair-Wise Testing, which

were described differently in fifteen different sources (Beizer 1984, Beizer 1990, BS 7925-2, Craig & Jaskiel 2002, Graham 1994, Grindal et al. 2004, Hetzel 1988, Jorgsensen 1995, Kaner 1988, Lewis 2000, Mandl 1985, Marick 1995, Mosley 1993, Murnane & Reed 2001, Myers 1979). This revealed a common set of attributes that could be used to characterise the test case design rules within the methods. The attributes were used to build the Atomic Rules schema (Section 3.2.2). This also evolved into a four-step test case design process for black-box testing (Section 3.2.1).

A significant number of other 'duplicate' publications of EP, BVA, ST and combinatorial testing methods were also cited during this investigation, including (Abbott 1986, Beizer 1995, Burnetein 2003, Copeland 2004, Grindal et al. 2005, Hetzel 1988, Hutcheson 2003, Jorgensen 2002, Kaner et al. 2001, Kit 1995, Mosley & Posey 2002, Myers 2004, Ould & Urwin 1986, Page et al. 2009, Parrington & Roper 1989, Patton 2006, Perry 2000, Pressman 1992, Rae et al. 1995, Sommerville 1994, Tamres 2002, Watkins 2001).

3.2.1 The Four-Step Black-Box Test Case Design Process

As introduced in Chapter 2, each black-box testing method typically focuses on just one of the four key steps of black-box test case design, as follows.

1. partitioning of the input domain of a program;
2. selection of individual test data values from each partition;
3. manipulation or "mutation" of the test data values; and
4. combining test data values to construct test cases.

Although some black-box testing methods cover more than one of these steps, a published definition of a method usually focuses on just one of these steps. For example, EP provides guidance on partitioning the input domain, selecting test data and designing of test cases. Although it covers three of the four steps of black-box test case design, it specialises in providing guidance for input and output domain partitioning. Also, several methods share common test case design steps. For example, boundary values are typically selected from the edges of equivalence classes that are typically identified through EP (e.g. see (Myers 1979)). While ST is not described as a partitioning approach and does not include explicit guidelines for partition selection, partitions must implicitly be selected for each input field to enable selection of syntax-based test data values. These insights led to the definition of a fundamental four-step test case design process for black-box testing that was defined in Chapter 1 (Figure 1-4) (Murnane et al. 2005, 2007). An analysis of the eleven different black-box testing methods studied in this research revealed that each could be decomposed into a set of Atomic Rules, where each Atomic Rule has a 'rule type' that corresponds to one of these four steps (see Chapter 1, Figure 1-4). When used in conjunction with a set of Atomic Rules from a particular method, this four-step process can be used to construct black-box test cases in the usual way (Section 3.4).

3.2.2 The Atomic Rules Schema

The Atomic Rules schema (Table 3-1) comprises fourteen attributes that were identified by analysing the common features of test case design rules from the eleven different black-box testing methods under examination (see Section 3.2). The attributes of this schema are as follows.

1. **Test Method.** The name of the test method the test case design rule was derived from (e.g. Equivalence Partitioning, Boundary Value Analysis, Syntax Testing).
2. **Number.** A unique identifier given to each rule, which starts with an abbreviation of the name of the test method and ends with an incremental number (e.g. *EP1*, *EP2*, *EP3*).
3. **Identifier.** An abbreviation of the Name field.
4. **Name.** The name of the rule.
5. **Description:** Describes the 'functionality' of the rule (i.e. test data or test cases it derives).
6. **Source.** The reference from which the rule was derived. For example, many EP rules were derived from (Myers 1979), while many ST rules were derived from (Beizer 1995) and (Marick 1995). Atomic Rules that were defined as a part of this research have "N/A" in this field.
7. **Rule Type.** There are four types of Atomic Rules, corresponding to the four-steps test case design process. Data-Set Selection Rules (DSSRs) partition the input and output domains of a program into equivalence classes, Data-Item Selection Rules (DISRs) select test data values from each partition, Data-Item Manipulation Rules (DIMRs) "mutate" test data values and Test Case Construction Rules (TCCRs) combine test data values into test cases.
8. **Set Type.** This corresponds to the set type of the input field the rule can be applied to. For example, some Atomic Rules can only be applied to contiguous data ranges (e.g. DISRs from BVA), while others can only be applied to list-based fields (e.g. DSSRs from EP that select values lists). Thus, the values List and Range can appear in this field.
9. **Valid or Invalid:** Describes whether the rule selects valid test data that a program should accept as correct or invalid test data that it should reject as incorrect.
10. **Original Datatype.** This records the datatypes of input and output fields that the rule can be applied to. For example, one ST rule substitutes a lowercase letter for an uppercase letter and this can only be applied to fields of datatype 'alpha.'
11. **Test Datatype**. This records the datatype of test data selected by the rule. For example, some EP rules specifically select invalid datatypes as test data.
12. **Test Data Length.** This contains the length (in characters) of test data selected by the rule. For example, some ST rules select one character as test data, while others select strings.
13. **# Fields Populated.** This records the number of fields that are populated with test data when the rule is applied. For example, BVA rules select test data for one field at a time, whereas Test Case Construction Rules can populate all fields of a test case at once.

14. **# Tests Derived.** For Test Case Construction Rules, this field contains an equation of the number of test cases derived.

Each of these characteristics is defined as an attribute in the Atomic Rules schema (Table 3-1).

Table 3-1: The Atomic Rules characterisation schema.

Attribute	Type	Definition
Test Method	enum	Black-box testing method that the Atomic Rule was originally derived from. Options are: Equivalence Partitioning, Boundary Value Analysis, Syntax Testing, Specification-Based Mutation Testing, Each Choice, Base Choice, Orthogonal Array Testing, Heuristic Pair-Wise, All Combinations, Specification-Based Mutation Testing and the combined approaches Base Choice/Orthogonal Array Testing and Base Choice/Heuristic Pair-Wise Testing, N/A (for new Atomic Rules that are defined through SMT).
Number	String	Unique identifier given to each rule.
Identifier	String	Abbreviation of rule name.
Name	String	Unique name given to each rule.
Description	String	Brief description of what the rule does.
Source	enum	References from which rule was derived. N/A denotes Atomic Rules defined in this thesis.
Rule Type	enum	Corresponds to the four-step test case design process. Options are: Data-Set Selection Rule (DSSR), Data-Item Selection Rule (DISR), Data Item Manipulation Rule (DIMR) or Test Case Construction Rule (TCCR) (see Section 3.2.1).
Set Type	enum	Specifies the set type each rule applies to. Options are: List, Range or Both.
Valid or Invalid	enum	Identifies whether the rule selects valid or invalid test data.
Original Datatype	*datatype*	Defines datatypes to which each rule can be applied. Options are: Integer, Real, Single Alpha, Multiple Alpha, Multiple Alphanumeric, Single Non-Alphanumeric, Multiple Non-Alphanumeric, Null, or “All” if rule applies to all datatypes.
Test Datatype	*datatype*	Defines datatype of selected test data. Options are: Integer, Real, Single Alpha, Multiple Alpha, Multiple Alphanumeric, Single Non-Alphanumeric, Multiple Non-Alphanumeric, Null or ‘Same as original’ if rule does not change the field’s datatype.
Test Data Length	integer	Specifies the maximum length of test data selected by the rule. If *original datatype* and *test datatype* are the same, then ‘Same as original’ will appear. If *test datatype* depends on the maximum length of selected data, then ‘Max’ will appear in this field.
# Fields Populated	string	Number of input fields for which the rule selects test data during one application.
# Tests Derived	string	Count of the number of test cases derived by the rule. DSSRs, DISRs and DIMRs do not select test cases, thus they have “0” under this attribute. TCCRs can hold an equation to calculate this, based on the number of fields in the test case.

The semantics of most of the attributes in the Atomic Rules schema (Table 3-1) are evident from their definition. However, there are three attributes that require further explanation:

- Set Type,
- Original Datatype and
- Test Datatype.

Set Type defines whether an Atomic Rule can be applied to an input data field that is specified as a 'list' or a 'range' (referred to as 'list-based' and 'range-based' fields). For example, some EP rules only apply to ranges of contiguous data, while others only apply to data stored in lists (Myers 1979). *Lists* can be expressed as $L ::= [v_1 \mid v_2 \mid \ldots \mid v_n]$ or $L ::= [v_1, v_2, \ldots, v_n]$ where n is the number of v values contained in list L. *Ranges* can be represented as $\{R : lb \leq R \leq ub\}$ or $R ::= [lb - ub]$, which denotes the range of values from lower boundary lb to upper boundary ub. These terms were adapted from similar concepts discussed in (Jorgensen 1995, Lewis 2000, Mosley 1993, Myers 1979, Page et al. 2009). For example, in their definition of EP, Page, Johnston and Rollison (2009) use the term 'Range' to describe a set of contiguous data and 'Group' to describe sets of related items that are each processed in the same way by the program.

Original Datatype and *Test Datatype* make use of eight 'base' datatypes that are required for defining black-box test case design rules. A characterisation schema for classifying datatypes was defined to specify each one in a standard notation (Table 3-2). The schema was then populated for eight base datatypes: integer, real, single alpha, multiple alpha, multiple alphanumeric, single non-alphanumeric, multiple non-alphanumeric and null (Table 3-3) ('single alphanumeric' was not defined as a datatype, since it is not possible to define a single character string with two datatypes represented). Defining eight Atomic Rules for EP that each correspond to one of these datatypes resolves *definition by exclusion* (see Section 3.3.1.1).

Table 3-2: Characterisation schema for defining the datatype of program input and output fields[1].

Attribute	Type	Definition
Name	string	A unique name for each datatype.
Set Type	enum	Describes the set type of the datatype. Options are List and Range.
Size	string	Max length of datatype in bytes. Length can depend on implementation using the datatype (Meek 1994), for which "Max buffer length" will appear.
Example	string	A simple example.

Table 3-3: Datatypes defined for use in the Atomic Rules schema (used by Atomic Rules schema fields *Original Datatype* and *Test Datatype*).

Name	Set Type	Size	Example
Integer	List or Range	Max buffer length	List: [-30, 4, 16, -1, 25] Range: [-16 – 335]
Real	List or Range	Max buffer length	List: [10.4, -100.5, 3.2] Range: [-12.1 – 54.23]
Single Alpha	List or Range	1 byte	List: [e, a, n, B, c, H, I] Range: [e – g]
Multiple Alpha	List	Max buffer length	List: [Melbourne, Sydney, Adelaide, Perth, Darwin, Hobart, Canberra, Brisbane]
Multiple Alphanumeric	List	Max buffer length	List: [4z3A, A83, b44]
Single Non-Alphanumeric	List or Range	1 byte	List: [", (, %, *, ", +, &] Range: ["– +]
Multiple Non-Alphanumeric	List	Max buffer length	List: [*&%, ()*&^&^$, {}:"<>?]
Null (empty)	List or Range	0 bytes	List: [ ] Range: [ ]

[1] The columns of Table 3-2 relate to the domains of Reed's KABASPP model (Reed 1990) as follows. *Name* (col. 1), *Set Type* (col. 2) and *Example* (col. 4) are from the application solution domain and *Size* (col. 3) is from the development and run-time domain.

3.3 Representing Black-Box Testing Methods as Atomic Rules

Fifty-two Atomic Rules were defined for the eleven black-box testing methods analysed in this thesis (see Appendix B). To demonstrate the decomposition of original descriptions of black-box testing methods into Atomic Rules, Atomic Rule definitions of EP, BVA, ST and All Combinations are provided below.

3.3.1.1 Decomposing Equivalence Partitioning

Using the above definitions, it is possible to decompose Myers' original definition of EP (Myers 1979) into a set of Atomic Rules, and to show that the test case design procedure for this method is readily concerted into the four-step test case design process (see Section 3.2.1) that has been developed. In addition, decomposition of test case design rules from two other definitions of EP (published in (Jeng & Weyuker 1989, BS 7925-2)) are included, allowing all method variations to be included (i.e. definitions from (Abbott 1988, BS 7925-2, Burnstein 2003, Copeland 2004, Craig & Jaskiel 2002, Jorgensen 1995, Kaner 1988, Kit 1995, Mosley 1993, Mosley & Posey 2002, Myers 1979, Page et al. 2009, Patton 2005, Parrington & Roper 1989, Pressman 1992, Tamres 2002) are included). This consolidation would be difficult to achieve without both the Atomic Rules and the four-step process having been defined.

Myers' (1979) test case design guidelines (Table 3-4, col. 1) are first decomposed into individual test case design rules (Table 3-4, col. 2) and then each rule is defined as an Atomic Rule (Table 3-4 col. 3 and Table 3-5). Consider Myers' (1979) first guideline, which is as follows:

> **"If an input condition specifies a range of values (e.g. "the item count can be from 1 to 999"), identify one valid equivalence class (1 ≤ item count ≤ 999) and two invalid equivalence classes (item count < 1 and item count > 999)."**

This guideline can be decomposed into the following test case design rules, which can then be documented as individual Atomic Rules.

1. Equivalence class 'item count < 1' is covered by a test case design rule that selects an equivalence class containing all values below the lower boundary of the field, which becomes Atomic Rule *EP1: Less Than Lower Boundary Selection.*
2. Equivalence class 'item count > 999' is covered by a test case design rule that selects an equivalence class containing all values above the upper boundary of the field, which becomes Atomic Rule *EP2: Greater Than Upper Boundary Selection.*
3. Equivalence class '1 ≤ item count ≤ 999' is covered by a test case design rule that selects an equivalence class containing all values between the lower and upper boundary of a field, which becomes Atomic Rule *EP3: Lower to Upper Boundary Selection.*

Thus, each test case design rule is documented as an Atomic Rule that selects test data for a range-based field (Table 3-5). This process allows *method overlap* to be resolved. For example, Myers' first and second guidelines both identify equivalence classes for range-based fields (Table 3-4 col. 1, guidelines 1 and 2).

Thus, they can be decomposed into three test case design rules (Table 3-4 col. 2, rules a to c) and three corresponding Atomic Rules (Table 3-4 col. 3, *EP1*, *EP2* and *EP3*). *EP1* to *EP3* can be applied to any field containing contiguous data, including alpha and non-alphanumeric, if the ASCII table is used to identify values that lie outside field boundaries. These three Atomic Rules can also be used to test field repetition. For example, consider the field *<vehicle> ::= [car | truck | motorbike | van]*[1-4], which specifies a list of vehicle types from which a user can choose one to four vehicles. These rules could be used to identify equivalence classes for testing when the user chooses less than one vehicle (i.e. zero vehicles, using *EP1*), greater than four vehicles (using *EP2*) and between one and four vehicles (using *EP3*).

Table 3-4: Decomposing Myers' definition of Equivalence Partitioning into Atomic Rules.

Myers' Definition of Equivalence Partitioning		**Corresponding Atomic Rules**[2]
Myers' (1979) Definition of EP	**Test Case Design Rules**	
1. "If an input condition specifies a range of values (e.g. "the item count can be from 1 to 999"), identify one valid equivalence class (1 ≤ item count ≤ 999) and two invalid equivalence classes (item count < 1 and item count > 999)" 2. "If an input condition specifies a number of values (e.g. "one through six owners can be listed for the automobile"), identify one valid class and two invalid equivalence classes (no owners and more than six owners)"	a. Select a partition containing values between the lower and upper boundary of a field (covers '1 ≤ item count ≤ 999' and 'one to six owners') b. Select a partition containing values below the lower boundary of a field (covers 'item count < 1' and '0 owners') c. Select a partition containing values below the lower boundary of a field (covers 'item count > 999' and 'more than six owners')	EP3: Lower to Upper Boundary Selection EP1: Less Than Lower Boundary Selection EP2: Greater Than Upper Boundary Selection
3. "If the input condition specifies a set of values and there is reason to believe that each is handled differently by the program (e.g. "type of vehicle must be BUS, TRUCK, TAXICAB, PASSENGER, or MOTORCYCLE"), identify a valid equivalence class for each one and one invalid equivalence class (e.g. "TRAILER")"	d. identify a valid equivalence class for each one	EP12: Valid List Selection
	e. identify one invalid equivalence class	EP4: Integer Replacement EP5: Real Number Replacement EP6: Single Alpha Replacement EP7: Multiple Alpha Replacement EP8: Multiple Alphanumeric Replacement EP9: Single Non-Alphanumeric Replacement EP10: Multiple Non-Alphanumeric Replacement
4. "If an input condition specifies a "must be" situation (e.g. "first character of the identifier must be a letter"), identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter)"	f. Select a partition containing all letters *(overlaps with f)*	EP12: Valid List Selection or EP3: Lower to Upper Boundary Selection
	g. Select a partition containing everything that is not a letter *(overlaps with f)*	EP4: Integer Replacement EP5: Real Number Replacement EP7: Multiple Alpha Replacement EP8: Multiple Alphanumeric Replacement EP9: Single Non-Alphanumeric Replacement EP10: Multiple Non-Alphanumeric Replacement
5. "If there is reason to believe that elements in an equivalence class are not handled in an identical manner by the program, split the equivalence class into smaller equivalence classes"	N/A	This guideline is part of the partitioning that occurs during application of guidelines 1 to 4. Thus, this does not require an Atomic Rule.

[2] See Appendix B for definitions of these Atomic Rules in the Atomic Rules characterisation schema.

Definition by exclusion in Myers' (1979) fourth guideline can also be resolved, as follows:

"If an input condition specifies a "must be" situation (e.g. "first character of the identifier must be a letter"), identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter)."

This guideline is decomposed into test case design rules that select valid partitions and 'replacement' rules that select invalid partitions by datatype (Table 3-4 col. 3, *EP3, EP4* to *EP10* and *EP12*). The selection of the valid partition 'it is a letter' is covered as follows[3].

- If the assumption is that that the input is treated by the programmer as a list then select an equivalence class containing all values in that list, which becomes *EP12*.
- If the assumption is that the input is treated as a range then select an equivalence class containing all values between the lower and upper boundaries of the field.

The selection of the invalid partition 'it is not a letter' is covered by defining the following 'replacement' rules that select invalid partitions by datatype:

- EP4: Integer Replacement, which replaces the field with an equivalence class containing integers
- EP5: Real Number Replacement, which replaces the field with an equivalence class containing real numbers
- EP7: Multiple Alpha Replacement, which replaces the field with an equivalence class containing multiple alphabetic characters
- EP8: Multiple Alphanumeric Replacement, which replaces the field with an equivalence class containing multiple alphanumeric characters
- EP9: Single Non-Alphanumeric Replacement, which replaces the field with an equivalence class containing individual non-alphanumeric characters
- EP10: Multiple Non-Alphanumeric Replacement, which replaces the field with an equivalence class containing multiple non-alphanumeric characters
- EP12: Null Item Replacement, which replaces the field with the empty set

One other replacement rule is required to select invalid equivalence classes for non-alpha fields:

- EP6: Single Alpha Replacement, which replaces the field with an equivalence class containing individual alphabetical characters

These replacement rules cover the 94 printable characters of the ASCII table, as well as contiguous datatypes 'integer' and 'real'[4]. They facilitate input domain partitioning by datatype, defining a 'universe of

[3] Depending on how the program was written, a programmer may treat the set of allowable alphabetical characters as an ASCII range (e.g. [ASCII(65) – ASCII(90)] or as a list (e.g. [A | B | C | … | Z]).

discourse' for program inputs (see Section 1.5). Replacement rules could be defined for other numerical datatypes such as 'float' and for other character sets such as those defined in the Unicode table (see Section 3.10.2.2).

Theoretically, replacement rules that select specific datatypes (i.e. *EP4* to *EP11*) can be applied to input fields of the same datatype, as long as they are used to select equivalence classes that exclude the valid inputs for that field. For example, *EP7: Multiple Alpha Replacement* could be applied to a field *<colour> ::= [red | green | blue]*, to select any combination of upper and lowercase alphabetical characters other than those in the valid set.

A number of other test case design rules for EP were utilised by Myers (1979) and other authors (see list below). Thus, the following Atomic Rules have been defined.

- EP14: Valid Test Case Constructor – Minimised, which constructs test cases by covering as many valid partitions as possible per test case (e.g. see (Myers 1979))
- EP15: Invalid Test Case Constructor – Maximised, which constructs test cases by covering as many invalid partitions as possible per test case (e.g. see (Myers 1979))
- EP16: Invalid Test Case Constructor – Minimised, which constructs test cases by covering one invalid partitions as possible per test case (e.g. see (BS 7925-2))
- EP18: Valid Test Case Constructor – Maximised, which constructs test cases by covering as many valid partitions as possible per test case (e.g. see (BS 7925-2))
- EP13: Random Data Value Selector, which covers the selection of a single randomly chosen test data value from an equivalence class (e.g. see (Jeng & Weyuker 1989)).
- EP17: Nominal Data Value Selector, which selects the mid-point of an equivalence class (e.g. see (Myers 1979))

Since Data-Set Selection Rules from EP are used by other black-box testing methods, they could be labelled as 'common' rules. As the strength of EP is in the selection of equivalence classes, these rules have remained within this method. This assumes that EP will always be used to identify equivalence classes prior to boundary value selection.

This completes the decomposition of EP into Atomic Rules.

[4] The particular range of integers or real numbers that is selected can depend on the programming language being used in the system under test, which is part of the "application solution domain" (Reed 1990) of the program. This information can be identified through grey-box testing and through GQASV (e.g. see Section 3.10.3).

Table 3-5: Three Atomic Rules from Equivalence Partitioning.

Attribute	Values	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning
Number	EP1	EP2	EP3
Identifier	LLBS	GUBS	LUBS
Name	Less than Lower Boundary Selection	Greater than Upper Boundary Selection	Lower to Upper Boundary Selection
Description	Select an equivalence class containing values below the lower boundary of a field	Select an equivalence class containing values above the upper boundary of a field	Select an equivalence class containing values between the boundaries of a field (including the on-boundary values)
Source	(Myers 1979)	(Myers 1979)	(Myers 1979)
Rule Type	DSSR	DSSR	DSSR
Set Type	Range	Range	Range
Valid or Invalid	Invalid	Invalid	Valid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric
Test Datatype	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original
# Fields Populated	1	1	1
# Tests Derived	0	0	0

For clarity, the following list explains how the attributes of one Atomic Rule, *EP1: Less Than Lower Boundary Selection* were assigned (see Table 3-5 col. 2).

1. **Test Method:** this Atomic Rule was derived from *Equivalence Partitioning*.
2. **Number:** this unique identifier combines an abbreviation of the test method name *EP* and an incremental number (starting at 1), resulting in *EP1*.
3. **Identifier:** *LLBS* is an abbreviation of the rule name 'Less than Lower Boundary Selection.'
4. **Name:** *Less than Lower Boundary Selection* describes the selection of a partition of data that lies below the lower boundary of a field.
5. **Description:** this describes the rule function in more detail. This rule selects a partition of data that lies below the lower boundary of a field.
6. **Source:** this rule was defined by Myers, thus its reference source is (Myers 1979).
7. **Rule Type:** this rule is a *Data-Set Selection Rule* (DSSR) because it partitions the input and/or output domain of a program.
8. **Set Type:** this rule applies to *ranges* of contiguous data.
9. **Valid or Invalid**: this selects an *invalid* partition that lies below the lower boundary of a field.
10. **Original Datatype:** since the rule only applies to range-based fields, it can only be applied to datatypes that are contiguous. Thus, it applies to datatypes *Integer*, *Real*, *Single Alpha* and *Single*

Non-Alphanumeric (if an ASCII table is used to select values below a lower boundary of a numeric, alpha or non-alphanumeric range).

11. **Test Datatype:** this rule does not alter the datatype of the original field. Thus, the value of this attribute is *Same as original*.
12. **Test Data Length:** *Same as original* appears under this attribute as the length of the selected test data depends on the length of the datatype the rule is being applied to. For example, if the rule is applied to an integer field, the programming language used to implement the program under test may impose a minimum to maximum range of -35535 to 35536.
13. **# Fields Populated:** this rule selects test data for 1 field at a time.
14. **# Tests Derived:** this rule does not construct test cases. Thus, the value of this attribute is 0.

The Atomic Rules four-step test case design process can now be documented for EP (Figure 3-3).

Figure 3-3: The four-step test design process for Equivalence Partitioning.

Equivalence Partitioning

1. Select equivalence classes as follows.
   a. If an input or output field has a *set type* ≡ *range* then apply Atomic Rules *EP1* to *EP11* to the field to select equivalence classes.
   b. If an input or output field has a *set type* ≡ *list* then apply Atomic Rules *EP4* to *EP12* to select equivalence classes.
2. Select one test data value from each equivalence class selected in step 1 by applying Atomic Rule *EP13* and/or *EP17* to each class.
3. Omit step 3, as EP does not cover test data manipulation.
4. Select test cases that are:
   a. *valid*: by applying *EP14* and *EP18* to valid test data values chosen in step 2;
   b. *invalid*: by applying *EP15* and *EP16* to invalid test data values chosen in step 2.

This section has demonstrated the decomposition of Myers' definition of EP into Atomic Rules in a manner which, as was pointed out at the beginning of this subsection, allows all major definitions of EP to be consolidated. Appendix B contains all Atomic Rules that were derived for EP, while Section 3.4.1.1 demonstrates the application of Atomic Rules from EP to an example input data specification.

3.3.1.2 Decomposing Boundary Value Analysis

Myers' guidelines for BVA (Table 3-6, col. 1) can also be decomposed into test case design rules (Table 3-6, col. 2) and then documented as Atomic Rules (Table 3-6, col. 3 and Table 3-7). During this process, *method overlaps* within BVA can be resolved (see Table 3-6, cols. 2 and 3). Myers' definition includes rules for testing the boundaries of range-based and list-based fields, as follows.

For *range-based fields*, Myers' (1979) provides the following guideline:

“If an input condition specifies a range of values, write test cases for the edges of the range, and invalid-input test cases for situations just beyond the ends. For instance, if the valid domain of an input value is -1.0 - +1.0, write test cases for the situations -1.0, 1.0, -1.001 and 1.001.”

This guideline can be decomposed into the following Atomic Rules (see Table 3-6, col. 3):

- BVA1: Lower Boundary – Selection
- BVA2: Lower Boundary Selection
- BVA5: Upper Boundary Selection
- BVA6: Upper Boundary + Selection

Table 3-6: Decomposition of Myers' definition of Boundary Value Analysis into Atomic Rules.

Myers' Guidelines for Boundary Value Analysis		Corresponding Atomic Rules[5]
Myers' (1979) Guidelines	**Test Case Design Rules**	
1. If an input condition specifies a range of values, write test cases for the edges of the range, and invalid-input test cases for situations just beyond the ends. For instance, if the valid domain of an input value is -1.0 - +1.0, write test cases for the situations -1.0, 1.0, -1.001 and 1.001.	a. Select a value on the lower boundary (covers -1.0)	BVA2: Lower Boundary Selection
	b. Select a value on the upper boundary (covers 1.0)	BVA5: Upper Boundary Selection
	c. Select a value just below the lower boundary (covers) -1.001	BVA1: Lower Boundary – Selection
	d. Select a value just above the upper boundary (covers 1.001)	BVA6: Upper Boundary + Selection
2. If an input condition specifies a number of values, write test cases for the minimum and maximum number of values and one beneath and beyond these values. For instance, if an input file can contain 1 – 255 records, write test cases for 0, 1, 255 and 256 records.	e. Select a value just below the lower boundary (covers 0 records) *(overlaps with a)*	BVA1: Lower Boundary – Selection
	f. Select a value on the lower boundary (covers 1 record) *(overlaps with b)*	BVA2: Lower Boundary Selection
	g. Select a value on the upper boundary (covers 255 records) *(overlaps with c)*	BVA5: Upper Boundary Selection
	h. Select a value just above the upper boundary (covers 256 records) *(overlaps with d)*	BVA6: Upper Boundary + Selection
3. Use guideline 1 for each output condition. For instance, if a program computes the monthly FICS deduction and if the minimum is $0.00 ad the maximum is $1165.25, write test cases that cause $0.00 and $1165.25 to be deducted. Also, see if it is possible to invent test cases that might causes a negative deduction or a deduction of more than $1165.25.	i. Test the lower boundary of the output field (covers $0.00)	BVA2: Lower Boundary Selection (can be applied only to the output field)
	j. Test the upper boundary of the output field (covers $1165.25)	BVA5: Upper Boundary Selection (can be applied only to the output field)
	k. Test just below the lower boundary of the field (covers negative deduction)	BVA1: Lower Boundary – Selection (can be applied only to the output field)
	l. Test just above the upper boundary of the field (covers more than $1165.25)	BVA6: Upper Boundary + Selection (can be applied only to the output field)
4. Use guideline 2 for each output condition. If an information retrieval system displays the most relevant abstracts based on an input request, but never more than four abstracts, write test cases such that the program displays zero, one and four abstracts, and write a test case that might cause the program to erroneously display five abstracts.	m. Run a query that tests just below the lower boundary of the output field (covers 0 abstracts)	BVA1: Lower Boundary – Selection (can be applied only to the output field)
	n. Run a query that tests on the lower boundary of the output field (covers 1 abstract)	BVA2: Lower Boundary Selection (can be applied only to the output field)
	o. Run a query that tests on the upper boundary of the output field (covers 4 abstracts)	BVA5: Upper Boundary Selection (can be applied only to the output field)
	p. Run a query that tests just above the upper boundary of the output field (covers 5 abstracts)	BVA6: Upper Boundary + Selection (can be applied only to the output field)
5. If the input or output of a program is an ordered set (e.g. a sequential file, linear list, table), focus attention on the first and last elements of the set.	q. First element	BVA7: First List Item Selection
	r. Last element	BVA8: Last List Item Selection
6. In addition, use your ingenuity to search for other boundary conditions.	NA	An Atomic Rule cannot be defined for this guideline.

[5] See Appendix B for definitions of these Atomic Rules in the Atomic Rules characterisation schema.

Myers' (1979) definition does not cover testing just inside the boundaries of range-based fields, which is included in other definitions of BVA (e.g. (BS 7925-2)). Thus, the following two rules are also defined.

- BVA3: Lower Boundary + Selection
- BVA4: Upper Boundary – Selection.

Each of these rules increment or decrement the upper or lower boundary of an equivalence class by one 'unit of measure', which is the minimum value that can be added or subtracted from the boundary according to the datatype of the field under test. For example, consider the examples below, where the Atomic Rule *BVA1: Lower Boundary – Selection* is applied to five different range-based fields.

1. If *BVA1* is applied to an integer partition *[1 – 9999]* it will subtract 1 from the lower boundary to select the test data value '0'.
2. If *BVA1* is applied to a real number partition *[2.00 – 4.00]* it will subtract 0.01 from the lower boundary to select the test data value '1.99'. Since real numbers may theoretically have infinite precision, the value that is added or subtracted from the boundary is the smallest possible decimal value from the field under test. The precision would typically depend on how the program was designed and may also depend on the maximum allowable field length rather than the allowable number of decimal places. If the precision of the field is not defined for the tester, they could use Goal/Question/Answer/Specify/Verify to obtain such a definition. In fact, difficulties can arise when the program does not actually constrain the precision of the input fields to that which was specified. For example, in this example, the program may accept 3.999 as input, but reject 4.001. In what follows, we assume that input precision is as specified, and that any deviations constitute faults that will be detected elsewhere.
3. If *BVA1* is applied to the alphabetical character partition *[A – Z]*, using the ASCII collating sequence, the test data value '@' would be selected as it lies just below the lower boundary of the partition (identification of test data values in this way may need to be guided through grey-box information; for example, by seeking guidance from developers).
4. If *BVA1* is applied to an non-alphanumeric range *[" – /]*, the test data value '!' would be selected as it lies just below the lower boundary of the partition in the collating sequence.
5. If *BVA1* is applied to a field that repeats, e.g. *<vehicle> ::= [car | truck | motorbike | van]*$^{1\text{-}4}$, it would subtract 1 from the lower boundary of the repetition partition *[1 – 4]*, which would test the field with *zero* repetitions (i.e. it would test the field with null).

For *list-based fields*, Myers' definition can be decomposed into the following two Atomic Rules.

- BVA7: First List Item Selection
- BVA8: Last List Item Selection

Although *BVA2: Lower Boundary Selection* and *BVA5: Upper Boundary Selection* could have been utilised to select the lower and upper boundaries of list-based fields, it was felt that a separate set of rules should be defined for lists, since only on-boundary and just-inside boundary values can be selected for lists.

While Myers' definition of BVA does not cover the testing of inside boundary values for list-based fields, this was used as a basis for selecting test cases by participants of an industry-based testing experiment that is discussed in Chapter 6. Thus, the following two Atomic Rules can be defined to cover these boundaries.

- BVA12: Second List Item Selection
- BVA13: Second Last List Item Selection

To ensure list-based and range-based fields are tested with null (e.g. for testing keyword-based fields with a zero string length), the following rule was also defined.

- BVA9: Null Item Replacement

Although *BVA9: Null Item Selection* overlaps with *EP11: Null Item Replacement*, both have been included to ensure that if only EP is applied to a program, then this rule will not be missed.

Myers (1979) also addresses testing the boundaries of **output** fields, as illustrated in his third and fourth guideline, for example:

> **"Use guideline 1 for each output condition. For instance, if a program computes the monthly FICS deduction and if the minimum is $0.00 ad the maximum is $1165.25, write test cases that cause $0.00 and $1165.25 to be deducted. Also, see if it is possible to invent test cases that might causes a negative deduction or a deduction of more than $1165.25."**

Although Atomic Rules *BVA1* to *BVA6* can be used to identify boundary values for **output** fields, using the current Atomic Rules approach, they cannot be used to identify the **input** values that are required to force the output boundaries to be exercised. For example, in Myers' third guideline the following boundary values are selected.

- Monthly FICS deduction = $0.00
- Monthly FICS deduction = $1165.25
- Monthly FICS deduction = negative deduction (i.e. -$0.01)
- Monthly FICS deduction = more than $1165.25 (i.e. $1165.26)

The **input** values that are needed to test these output boundaries currently cannot be selected by Atomic Rules, since the approach to converting inputs to outputs needs to be identified and understood either by a 'test oracle' (defined by Howden (1981)) or by a human tester (see Section 3.8 for more information). The input values that may be capable of producing these output BVA values often can only be obtained by

tracing backwards from them, through the application solution domain algorithms, to effectively 'solve the equations' represented by the specification. This is considered to be beyond the current work.

The four-step test case design process can now be demonstrated for BVA (Figure 3-4). Since boundary values are usually selected from the edges of equivalence classes (Myers 1979), partitioning rules from EP are utilised within this definition. Invalid datatype replacement rules *EP4* to *EP11* are not included, as it is unnecessary to select boundary values from partitions have already been covered during EP. Test Case Construction Rules from EP are utilised, since BVA does not include unique rules for test case selection. Appendix B covers definitions of all Atomic Rules for BVA, while Section 3.4.1.2 illustrates the application of Atomic Rules from BVA to an example input data specification.

Table 3-7: Three Atomic Rules from Boundary Value Analysis.

Attribute	Values	Values	Values
Test Method	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis
Number	BVA1	BVA2	BVA3
Identifier	LBM1	LB	LBP1
Name	Lower Boundary – Selection	Lower Boundary Selection	Lower Boundary + Selection
Description	Select value just below the lower boundary	Select a value on the lower boundary	Select a value just above the lower boundary
Source	(BS 7925-2)	(Myers 1979)	(Myers 1979)
Rule Type	DISR	DISR	DISR
Set Type	Range	Range	Range
Valid or Invalid	Invalid	Valid	Valid
Original Datatype	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric
Test Datatype	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original
# Fields Populated	1	1	1
# Tests Derived	0	0	0

Figure 3-4: The four-step test design process for Boundary Value Analysis.

Boundary Value Analysis

1. Select equivalence classes that exercise boundary values, as follows:
   a. If an input has a *set type* ≡ *range* then apply Atomic Rules *EP1*, *EP2* and *EP3* to select equivalence classes.
   b. If an input has a *set type* ≡ *list* then apply rule *EP12* to select equivalence classes.
2. Select boundary values as follows:
   a. For each range-based equivalence class chosen in step 1a, apply *BVA1* to *BVA6* and *BVA9* to select individual boundary values.
   b. For each list-based equivalence class chosen in step 1b, apply *BVA7* to *BVA13* to select individual boundary values.
3. Omit step 3, since BVA does not cover test data manipulation
4. Select test cases that are:
   a. *valid*: by applying *EP14* and *EP18* to valid test data values chosen in step 2;
   b. *invalid*: by applying *EP15* and *EP16* to invalid test data values chosen in step 2.

3.3.1.3 Decomposing Syntax Testing

Five different versions of ST were analysed to identify test case design rules and to identify corresponding Atomic Rules for this method (Table 3-8 cols. 1 and 2). During this process, *method overlaps* were able to be resolved both within the method and with other methods (Table 3-8, col. 3).

For example, one test case design rule defined for ST is to "Introduce an invalid value for a field" (Table 3-8, rule 2). This is covered by Atomic Rules from EP that select partitions outside field boundaries (*EP1* and *EP2*) and that replace fields with invalid datatypes (*EP4* to *EP11*). Although the first publications of syntax testing methods were in the 1960's (e.g. see (Sauder 1962)) and this preceded the first definition of EP as a black-box testing method in 1979 (Myers 1979), test case design rules from EP are generally defined in more detail. Thus, Atomic Rules from EP can be utilised during ST to provide more precision to test case design (see Figure 3-5, step 1). A complete mapping of test case design rules from ST to Atomic Rules from other methods is provided in Table 3-8.

From the five versions of ST that were reviewed, only five Atomic Rules were found to be unique to ST (see Table 3-8, rules 6, 7, 9, 28, 29). These are as follows.

- ST1: remove last character
- ST3: add extra character to end
- ST14: select all list alternatives in reverse order
- ST11: add a field
- ST10: duplicate a field

Based on the 'spirit' of these rules, a number of new Atomic Rules were defined for ST, as follows.

- ST2: replace last character (derived from Rule # 6 and 7)
- ST4: remove first character (derived from Rule # 6)
- ST5: replace first character (derived from Rule # 6 and 7)
- ST6: add extra character to start (derived from Rule # 6)
- ST7: uppercase a lowercase letter (derived from Rule # 2 for alphabetic fields)
- ST8: lowercase an uppercase letter (derived from Rule # 2 for alphabetic fields)
- ST9: Null all input (derived from Rule # 3)
- ST12: select each list alternative (derived from Rule # 9)
- ST13: select all list alternatives (derived from Rule # 9)

Each of these was documented in the Atomic Rules schema (e.g. see Table 3-9). The complete set of Atomic Rules for this method can be found in Appendix B. An example of this method being applied to an input data specification can be found in Section 3.4.1.3.

Table 3-8: Decomposition of Syntax Testing into Atomic Rules (from Section 2.4).

Rule # [6]	Syntax Testing Test Case Design Rules	Corresponding Atomic Rules[7]
1	Introduce errors at highest level of AST; e.g. through invalid field combinations	Method overlap – covered by Atomic Rules for combinatorial testing (see Appendix B)
2	Introduce an invalid value for a field	Method overlap – covered by *EP1*, *EP2* and *EP4* to *EP11*
3	Introduce an invalid value for all fields	Method overlap – covered by *EP4* to *EP11*
4	Choose invalid symbols for a field (e.g. subtraction instead of addition sign)	Method overlap – covered by *EP9*
5	Choose invalid datatypes (e.g. numbers or symbols instead of alphas)	Method overlap – covered by *EP4* to *EP11*
6	Remove characters from the end of a field (e.g. “DI” instead of “DIR”)	ST1: Remove Last Character
7	Add extra characters to the end of a field (e.g. “DIRR” instead of “DIR”)	ST3: Add Extra Character to End
8	Choose none of the legal alternatives for a field that contains alternatives	Method overlap – covered by *EP12*
9	Choose all alternatives for one field in one test case in reverse order	ST14: Select All List Alternatives in Reverse Order
10	Leave out a delimiter	Method overlap – covered by *EP12*
11	Choose a delimiter that is valid at another syntax level but not at the current level	Method overlap – covered by Atomic Rules for combinatorial testing (see Appendix B)
12	Substitute another field for a delimiter	Method overlap – covered by Atomic Rules for combinatorial testing (see Appendix B)
13	Repeat a delimiter	Method overlap – covered by *BVA6*
14	Create errors in paired delimiters (e.g. add or remove delimiters)	Method overlap – covered by *EP4* to *EP12*
15	One less than the minimum number of repetitions	Method overlap – covered by *BVA1*
16	Minimum number of repetitions	Method overlap – covered by *BVA2*
17	One more than min number of repetitions	Method overlap – covered by *BVA3*
18	1 repetition	Method overlap – covered by *BVA1* to *BVA6* (whether ‘1’ is a lower or upper boundary)
19	One less than max number of repetitions	Method overlap – covered by *BVA4*
20	Maximum number of repetitions	Method overlap – covered by *BVA5*
21	One more than the maximum number of repetitions	Method overlap – covered by *BVA6*
22	> 1 repetition	Method overlap – covered by *BVA1* to *BVA6* (whether ‘1’is a lower or upper boundary)
23	Incorrect value in the last repetition of a field	Method overlap – covered by *EP4* to *EP12*
24	Select invalid values for input fields	Method overlap – covered by *EP4* to *EP12*
25	Substitute a field that is correct at another level of syntax but not the current level	Method overlap – covered by Atomic Rules for combinatorial testing (see Appendix B)
26	Substitute fields from same level of syntax, creating invalid order of valid fields	Method overlap – covered by Atomic Rules for combinatorial testing (see Appendix B)
27	Miss a field	Method overlap – covered by *EP12*
28	Add an extra field	ST11: Add a Field
29	Repeat a field	ST10: Duplicate a Field
30	Select values relating to database variable type input is stored in. e.g. if field is string 0 to 255 characters, try 0, 255 and 256	Method overlap – covered by *BVA2*, *BVA5*, *BVA6*
31	State dependency errors	Method overlap – covered by EP, BVA and ST, but expected outcome depends on system state

[6] These rules and corresponding numbers were defined in Chapter 2, section 2.4.
[7] See Appendix B for definitions of these Atomic Rules in the Atomic Rules characterisation schema.

Table 3-9: Three Atomic Rules from Syntax Testing.

Attribute	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST1	ST2	ST3
Identifier	RMLC	RPLC	AECE
Name	Remove last character	Replace last character	Add extra character to end of field
Description	Remove the last character of an input string	Replace the last character of a string with an invalid value	Add an extra character to the end of a string
Source	(Beizer 1990, Marick 1995)	(Marick 1995)	(Beizer 1995, Marick 1995)
Rule Type	DISR	DISR	DISR
Set Type	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid
Original Datatype	All	All	All
Test Datatype	Same as original	Same as original	Same as original
Test Data Length	m - 1, where m is the original field length	Same as original	m + 1, where m is the original field length
# Fields Populated	1	1	1
# Tests Derived	0	0	0

The four-step test case design process can now be described for ST (Figure 3-5). As discussed in Section 2.4, partitioning is implicitly performed during ST prior to the selection and mutation of test data values. Thus, the ST process utilises Data-Set Selection Rules from EP. Also, since some versions of ST include the selection of boundary values and nominal values, corresponding Atomic Rules from EP and BVA are utilised (Table 3-8, col. 3). This process also utilises a number of combinatorial Test Case Construction Rules (Figure 3-5, step 4c). A demonstration of the Atomic Rules definition of ST being applied to an example specification is provided in Section 3.4.1.3. Although the process of creating an abstract syntax tree cannot be described specifically by the Atomic Rules approach, one could be constructed for the program under test, prior to the application of each Atomic Rule.

Figure 3-5: The four-step test design process for Syntax Testing.

Syntax Testing

1. Select equivalence classes as follows.
   a. If an input or output field has a *set type* ≡ *range* then apply Atomic Rules *EP1* to *EP11* to the field to select equivalence classes.
   b. If an input or output field has a *set type* ≡ *list* then apply Atomic Rules *EP4* to *EP12* to select equivalence classes.
2. Select individual test data values as follows.
   a. For each range-based equivalence class chosen in step 1a, apply *BVA1* to *BVA6*, *BVA9* and *EP13* and/or *EP17* to select individual test data values.
   b. For each list-based equivalence class chosen in step 1b, apply *BVA7* to *BVA13* to select individual test data values.
3. Manipulate the test data values chosen in step 2 by applying *ST1* to *ST14*, *ST17* and *ST18* to each value.
4. Select test cases that are:
   a. *valid*: by applying *ST14*, *EP14* or *EP16* to the valid test data values chosen in steps 2 and 3,
   b. *invalid*: by applying *ST15*, *EP15* or *EP16* to the invalid test data values chosen in steps 2 and 3.
   c. *combinatorial*: by applying a selection of Atomic Rules from *EP15*, *EP16* and *EP18*, *CT1* to *CT6* and *SBMT1* to *SBMT4*.

3.3.1.4 Decomposing the Combinatorial Method Each Choice

Combinatorial testing methods facilitate the generation of black-box test cases typically via the application of algorithms to the test data values that are derived through the use of other black-box testing methods, such as EP, BVA and ST. Each combinatorial test method can be decomposed into one Test Case Construction Rule that can then be described using the Atomic Rules schema. In this section, the method *All Combinations* is decomposed into an Atomic Rule as an example of this process (Table 3-10). Atomic Rules for other combinatorial methods are provided in Appendix B.

For *All Combinations*, the fields of the Atomic Rules schema are populated as follows.

1. **Test Method:** this Atomic Rule is from *Combinatorial Testing*.
2. **Number:** this unique identifier combines an abbreviation of the test method name (CT) with an incremental number (starting at 1), resulting in *CT1*.
3. **Identifier:** *AC* is an abbreviation of the rule name *All Combinations*.
4. **Name:** All Combinations is the name of the test method.
5. **Description:** this rule constructs test cases by generating all possible combinations of test data values chosen by other black-box testing methods. For example, it could be used to select all combinations of boundary values or all combinations of system configurations, such as when testing the compatibility of several Internet browsers on various different operating systems.
6. **Source:** this version of the method was sourced from (Grindal et al. 2004).

7. **Rule Type:** this is a *Test Case Construction Rule* (TCCR).
8. **Set Type:** this rule can be applied to any type of field; thus it applies to Lists and Ranges.
9. **Valid or Invalid**: this depends on whether this rule is applied to valid or invalid test data.
10. **Original Datatype:** this rule can be applied to an input field of any datatype.
11. **Test Datatype:** the rule does not alter the input field datatype; thus *Same as original* appears.
12. **Test Data Length:** *Same as original* appears under this attribute as the length of the selected test data depends on the length of the original datatype.
13. **# Fields Populated:** this rule populates all input fields of a test case; thus, it populates *n* fields, where *n* is the number of input fields in the test case.
14. **# Tests Derived:** this rule selects approximately $\prod_{i=1}^{N} V_i$ test cases, where N is the number of parameters in the input string and where each parameter has V_i values.

This results in the definition of one Atomic Rule for combinatorial testing (Table 3-10).

Table 3-10: An Atomic Rule for the combinatorial testing method All Combinations.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT1
Identifier	AC
Name	All Combinations
Description	Construct every possible combination of test data values, which may be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Grindal et al. 2004)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	Approximately $\prod_{i=1}^{N} V_i$ test cases, where N is the number of parameters in the input string and where each parameter has V_i values.

The four-step test case design process can now be defined once for all combinatorial testing methods (Figure 3-6). Since these methods do not include test case design rules for partitioning the input domain or selecting or manipulating test data values, Atomic Rules from EP, BVA and ST can be utilised.

Figure 3-6: The four-step test design process for Combinatorial Testing Methods.

Combinatorial Testing

1. Select equivalence classes as follows.
   a. If an input field has a *set type* ≡ *range* then apply Atomic Rules *EP1* to *EP11* to the field to select equivalence classes.
   b. If an input field has a *set type* ≡ *list* then apply Atomic Rules *EP4* to *EP12* to select equivalence classes.
2. Select individual test data values from the equivalence classes derived in step 1 by applying Atomic Rules *BVA1* to *BVA9* and *EP13* and/or *EP17* to each partition.
3. Optionally manipulate the test data values chosen in step 2 by applying *ST1* to *ST14*, *ST17* and *ST18* to each test data value.
4. Select test cases by applying a selection of Atomic Rules *CT1* to CT6 or *SBMT1* to *SBMT4* to the test data values chosen in steps 2 and 3.

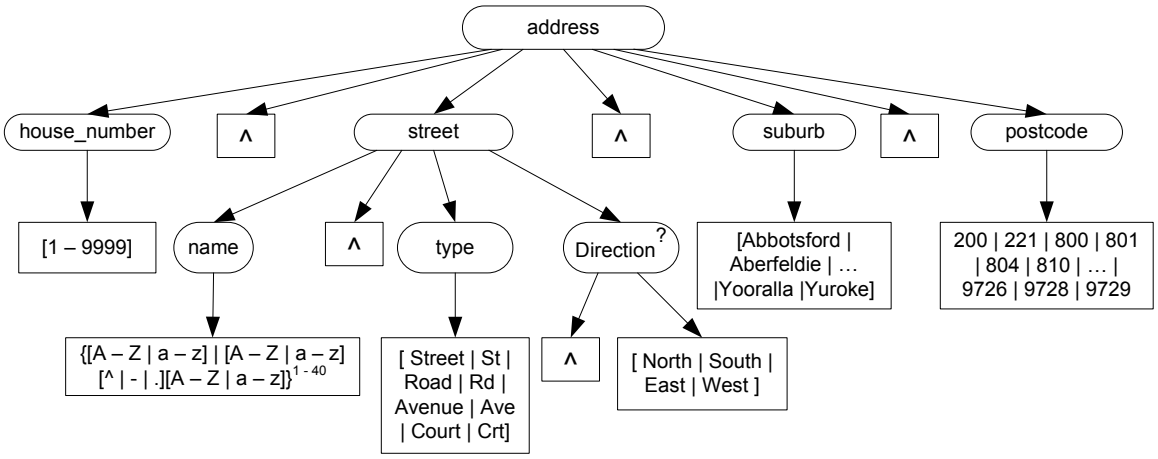
3.4 Demonstration of the Atomic Rules Approach

The application of the Atomic Rules representation of EP, BVA and ST is illustrated in the sections below for an input data specification of an Address Parser (Figure 3-7; the specification is an adaptation of one that was developed by Reed (1981)). The specification consists of many different field types, including a range field <*house_number*>, list fields <*street_type*> and <*direction*> and optional fields $\{\wedge$ <*direction*>$\}^{0\text{-}1}$ (where '$^{0\text{-}1}$' indicates that the fields are optional). It also includes a 'complex' field <*street_name*>, which can contain 1 to 40 alphabetical characters or alphabetical characters followed by a space, hyphen or period, followed by alphabetical characters.

The Atomic Rules approach requires a detailed definition of the program input domain, such as the one that is provided in Figure 3-7. If a detailed definition is not available, then Goal/Question/Answer/Specify/Verify could be used to obtain one.

Figure 3-7: Simplified specification of the inputs to an Address Parser program.

```
     <address>  ::= <house_number> ∧ <street> ∧ <suburb> ∧ <postcode>
<house_number>  ::= [1 – 9999]
      <street>  ::= <street_name>  ∧ <street_type> {∧ <direction>}0-1
 <street_name>  ::= {[A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]}1 - 40
 <street_type>  ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]
   <direction>  ::= [North | South | East | West ]
      <suburb>  ::= [Abbotsford | Aberfeldie | … |Yooralla |Yuroke]
    <postcode>  ::= [200 | 221 | 800 | 801 | 804 | 810 | … | 9726 | 9728 | 9729]
                ∧   ::= [ ]  #i.e. one space
```



Usually, when the Atomic Rules definition of a black-box testing method is applied to an input data specification, every Atomic Rule from the method would be systematically applied to every applicable input field. For the Address Parser, Atomic Rules with a 'Set Type' of 'Range' would be applied to all range-based fields. For example, *EP1* to *EP3* could be applied to the <*house_number*>field to select equivalence classes, followed by *BVA1* to *BVA6* to select boundary values. These same rules could be applied to the 'length' of <*street_name*> (i.e. to the 1 to 40 character range). Similarly, Atomic Rules with a 'Set Type' of 'List' could be applied to <*street_type*>, <*direction*>, <*suburb*>, <*postcode*> and whitespace (∧). However, to reduce the scale of examples in the sections below, Atomic Rules from EP, BVA and ST are only applied to a small number of input fields (see sections for details).

The Test Matrix below provides a complete mapping of Atomic Rules from EP, BVA and ST to the input fields of this program (Table 3-11). While this can result a very large number of test data values and test cases, it also enables very thorough black-box testing. When such rigorous testing is not required, the Test Matrix can be used to assess the maximum rule-to-field coverage achievable, from which a tester can selectively apply a subset of Atomic Rules. Automatic generation of test data and test cases can also assist with this process by increasing test case derivation efficiency (see Chapter 4).

Table 3-11: Test Matrix indicating which Atomic Rules from EP, BVA and ST can be applied to the input fields of the Address Parser program.

Atomic Rule #	Atomic Rule Name	**Key:** ✓ indicates that the Atomic Rule can be applied to the field. × indicates that the Atomic Rule cannot be applied to the field due to incompatibility between the Rule Type and the Set Type of the field. **Input Fields**							
		<house_number> ::= [1 – 9999]	<street_name> ::= {[A – Z | a – z] | [A – Z | a – z] [^ | - | .][A – Z | a – z]} (i.el contents only)	<street_name> length 1to 40 characters	<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	<direction> ::= [North | South | East | West]	<suburb> ::= [Abbotsford | Aberfeldie | … |Yooralla |Yuroke]	<postcode> ::= [200 | 221 | 800 | 801 | 804 | 810 | … | 9726 | 9728 | 9729]	^::= [ ] #i.e. one space
EP1	Less Than Lower Boundary Selection	✓	✓[8]	✓	×	×	×	×	✓[9]
EP2	Greater Than Upper Boundary Selection	✓	✓	✓	×	×	×	×	✓
EP3	Lower to Upper Boundary Selection	✓	✓	✓	×	×	×	×	✓
EP4	Integer Replacement	×	✓	×	✓	✓	✓	✓	✓
EP5	Real Number Replacement	✓	✓	×	✓	✓	✓	✓	✓
EP6	Single Alpha Replacement	✓	×	×	✓	✓	✓	✓	✓
EP7	Multiple Alpha Replacement	✓	×	×	✓	✓	✓	✓	✓
EP8	Multiple Alphanumeric Replacement	✓	✓	×	✓	✓	✓	✓	✓
EP9	Single Non-Alphanumeric Replacement	✓	✓	×	✓	✓	✓	✓	✓
EP10	Multiple Non-Alphanumeric Replacement	✓	✓	×	✓	✓	✓	✓	✓
EP11	Null Item Replacement	✓	✓	✓	✓	✓	✓	✓	✓
EP12	Valid List Item Selection	×	✓	×	✓	✓	✓	✓	✓
EP13	Random Data Value Selector	✓	✓	✓	✓	✓	✓	✓	✓
EP14	Valid Test Case Constructor – Maximised	✓	✓	✓	✓	✓	✓	✓	✓
EP15	Invalid Test Case Constructor – Maximised	✓	✓	✓	✓	✓	✓	✓	✓
EP16	Invalid Test Case Constructor – Minimised	✓	✓	✓	✓	✓	✓	✓	✓
EP17	Nominal Value Selector	✓	✓	✓	✓	✓	✓	✓	✓
EP18	Valid Test Case Constructor – Maximised	✓	✓	✓	✓	✓	✓	✓	✓
BVA1	Lower Boundary – Selection	✓	✓[10]	✓	×	×	×	×	×
BVA2	Lower Boundary Selection	✓	✓	✓	×	×	×	×	×
BVA3	Lower Boundary + Selection	✓	✓	✓	×	×	×	×	×
BVA4	Upper Boundary – Selection	✓	✓	✓	×	×	×	×	×
BVA5	Upper Boundary Selection	✓	✓	✓	×	×	×	×	×
BVA6	Upper Boundary + Selection	✓	✓	✓	×	×	×	×	×
BVA7	First List Item Selection	×	✓[11]	×	✓	✓	✓	✓	✓
BVA8	Last List Item Selection	×	✓	×	✓	✓	✓	✓	✓
BVA9	Null Item Replacement	✓	✓	✓	✓	✓	✓	✓	✓
BVA10	Attempt First List Item – Selection	×	✓	×	✓	✓	✓	✓	×
BVA11	Attempt Last List Item + Selection	×	✓	×	✓	✓	✓	✓	×
BVA12	Second List Item Selection	×	✓	×	✓	✓	✓	✓	×
BVA13	Second Last List Item Selection	×	✓	×	✓	✓	✓	✓	×
ST1	Remove last character	✓	✓	×	✓	✓	✓	✓	✓
ST2	Replace last character	✓	✓	×	✓	✓	✓	✓	✓
ST3	Add extra character to end	✓	✓	×	✓	✓	✓	✓	✓

[8] EP1, EP2 and EP3 can be applied to the contents of the <*street_name*> field, if character ranges [A – Z] and [a – z] are treated as ASCII ranges by the program (i.e. as ASCII 65 to 90 and ASCII 79 to 122).
[9] EP1 to EP3 can be applied to the number of times the whitespace appears (i.e. once is valid, more or less than once is invalid).
[10] BVA1 to BVA6 can be applied to the edges of the alphabetic ranges [A – Z] and [a – z] if they are treated as ASCII ranges.
[11] BVA7 and BVA8 can be applied to the character partitions [A – Z] and [a – z] if they are treated by the program as lists.

Atomic Rule #	Atomic Rule Name	**Key:** ✓ indicates that the Atomic Rule can be applied to the field. × indicates that the Atomic Rule cannot be applied to the field due to incompatibility between the Rule Type and the Set Type of the field.							
		Input Fields							
		<house_number> ::= [1 – 9999]	<street_name> ::= {[A – Z | a – z] | [A – Z | a – z] [^ | - | .][A – Z | a – z]} (i.el contents only)	<street_name> length 1to 40 characters	<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	<direction> ::= [North | South | East | West]	<suburb> ::= [Abbotsford | Aberfeldie | ... |Yooralla |Yuroke]	<postcode> ::= [200 | 221 | 800 | 801 | 804 | 810 | ... | 9726 | 9728 | 9729]	^::= [ ] #i.e. one space
Table continued from previous page…									
ST4	Remove first character	✓	✓	×	✓	✓	✓	✓	✓
ST5	Replace first character	✓	✓	×	✓	✓	✓	✓	✓
ST6	Add extra character to start	✓	✓	×	✓	✓	✓	✓	✓
ST7	Uppercase a lowercase letter	×	✓	×	✓	✓	✓	×	×
ST8	Lowercase an uppercase letter	×	✓	×	✓	✓	✓	×	×
ST9	Null all input	✓	✓	✓	✓	✓	✓	✓	✓
ST10	Duplicate field	✓	✓	✓	✓	✓	✓	✓	✓
ST11	Add a field	✓	✓	✓	✓	✓	✓	✓	✓
ST12	Select each list alternative	×	✓	×	✓	✓	✓	✓	✓
ST13	Select all list alternatives	×	✓	×	✓	✓	✓	✓	✓
ST14	Select all list alternatives in reverse order	×	✓	×	✓	✓	✓	✓	✓
ST15	Reference Replacement	✓	✓	✓	✓	✓	✓	✓	✓
ST16	Syntax Cover	✓	✓	✓	✓	✓	✓	✓	✓
ST17	Add Middle Character	✓	✓	✓	✓	✓	✓	✓	✓
ST18	Remove Middle Character	✓	✓	✓	✓	✓	✓	✓	✓
ST19	Reverse All Fields	✓	✓	✓	✓	✓	✓	✓	✓

3.4.1.1 Applying the Atomic Rules Definition of Equivalence Partitioning

In this example, the Atomic Rules representation of EP is applied to a range-based field <*house_number*>, a list-based field <*street_type*> and the 'complex' field <*street_name*>. These fields were chosen as they cover a wide range of field types. EP implements three of the four black-box test case design steps. In step 1, partitions are identified by applying *EP1* to *EP11* to <*house_number*>, *EP4* to *EP12* to <*street_type*> and *EP1* to *EP12* to <*street_name*> (Table 3-12). In step 2, *EP17* is applied to each partition to select the nominal value from each class (Table 3-13). Step 3 is not applied since it is only used by test methods that manipulate test data values (Table 3-14). In step 4, *EP14* and *EP16* are applied to derive two example test cases (Table 3-14). In practice, this fourth step would be applied repeatedly until all test data values have been covered by at least one test case. Steps 1, 2 and 4 would be also be applied to all other input fields (i.e. <*street_name*>, <*direction*>, <*suburb*>, <*postcode*> and whitespace).

Table 3-12: Example of applying step 1 of the Atomic Rules definition of Equivalence Partitioning to an example Address Parser specification.

Step 1. Select equivalence classes as follows:
a. if set type ≡ range then apply Atomic Rules EP1 to EP11,
b. if set type ≡ list then apply Atomic Rules EP4 to EP12.

Input Field Definition	Equivalence Classes		
Field: <house_number> **Field Type:** Range **Field Definition:** <house_number> ::= [1 – 9999]	**Atomic Rule #**	**Equivalence Class Selected**	**Valid or Invalid**
	EP1	values < 1	Invalid
	EP2	values > 9999	Invalid
	EP3	values 1 to 9999	Valid
	EP4	any integer, except values 1 to 9999	Invalid
	EP5	any real number	Invalid
	EP6	any single alpha character	Invalid
	EP7	any multiple alpha characters	Invalid
	EP8	any multiple alphanumeric characters	Invalid
	EP9	any single non-alphanumeric symbol	Invalid
	EP10	any multiple non-alphanumeric symbols	Invalid
	EP11	null	Invalid
Field: <street_name> **Field Type:** List and Range **Field Definition:** <street_name> ::= {[A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]}$^{1-40}$	***Partitions for <street_name> length: 1 – 40***		
	Atomic Rule #	**Equivalence Class Selected**	**Valid or Invalid**
	EP1	< 1 character	Invalid
	EP2	> 40 characters	Invalid
	EP3	1 to 40 characters	Valid
	Partitions for <street_name> contents: [A – Z | a – z | ∧ | -]		
	Atomic Rule #	**Equivalence Class Selected**	**Valid or Invalid**
	EP4	any integer	Invalid
	EP5	any real number	Invalid
	EP8	any multiple alphanumeric characters	Invalid
	EP9	any single non-alphanumeric symbols excluding ∧ and -	Invalid
	EP10	any multiple non-alphanumeric symbols excluding ∧ and -	Invalid
	EP11	null – already covered by application of EP1 to <street_name> length	
	EP12	Anything from the valid list [A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]	Valid[12]
Field: <street_type> **Field Type:** List **Field Definition:** <street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	**Atomic Rule #**	**Equivalence Class Selected**	**Valid or Invalid**
	EP4	any integer	Invalid
	EP5	any real number	Invalid
	EP6	any single alpha character	Invalid
	EP7	any multiple alpha characters, except those in valid set	Invalid
	EP8	any multiple alphanumeric characters	Invalid
	EP9	any single non-alphanumeric symbol	Invalid
	EP10	any multiple non-alphanumeric symbols	Invalid
	EP11	null	Invalid
	EP12	Anything from the valid list [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	Valid

[12] When the "∧" character is chosen as a test data value, it will print as one white space " ".

Table 3-13: Example of applying step 2 of the Atomic Rules definition of Equivalence Partitioning to an example Address Parser specification.

Step 2. Select one test data value from each equivalence class selected in step 1 by applying EP13 or EP17 to each class (EP17 is applied in this particular example).		
Equivalence Classes	**Test Data Values**	
For the <house_number> field:	**Test Data Values Selected by EP17**	**Valid or Invalid**
value < 1	-15000	Invalid
value > 9999	15000	Invalid
value 1 to 9999	5000	Valid
any integer excluding 1 to 9999	10000	Invalid
any real number	23.53	Invalid
any single alpha character	M	Invalid
any multiple alpha characters	OnM	Invalid
any multiple alphanumeric characters	O56M	Invalid
any single non-alphanumeric symbol	>	Invalid
any multiple non-alphanumeric symbols	>=<	Invalid
null		Invalid
For the length of the <street_name>:	**Corresponding Test Data Values**	**Valid or Invalid**
Street name length < 1 character		Invalid
Street name length > 40 characters	abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz	Invalid
Street name length 1 to 40 characters	abcdefghijkl	Valid[13]
For the contents of the <street_name>:	**Corresponding Test Data Values**	**Valid or Invalid**
any integer	10000	Invalid
any real number	23.53	Invalid
any multiple alphanumeric characters	O56M	Invalid
any single non-alphanumeric symbol except ∧ and -	>	Invalid
any multiple non-alphanumeric symbols except ∧ and -	>=<	Invalid
		Invalid
anything from the valid list [A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]	Lm nop	Valid
For the <street_type> field:	**Corresponding Test Data Values**	**Valid or Invalid**
any integer	10000	Invalid
any real number	23.53	Invalid
any single alpha character	M	Invalid
any multiple alpha characters, except valid set	OnM	Invalid
any multiple alphanumeric characters	O56M	Invalid
any single non-alphanumeric symbol	>	Invalid
any multiple non-alphanumeric symbols	>=<	Invalid
null		Invalid
anything from the valid list [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	Rd	Valid

[13] Testing the <*street_name*> field with test data values "abcdefghijklmnopqurt" and "lm nop" will test the program for how it handles syntactically valid and invalid test data. This will not test the program with semantically correct test data. Semantic correctness would be tested if the valid street names were derived from a valid street name file.

Table 3-14: Example of applying steps 3 and 4 of the Atomic Rules definition of Equivalence Partitioning to an example Address Parser specification.

<table>
<tr><td colspan="2">Step 3. This step manipulates test data values and is currently covered by Syntax Testing. Thus, it is not applicable.</td></tr>
<tr><td colspan="2">Step 4. Select test cases that are:
a. valid, by applying EP14 and EP18 to valid data values chosen in step 2,
b. invalid, by applying EP15 and EP16 to invalid data values chosen in step 2.</td></tr>
<tr><th>Test Data Values</th><th>Test Cases</th></tr>
<tr><td>Test data values for <house_number>
-15000
15000
5000
10000
23.53
M
OnM
O56M
>
>=<
Test data values for <street_name>
(i.e. null)
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
abcdefghijkl
10000
23.53
O56M
>
>=<
Lm nop
Test data values for <street_type>
10000
23.53
M
OnM
O56M
>
>=<
(i.e. null)
Road</td><td>Depending on which Test Case Construction Rule was applied (EP14, EP15 or EP16), this would result in a large number of test cases. Thus, the complete test set that would are derivable is not shown here. Instead, two example test cases are shown.

Atomic Rule EP14 (Valid Test Case Constructor) could be applied to cover valid test data values <house_number> = 5000, <street_name> = abcdefghijkl and <street_type> = Rd, while all other fields were assigned their nominal value, to select the following test case.

5000 abcdefghijklmnopqurt Rd La Trobe 3086

Atomic Rule EP16 (Invalid Test Case Constructor - Minimised) could be applied to derive a test case to cover the invalid test data value <house_number> = -10000, while all other fields were assigned their nominal value, as follows.

-10000 abcdefghijklmnopqurt Rd La Trobe 3086

In the above, two test cases have been derived covering four test data values. In practice, this would be repeated until all test data values (in the left hand column) are covered by at least one test case.</td></tr>
</table>

3.4.1.2 Applying the Atomic Rules Definition of Boundary Value Analysis

In the following example, the Atomic Rules definition of BVA is applied to the same three fields of the Address Parser <*house_number*>, <*street_name*> and <*street_type*> as follows. In step 1, Atomic Rules from EP are utilised to partition the input domain (Table 3-15). In step 2, *BVA1* to *BVA9* are applied to select boundary values from each partition (Table 3-15 step 2). Step 3 is not applied since this only applies to test methods that manipulate test data values (i.e. ST) (Table 3-16). In step 4, *EP14* and *EP16* are utilised twice to derive two example test cases (Table 3-17). In practice, step 4 would be repeated until all test data values had been covered by at least one test case, while steps 1, 2 and 4 would be reapplied for all other input fields.

Table 3-15: Example of applying step 1 of the Atomic Rules definition of Boundary Value Analysis to an example Address Parser specification.

Step 1. Select equivalence classes as follows:
a. if set type ≡ range then apply Atomic Rules EP1 to EP3,
b. if set type ≡ list then apply Atomic Rule EP12.

Input Field Definition	Equivalence Classes		
Field: <house_number> **Field Type:** Range **Field Definition:** <house_number> ::= [1 – 9999]	**Atomic Rule #**	**Equivalence Class Selected**	**Valid or Invalid**
	EP1	values < 1	Invalid
	EP2	values > 9999	Invalid
	EP3	values 1 to 9999	Valid
Field: <street_name> **Field Type:** List and Range **Field Definition:** <street_name> ::= {[A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]}$^{1-40}$	**Equivalence classes for <street_name> length:** 1 – 40		
	Atomic Rule #	**Equivalence Class Selected**	**Valid or Invalid**
	EP1	< 1 character	Invalid
	EP2	> 40 characters	Invalid
	EP3	1 to 40 characters	Valid
	Equivalence classes for <street_name> contents: [A – Z | a – z | ∧ | -]		
	Atomic Rule #	**Equivalence Class Selected**	**Valid or Invalid**
	EP12	Anything from the valid list [A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z]	Valid
Field: <street_type> **Field Type:** List **Field Definition:** <street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	**Atomic Rule #**	**Equivalence Class Selected**	**Valid or Invalid**
	EP12	Anything from the valid list [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	Valid

Table 3-16: Example of applying step 2 of the Atomic Rules definition of Boundary Value Analysis to an example Address Parser specification.

Step 2. Select boundary values as follows. a. For each range-based equivalence class chosen in step 1a, apply BVA1 to BVA6 and BVA9 to select individual data values. b. For each list-based equivalence class chosen in step 1a, apply BVA7 to BVA13 to select individual data values			
Equivalence Classes	**Test Data Values**		
For the <house_number> field: **value < 1** Assume value is stored as a 16-bit integer, minimum value −32768 **Partition tested: [-32768 – 0]**	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	-32769	Invalid
	BVA2	-32768	Invalid
	BVA3	-32767	Invalid
	BVA4	-1	Invalid
	BVA5	0	Invalid
	BVA6	1	Valid
	BVA9		Invalid
For the <house_number> field: **value > 9999** Assume value is stored as a 16-bit integer, maximum value 32767 **Partition tested: [10000 – 32767]**	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	9999	Valid
	BVA2	10000	Invalid
	BVA3	10001	Invalid
	BVA4	32766	Invalid
	BVA5	32767	Invalid
	BVA6	32768	Invalid
For the <house_number> field: Value 1 to 9999	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	0 (already covered)[14]	NA
	BVA2	1 (already covered)	NA
	BVA3	2	Valid
	BVA4	9998	Invalid
	BVA5	9999 (already covered)	NA
	BVA6	10000 (already covered)	NA
For the length of the <street_name>: Street name length < 1 character Minimum length = 0 **Partition tested: [0 – 1]**	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	cannot select string of < 0 characters	NA
	BVA2	0 characters (already covered)	NA
	BVA3	1 character	Valid
	BVA4	0 character (already covered)	NA
	BVA5	1 character (already covered)	NA
	BVA6	2 characters	Valid
	BVA9	0 characters (already covered)	NA
For the length of the <street_name>: Street name length > 40 characters Assume 8-bit string, maximum length = 256 Partition tested: [41 – 256]	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	39 characters	Valid
	BVA2	40 characters	Invalid
	BVA3	41 characters	Invalid
	BVA4	255 characters	Invalid
	BVA5	256 characters	Invalid
	BVA6	257 characters	Invalid
For the length of the <street_name>: Street name length 1 to 40 characters	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA1	0 characters (already covered)	NA
	BVA2	1 character (already covered)	NA
	BVA3	2 characters (already covered)	NA
	BVA4	39 characters (already covered)	NA
	BVA5	40 characters (already covered)	NA
	BVA6	41characters (already covered)	NA
For the contents of the <street_name>: anything from the valid list [A – Z | a – z] | [A – Z | a – z] [∧ | - | .][A – Z | a – z] Assume the characters are stored in lists rather than ASCII ranges	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA7	A	Valid
	BVA7	a	Valid
	BAV7	(white-space)	Valid
	BVA7	-	Valid
	BVA7	.	Valid
	BVA8	Z	Valid
	BVA8	z	Valid
	BVA8	(white-space) (already covered)	NA
	BVA8	- (already covered)	NA
	BVA8	. (already covered)	NA
	BVA9	(null) (already covered)	NA
	BVA12	B	Valid
	BVA12	b	Valid
	BVA13	Y	Valid
	BVA13	y	Valid
For the <street_type> field: anything from the valid list [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	**Atomic Rule #**	**Boundary Value Selected**	**Valid or Invalid**
	BVA7	Street	Valid
	BVA8	Crt	Valid
	BVA9	(null) (already covered)	NA

[14] If a boundary has previously been covered by another test data value, then it does not need to be covered again.

Table 3-17: Example of applying steps 3 and 4 of the Atomic Rules definition of Boundary Value Analysis to an example Address Parser specification.

<table>
<tr><td colspan="2">Step 3. This step manipulates test data values and is currently covered by Syntax Testing. Thus, it is not applicable for BVA.</td></tr>
<tr><td colspan="2">Step 4. Select test cases that are:
a. valid, by applying EP14 and EP18 to valid data values chosen in step 2,
b. invalid, by applying EP15 and EP16 to invalid data values chosen in step 2.</td></tr>
<tr><th>Test Data Values</th><th>Test Cases</th></tr>
<tr><td>Test data values for <house_number>
-32769 10000
-32768 10001
-32767 32766
-1 32767
0 32768
1 9998
2 9999 9998
Test data values for <street_name> Length
0 characters 41 characters
1 character 255 characters
2 characters 256 characters
39 characters 257 characters
40 characters
Test data values for <street_name> Conrtents
A Z
A z
(white space) B
- b
. Y
y
Test data values for <street_type>
Street
Crt</td><td>Depending on which Test Case Construction Rule was applied (EP14, EP15 or EP16), this would result in a large number of test cases. Thus, the complete test set that would are derivable is not shown here. Instead, two example test cases are shown.

Atomic Rule EP14 (Valid Test Case Constructor) could be applied to cover valid test data values <house_number> = 1, <street_name> = A and <street_type> = Street, with all other fields being assigned their nominal value, which could result in the following test case[13].

1 A Street La Trobe 3086

Atomic Rule EP16 (Invalid Test Case Constructor – Minimised) could be applied to cover the invalid test data value <street_name> length = 0 characters, with all other fields being assigned their nominal value, as follows.

1 Street La Trobe 3086

In the above, two test cases have been derived, covering four test data values. In practice, this would be repeated until all test data values (in the left hand column) are covered by at least one test case.</td></tr>
</table>

3.4.1.3 Demonstration of Syntax Testing

The Atomic Rules definition of ST can be applied to the <*house_number*> field of the Address Parser specification (Figure 3-8) as follows. ST implements all four steps of the black-box test case design process, as follows. In step 1, Atomic Rules *EP1* to *EP11* are utilised to partition the <*house_number*> field into equivalence classes (Figure 3-8 step 1). In step 2, *EP13* is utilised to select random values from each equivalence class (Figure 3-8, step 2). In step 3, *ST2* is applied to manipulate the test data values chosen in step 2 (Figure 3-8, step 3), In practice, step 3 would be repeated with *ST1* to *ST8, ST17* and *ST18* to manipulate the test data values in other ways. In step 4, *EP15* is applied to derive a sample of test cases (Figure 3-8, step 4). In practice, the fourth step would also be repeated until all test data values chosen in step 3 were covered by at least one test case. All four steps in this process would also be repeated for all other input fields of the Address Parser specification.

Figure 3-8: Demonstration of the application of the Atomic Rules definition of Syntax Testing.

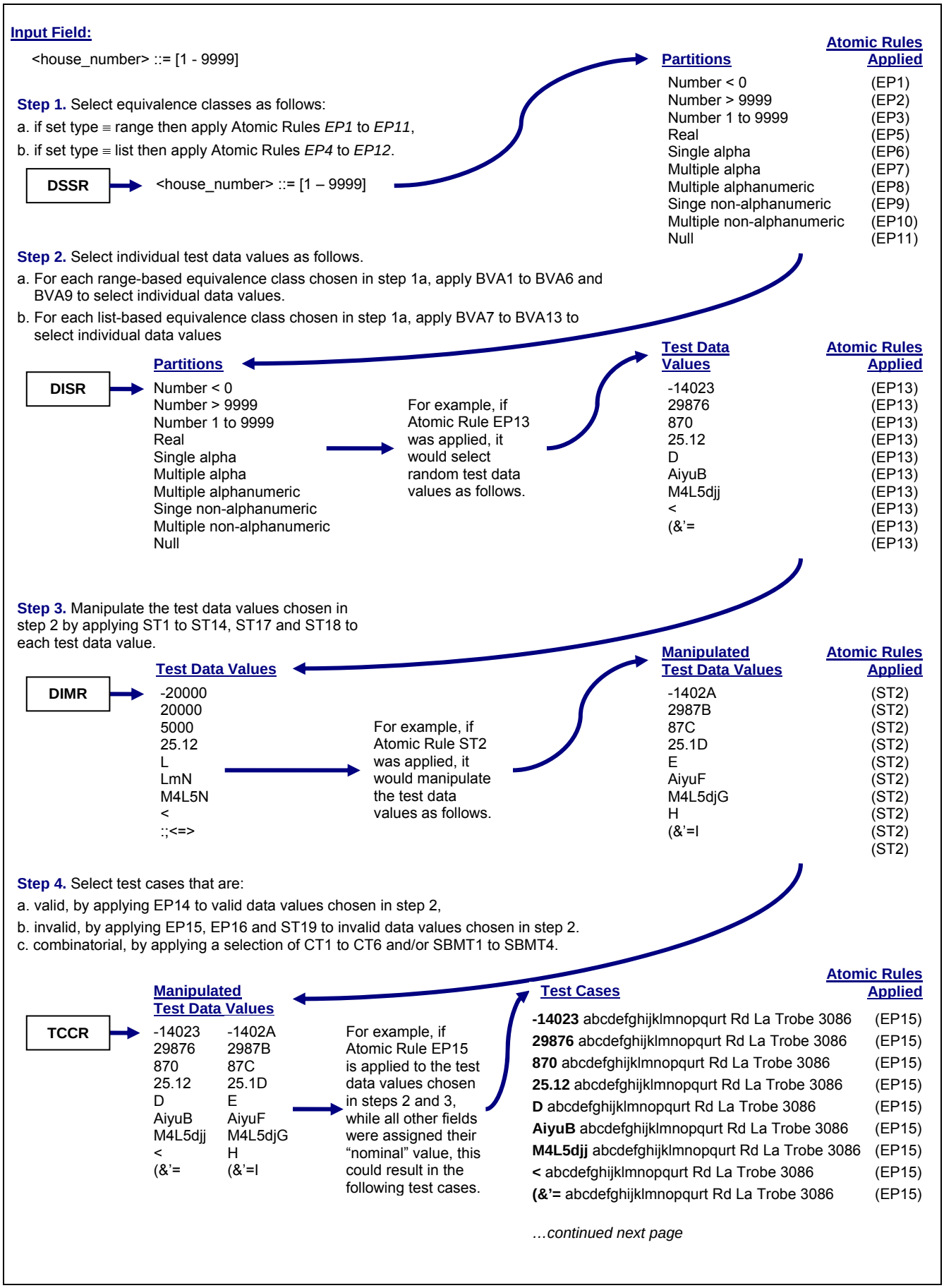
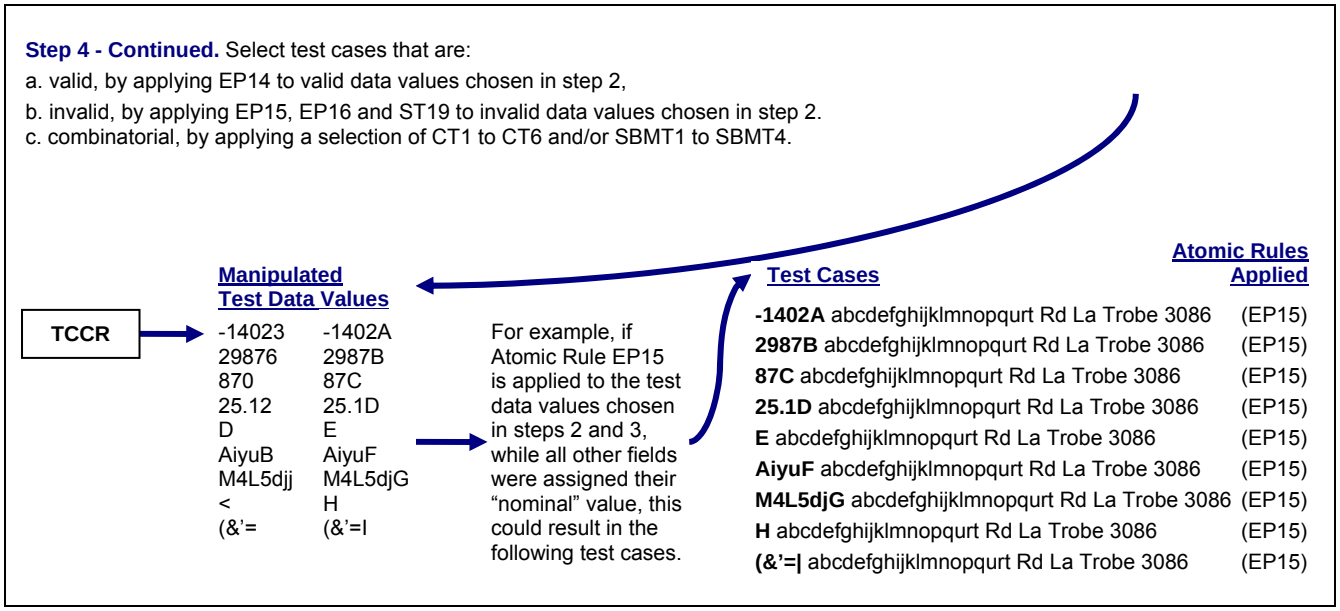


Figure 3-9: Demonstration of the application of the Atomic Rules definition of Syntax Testing (continued).



3.5 Auditing the Completeness of Black-Box Testing

A major benefit of the Atomic Rules approach is that it simplifies auditing the completeness of black-box testing through the use of Test Matrices. For example, consider the coverage that has been achieved so far by applying EP to the Address Parser specification in the previous section. Through the Atomic Rules approach, this coverage can be easily traced via a Test Matrix that tracks which Atomic Rules have (or have not) been applied to each input field (Table 3-18).

This level of traceability is particularly useful for organisations that are developing software to meet regulatory standards that dictate the use of particular black-box testing methods. For example, the railway engineering standard EN 50128 (which is used by railway engineering organisations in Australia including Westinghouse and Ansaldo STS) "highly recommends" the use of EP and BVA for testing any safety-related system (BS-EN 50128:2001). An assessor who is checking an organisation's compliance to this standard would require the organisation to demonstrate that they have applied EP and BVA adequately. Since the standard refers to Myers' (1979) definition of these methods, the assessor may not be able to definitively determine whether all possible equivalence classes and boundary values had been covered during testing. In contrast, this would be very obvious and much more easily demonstrable for any organisation using the Atomic Rules approach.

Table 3-18: Coverage of Atomic Rules from Equivalence Partitioning achieved against the fields of the Address Parser specification.

Atomic Rule #	Atomic Rule Name	**Key:** • ✓ indicates Atomic Rules that have been applied to each field. • **NA** indicates Atomic Rules that cannot be applied to each field due to incompatibility between the Rule Type and the Set Type of the field. • **Whitespaces** indicate Atomic Rules that have not yet been applied. **Input Fields**							
		<house_number> ::= [1 – 9999]	<street_name> ::= {[A – Z | a – z] | [A – Z | a – z] [^ | - | .][A – Z | a – z]} (i.e. contents only)	<street_name> length 1 to 40 characters	<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]	<direction> ::= [North | South | East | West]	<suburb> ::= [Abbotsford | Aberfeldie | … |Yooralla |Yuroke]	<postcode> ::= [200 | 221 | 800 | 801 | 804 | 810 | … | 9726 | 9728 | 9729]	^::= [ ] #i.e. one space
EP1	Less Than Lower Boundary Selection	✓	NA	✓	NA	NA	NA	NA	NA
EP2	Greater Than Upper Boundary Selection	✓	NA	✓	NA	NA	NA	NA	NA
EP3	Lower to Upper Boundary Selection	✓	NA	✓	NA	NA	NA	NA	NA
EP4	Integer Replacement	✓	✓	NA	✓				
EP5	Real Number Replacement	✓	✓	NA	✓				
EP6	Single Alpha Replacement	✓	✓	NA	✓				
EP7	Multiple Alpha Replacement	✓	✓	NA	✓				
EP8	Multiple Alphanumeric Replacement	✓	✓	NA	✓				
EP9	Single Non-Alphanumeric Replacement	✓	✓	NA	✓				
EP10	Multiple Non-Alphanumeric Replacement	✓	✓	NA	✓				
EP11	Null Item Replacement	✓	✓		✓				
EP12	Valid List Item Selection		✓		✓				
EP13	Random Data Value Selector								
EP14	Valid Test Case Constructor – Minimised	✓	✓		✓				
EP15	Invalid Test Case Constructor – Maximised								
EP16	Invalid Test Case Constructor – Minimised	✓							
EP17	Nominal Value Selector	✓	✓	✓	✓				
EP18	Valid Test Case Constructor – Maximised								

3.6 Using the Atomic Rules Approach in other Black-Box Testing

The Atomic Rules defined for EP, BVA and ST can be used to enable prescriptive test data design for supporting other black-box testing methods, such as:

- State Transition Testing (a form of model-based testing);
- Use Case Testing; and
- Category Partition Testing and Classification Trees.

This is demonstrated in the following three sections.

3.6.1 Applying Atomic Rules to State Transition Testing

In State Transition Testing, test cases are derived from a model of the state behaviour of a program to test its behaviour when certain events occur (e.g. see (BS 7925-2)). A State Transition Diagram illustrates

the states a program may occupy, transitions between states, events that cause transitions between states and actions or outcomes of the transition, where appropriate. Test cases are constructed to exercise transitions, states and combinations thereof. At the simplest level, test cases comprise a start state, an event (i.e. an input) that causes a transition, (usually) an outcome and an end state.

Definitions of State Transition Testing typically describe techniques for constructing test cases that ensure that valid transitions are correctly exercised for valid sequences of events and that 'error' states are reached in response to invalid input. They do not provide prescriptive guidelines for selecting (valid and invalid) test data values that are required to 'activate' each transition. Atomic Rules from EP, BVA and ST can be utilised to fill this gap, providing precise method of deriving those test data values.

Consider the specification for a component *manage_display_changes* that controls a digital clock display (Figure 3-10) (originally defined in (BS 7925-2)). The clock can either display the date or time or can be in states where it is being reset. The inputs to the component are commands to set the date (DS) and time (TS), to change modes between displaying the date and time (CM) and reset (R). From the textual specification of this component, a state transition diagram can be drawn (Figure 3-11).

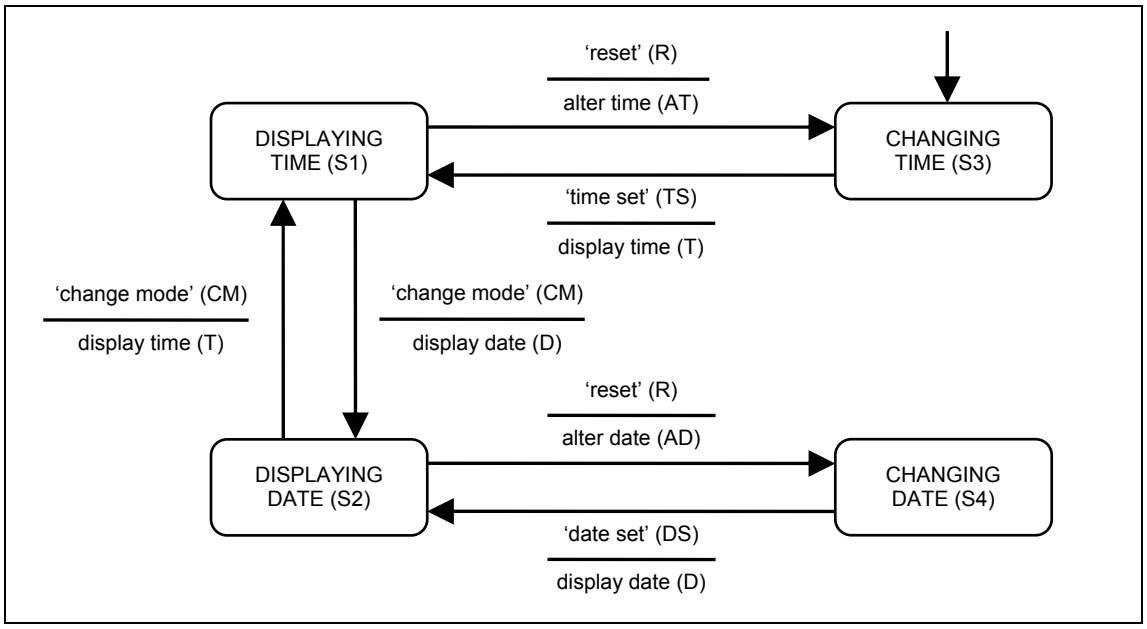
Figure 3-10: Specification for a component that manages the display on a clock (BS 7925-2).

Component manage_display_changes

The component responds to input requests to change an externally held display mode for a time display device. The external display mode can be set to one of four values: Two correspond to displaying either the time or the date, and the other two correspond to modes used when altering either the time or date.

There are four possible input requests: 'Change Mode', 'Reset', 'Time Set' and 'Date Set'. A 'Change Mode' input request shall cause the display mode to move between the 'display time' and 'display date' values. If the display mode is set to 'display time' or 'display date' then a 'Reset' input request shall cause the display mode to be set to the corresponding 'alter time' or 'alter date' modes. The 'Time Set' input request shall cause the display mode to return to 'display time' from 'alter time' while similarly the 'Date Set' input request shall cause the display mode to return to 'display date' from 'alter date'.

Figure 3-11: State Transition Diagram for 'manage_display_changes' (from (BS 7925-2)).



Six test cases are required to cover the six individual transitions in the State Transition Diagram (Table 3-19) (BS 7925-2). This is commonly known as "0-switch" coverage (Chow 1978).

Table 3-19: Test cases to achieve 0-switch coverage of manage_display_changes (BS 7925-2).

Test Case	1	2	3	4	5	6
Start State	S1	S1	S3	S2	S2	S4
Input	CM	R	TS	CM	R	DS
Expected Output	D	AT	T	T	AD	D
Finish State	S2	S3	S1	S1	S4	S2

Since State Transition Testing focuses only on transition coverage, not input domain coverage, Table 3-19 is missing actual *test data values* that would need to be included with the commands 'date set' (DS) and 'time set' (TS), in order to cause transition from one state to another. For example, it is assumed that the tester would need to enter an actual test data values like '1200' to set the time to midday. The Atomic Rules approach can be utilised at this point to enable prescriptive test data design when selecting inputs to these test cases. For example, one approach to specifying the time field is *<time> ::=[00:00 – 23:59]*. If the transition from state S3 to S1 were being tested, then the following Data-Set Selection Rules from EP could be applied to this field to select equivalence classes.

1. EP1: < 00:00 (invalid and presumably impossible, but worth recording as a potential test)
2. EP2: > 23:59 (invalid)
3. EP3: 00:00 to 23:59 (valid)

The <time> field could also be specified as <time> ::= <hh>:<ss>, where <hh> ::= [00 – 23] and <ss> ::= [00 – 59], which would ensure that the boundaries of the 'seconds' field are appropriately tested (e.g. see Table 3-22 in Section 3.6.3).

Data-Item Selection Rules *EP13* (random value), *EP17* (nominal value) and *BVA1* to *BVA6* (boundary values) could then be applied to each partition to select test data values. For example, they could be applied to partition number 3 to select the following values.

4. Partition 3: 00:00 to 23:59 (valid)
    a. EP13: 13:34 (valid)
    b. EP17: 12:00 (valid)
    c. BVA1: -99:99 (invalid and assumed impossible, but worth noting)
    d. BVA2: 00:00 (valid)
    e. BVA3: 00:01 (valid)
    f. BVA4: 23:58 (valid)
    g. BVA5: 23:59 (valid)
    h. BVA6: 23:60 (invalid)

Execution of the transition from S3 to S1 with these test data values would ensure that the transition is tested and source code that validates the input data just prior to transition is also rigorously tested.

The selection of other invalid test data values (e.g. invalid datatypes) is limited by the input domain of the system under test. Unlike a program that takes inputs from a keyboard, the input domain of this system is limited by the available buttons on the clock interface, some of which will be considered invalid at state S3. Since there are no existing Atomic Rules in EP, BVA or ST that can be used to select this type of invalid equivalence class, a new rule *STT1* can be defined as follows.

5. STT1: Invalid List Selection (see Appendix B, Section B.4)
    a. Selects the invalid partition *[date set | change mode | reset]*

EP18 was defined specifically to support state transition testing. Since there are only three items selected by the rule in this invalid partition, it would be reasonable to design test cases that cover all three values. Test Case Construction Rule *ST12* could be applied derive these.

This example has illustrated how the Atomic Rules approach can be utilised to provide prescriptive test data design to support State Transition Testing.

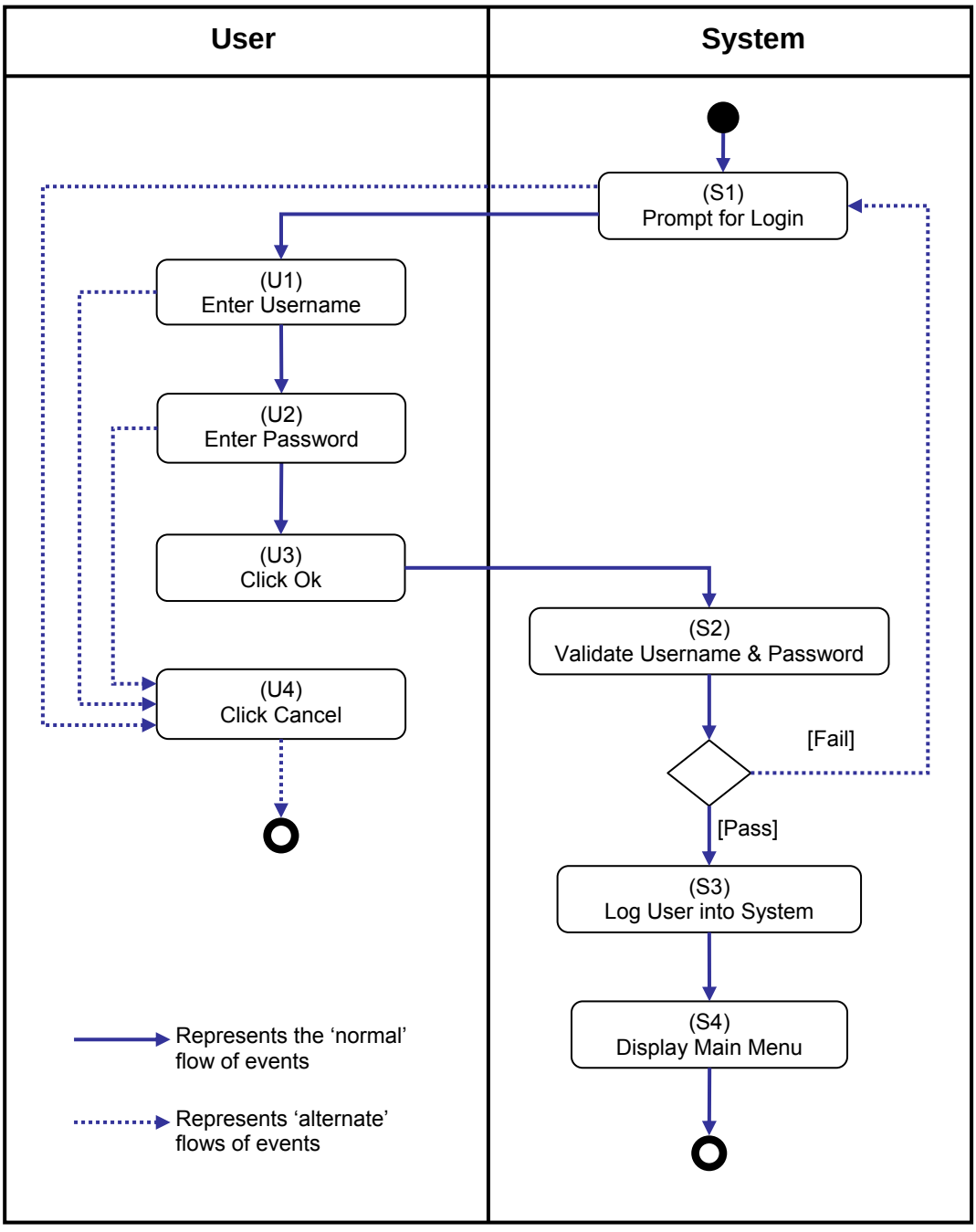
3.6.2 Applying Atomic Rules to Use Case Testing

Activity diagrams allow the flow of events through a use case to be specified by depicting the interaction between the system and the user (Chonoles & Schardt 2003) and are one of the modelling notations defined in the Unified Modelling Language (UML). Business Analysts and developers in industry

often use Activity Diagrams to specify business workflows through systems, while testers can use them to identify test cases for testing those workflows. Atomic Rules from EP, BVA and ST can be used to prescriptively generate inputs for populating test cases derived through Use Case Testing.

Consider an activity diagram for a login screen (Figure 3-12). The 'normal' flow of events tracks the user entering a valid username and password and clicking the 'Ok' button, while alternate flows cover the user clicking 'Cancel' and entering an invalid username/password combination (Figure 3-12).

Figure 3-12: An activity diagram that illustrating the flow of events for a login screen.



Test cases can be derived from this diagram to cover normal (Table 3-20) and alternate flows.

Table 3-20: Test case covering the 'normal' path of login (adapted from (Nguyen et al. 2003)).

Use Case ID	Login 01
Use Case Name	Login Screen
Test Case Number	TC1
Test Case Name	Login Screen – Normal
Path Covered	S1, U1, U2, U3, S2, S3, S4
Pre-conditions	User has started the program
Inputs	Enter valid username Enter valid password Click Ok button
Expected Results	User is logged into system, Main Menu is displayed
Post-conditions	Flag is set in database to show user is logged in

Although these test cases ensure that workflows through the system are thoroughly tested, use case testing itself does not provide assistance with the selection of test data values for population of the 'Inputs' field of the test case (Table 3-20 row 7). Atomic Rules from EP, BVA and ST can be utilised at this point, to provide prescriptive design of valid and invalid test data values.

For example, let us assume that the username and password fields may contain between 8 and 20 lowercase alphas, uppercase alphas, integers [0 – 9] and all non-alphanumeric ASCII characters. The following Data-Set Selection Rules could be applied to test the contents of the two fields.

1. EP12: partition containing all lowercase alpha characters (valid)
2. EP12: partition containing all uppercase alpha characters (valid)
3. EP12: partition containing all non-alphanumeric characters (valid)
4. EP12: partition containing integers [0 – 9] (valid)

Invalid datatype selection rules EP4 to EP12 do not have to be applied, since all datatypes they cover are included in the partitions above. However, the length of the field can be tested.

5. EP1: < 8 characters (invalid)
6. EP2: > 20 characters (invalid)
7. EP3: 8 to 20 characters (valid)

A variety of Data-Item Selection Rules could then be applied to select individual test data values. For example, DISR *EP13* could be applied to select a random value from each partition, as follows.

1. EP12: partition containing all lowercase alpha characters
    a. Apply EP17 to select character 'b'
2. EP12: partition containing all non-alphanumeric characters
    a. Apply EP17 to select character 'F'

3. EP12: partition containing all non-alphanumeric characters
    a. Apply EP17 to select '+'
4. EP12: partition containing integers [0 – 9]
    a. Apply EP17 to select integer '3'
5. EP1: < 8 characters
    a. Apply EP17 to select 2 characters
6. EP2: > 20 characters
    a. Apply EP17 to select 58 characters
7. EP3: 8 to 20 characters
    a. Apply EP17 to select 14 characters

A test case could be designed to cover 1a, 1b, 1c, 1d and 7a, resulting in a test data value *bF+3bF+3bF+3bF*, which would execute the normal path (Table 3-21) (this test data was constructed by repeating the pattern of test data values that were selected under 1a, 1b, 1c, 1d until the 14 characters defined by 7a were covered). For the test to pass, the database would need to include a 'valid' user with username and password *bF+3bF+3bF+3bF*.

Table 3-21: Updated test case covering the 'normal' path of the login screen.

Use Case ID	Login 01
Use Case Name	Login Screen
Test Case Number	TC1 – version 2
Test Case Name	Login Screen – Normal
Path Covered	S1, U1, U2, U3, S2, S3, S4
Pre-conditions	User has started the program
Inputs	Enter valid username *bF+3bF+3bF+3bF* Enter valid password *bF+3bF+3bF+3bF* Click Ok button
Expected Results	User is logged into system, Main Menu is displayed
Post-conditions	Flag is set in database to show user is logged in

This example has illustrated how the Atomic Rules approach can be utilised to prescriptively derive test data values for supporting Use Case Testing.

3.6.3 Applying Atomic Rules to the Category Partition Method

The Category Partition Method (CPM) was developed by Ostrand and Balcer (1988) to formalise the documentation of black-box test cases in a language they named the 'Test Specification Language'. One of the criticisms of the CPM in Chapter 2 (Section 2.9) is that it does not provide any guidance on the selection of 'choices' (i.e. equivalence classes) for input fields. The Atomic Rules approach can be utilised during step 2 of the CPM to formalise the process of identifying 'choices' via the application of Data-Set Selection Rules from EP.

Consider the component *manage_display_changes* (Table 3-22, col. 1), which was introduced in Section 3.6.1. The input fields in this specification are defined in BNF, enabling thorough black-box testing of this component[15] (see Table 3-22, col. 1, *Syntax*). Data-Set Selection Rules from EP can be applied to each input field, to select a set of 'choices' (i.e. partitions), which can then be documented in the Test Specification Language (see Table 3-22, col. 2). This demonstrates how the Atomic Rules approach can be utilised by CPM to formalise the process of deriving and documenting choices.

Table 3-22: Specification of the manage_display_changes command.

Component Specification	Corresponding TSL Specification*
Command manage_display_changes **Syntax** <manage_display_changes> ::= <command> [<date> | <time>]$^{0-1}$ <command> ::= [CM | R | DS | TS] <date> ::= <day><month><year> <day> ::= [1 – 31] <month> ::= [1 – 12] <year>[16] ::= [1970 – 3000] <time> ::= <hh><ss> <hh> ::= [00 – 23] <ss> ::= [00 – 59] **Function – manage_display_changes** The component responds to input requests to change an externally held display mode for a time display device. The external display mode can be set to one of four values: Two correspond to displaying either the time or the date, and the other two correspond to modes used when altering either the time or date. There are four possible input requests: 'Change Mode', 'Reset', 'Time Set' and 'Date Set'. A 'Change Mode' input request shall cause the display mode to move between the 'display time' and 'display date' values. If the display mode is set to 'display time' or 'display date' then a 'Reset' input request shall cause the display mode to be set to the corresponding 'alter time' or 'alter date' modes. The 'Time Set' input request shall cause the display mode to return to 'display time' from 'alter time' while similarly the 'Date Set' input request shall cause the display mode to return to 'display date' from 'alter date'. **Examples** DS 01012000 displays the date 01/01/2000 TS 1459 displays the time 14:59 (i.e. 2:59 pm) R resets the date or time that was displayed CM Changes the display from date to time and visa versa	**Equivalence Classes for each Parameter** **Command** EP12: valid item from list [CM | R | DS | TS] **Day** EP1: day < 1 EP2: day > 31 EP3: day 1 – 31 **Month** EP1: month < 1 EP2: month > 12 EP3: month 1 – 12 **Year** EP1: year < 1970 EP2: year > 3000 EP3: year 1970 – 3000 **hh** EP1: hh < 00 EP2: hh > 23 EP3: hh 00 – 23 **ss** EP1: ss < 00 EP2: ss > 59 EP3: ss 00 – 59 **Environments** Not required, as only one command can be given to the system at once. ***** Additional test cases could be identified by a human tester for testing valid and invalid combinations of days, months and years. For example, testing days like February 29 within leap years and non-leap years or testing for the 31st day in months with and without 31 days. This form on of combination testing is outside the scope of the Atomic Rules approach (see Section 3.8).

[15] The syntax for this input could have been defined using Goal/Question/Answer/Specify/Verify (GQASV) (see Section 3.10).
[16] This year range was chosen as the following formula (implemented in C#) can be utilised to calculate leap years between these dates: return (year % 4 == 0) && (year % 100 != 0) || (year % 400 = 0) (returns true for leap years) (Page et al. 2009).

3.7 Evolution of the Atomic Rules Approach

The first version of the Atomic Rules approach (Murnane, Hall & Reed 2005) was defined through an analysis of EP and BVA. This definition was then improved by carrying out experiments with university students and industry professionals (see Chapters 5 and 6) and by extending the research to include ST. Since EP and BVA do not manipulate (i.e. 'mutate') test data, the original Atomic Rules test case design process included only three steps (Figure 3-13). The fourth step of the process was added when the investigation was extended to ST (e.g. the definition of ST in (BS 7925-2) includes field manipulation).

Figure 3-13: The original Atomic Rules three-step test selection process (Murnane et al. 2005).

1. Select valid and invalid data sets called 'equivalence classes' or 'partitions' for each input field by applying a *Data Set Selection Rule* (DSSR) to each field.
2. Select at least one individual data value from each partition chosen in (1) by applying a *Data Item Selection Rule* (DISR) to each partition.
3. Select combinations of data values chosen in (2) to construct test cases by applying a *Test Case Construction Rule* (TCCR) to the data values.

Atomic Rules for EP were enhanced, based on the outcome of the university experiment. In the original EP rule set, datatype replacement rules *EP4* to *EP11* could only be applied to fields of a different datatype to that of the field under test. For example, *EP4: Integer Replacement* could only be applied to non-integer fields. During the university experiment it was realised that replacement rules could be applied to fields of the same datatype, provided that the equivalence class selected excludes all values in the valid class. This is evident in a previous example where *EP7: Multiple Alpha Replacement* was applied to the <*street_type*> to select any multiple alpha character not in the valid set (see Section 3.4).

The final improvement was the definition of new Atomic Rules for BVA (*BVA10* to *BVA13*) and ST (*ST17* to *ST19*), which were identified by participants of the industry experiment.

3.8 Limitations of the Atomic Rules Approach

One limitation of the Atomic Rules approach is that it does not support testing of input field dependencies, where the test data value chosen from the equivalence class of one field depends on the value chosen from the class of another field. Consider the example of a date picker whose implementation is based on the Gregorian calendar and which is composed of three input fields.

```
  <day>  ::= [1 – 31]
<month>  ::= [1 – 12]
 <year>  ::= [1582 – 9999]
```

Invalid combinations include 31/2 and selecting 29 days in non-leap years such as 29/2/2009. While a function could be defined for calculating whether a particular combination is possible, Atomic Rules cannot be defined for selecting these as aside from automatic selection, they can only be manually chosen by a tester who has appropriate domain knowledge of valid day/month/year combinations.

A similar limitation is that Atomic Rules cannot select combinations of test data values for testing output partitions (introduced in Section 3.3.1.2), as this can only be carried out manually by a human tester. Consider the following example that demonstrates the application of EP to the specification of a component *generate_grading*, which takes as input a coursework mark out of 25 and an exam mark out of 75 and generates a total grade in the range 'A' to 'D' (BS 7925-2) (Figure 3-14).

Figure 3-14: Specification for the component generate_grading (BS 7925-2).

Component generate_grading

The component is passed an exam mark (out of 75) and a coursework (c/w) mark (out of 25), from which it generates a grade for the course in the range 'A' to 'D'. The grade is calculated from the overall mark which is calculated as the sum of the exam and c/w marks, as follows:

greater than or equal to 70 - 'A'

greater than or equal to 50, but less than 70 - 'B'

greater than or equal to 30, but less than 50 - 'C'

less than 30 - 'D'

Where a mark is outside its expected range then a fault message ('FM') is generated. All inputs are passed as integers.

Partitions can be identified for *coursework mark* and *exam mark* by applying *EP1* to *EP3*:

1. EP1: coursework mark < 0 (invalid)
2. EP2: coursework mark > 25 (invalid)
3. EP3: coursework mark 0 – 25 (valid)
4. EP1: exam mark < 0 (invalid)
5. EP2: exam mark > 75 (invalid)
6. EP3: exam mark 0 – 75 (valid)

Invalid datatype replacement rules can also be applied to these fields, to ensure that Fault Messages appears for non-integers. Since *EP4* is covered by partitions 1, 2, 4 and 5, it is not included below.

7. EP5: real number in place of the coursework mark (invalid)
8. EP5: real number in place of the exam mark (invalid)
9. EP5: a single alpha character in place of the coursework mark (invalid)
10. EP5: a single alpha character in place of the exam mark (invalid)
11. EP7: multiple alphas characters in place of the coursework mark (invalid)
12. EP7: multiple alphas characters in place of the exam mark (invalid)

13. EP8: multiple alphanumeric characters in place of the coursework mark (invalid)
14. EP8: multiple alphanumeric characters in place of the exam mark (invalid)
15. EP9: a single non-alphanumeric character in place of the coursework mark (invalid)
16. EP9: a single non-alphanumeric character in place of the exam mark (invalid)
17. EP10: multiple non-alphanumeric characters in place of the coursework mark (invalid)
18. EP10: multiple non-alphanumeric characters in place of the exam mark (invalid)
19. EP11: null in place of the coursework mark (invalid)
20. EP11: null in place of the exam mark (invalid)

Each partition could then be tested by applying *EP13* or *EP17* to select nominal or random values.

The complexity arises when the output partitions of *total mark* are considered, as follows.

21. EP3: grade 'A' induced by total mark [70 – 100] (valid)
22. EP3: grade 'B' induced by total mark [50 – 69] (valid)
23. EP3: grade 'C' induced by total mark [30 – 49] (valid)
24. EP3: grade 'D' induced by total mark [0 – 29] (valid)
25. EP1: Fault Message induced by total mark < 0 (invalid)
26. EP2: Fault Message induced by total mark > 100 (invalid)

To ensure all output partitions are 'covered' by at least one test case, test data values for *exam mark* need to be chosen based on the value of *coursework mark*. For example, test data values *exam mark = 20* and *coursework mark = 60* exercise partition number 21. It is not possible to define an Atomic Rule that will mechanically identify all combinations of input exam marks and coursework marks that cause a particular output partition to be exercised. A human tester is required to complete this step. Although this is a current limitation of the approach, this could be solved in future research (see Chapter 7).

3.9 Related Research

The Atomic Rules characterisation schema provides a classification system for representing black-box testing methods more prescriptively. Such classification systems are common in software engineering. Characterisation schemas have been used to standardise other software engineering 'technologies' facilitating the selection of appropriate techniques with respect to specific problem domains.

For example, Prieto-Díaz and Freeman used faceted classification to develop a system for the characterisation and retrieval of reusable code components (Prieto-Díaz 1991, Prieto-Díaz & Freeman 1987). Maiden and Rugg (1996) used faceted classification to develop the ACquisition of REquirements (ACRE) framework, which guides the selection of suitable requirement acquisition (RA) methods. ACRE consists of twelve methods that are chosen as representatives of all methods, six facets that determine method selection and six tables that rank how well each method fits the terms of each facet. Continuing this

research, Rugg, McGeorge and Maiden (2000) decomposed RA and knowledge acquisition (KA) methods into discrete sub-tasks called "method fragments." They noted that while there were there a variety of methods available, many had common sub-tasks. Also, many versions of each method exist, some of which produce different results given the same elicitation problem. Method decomposition made their strengths and weaknesses easier to assess. ACRE also facilitated the development of customised elicitation methods. These concepts align well with the Atomic Rules approach and were used during the development of the approach to refine the concept of Atomic Rules, the identification of method versions and the development of Systematic Method Tailoring.

Birk (1997) used faceted classification and the Goal/Question/Metric (GQM) paradigm (Basili & Selby 1984, Basili & Weiss 1984, Basili 1991, Basili 1992, Basili et al. 1994) to develop a schema that characterized software 'technologies' and aided in their selection. The aim was to develop a decision support system for technology selection. Future work on the prototype Atomic Rules testing tool will include such a system in which users can select the Rules that apply to their specific problem domain (see Chapter 4).

Vegas, Juristo and Basili (2003) developed an instantiated characterization schema to classify testing methods, to facilitate the selection of the "best suited" methods for specific projects (see Chapter 2, Section 2.5.3). While their schema enabled classification of various black-box testing methods, it did not specifically provide solutions to the seven problems with existing black-box testing methods that are addressed in this thesis, nor did it address the need for precise, prescriptive methods. It also does not clearly identify the conditions under which one specific black-box testing method should be used over another. For example, there is little difference between the definitions of BVA and RT (see Chapter 2, Table 2-10).

3.10 Goal/Question/Answer/Specify/Verify and Systematic Method Tailoring

Goal/Question/Answer/Specify/Verify and Systematic Method Tailoring are two additional approaches that were developed in support of the Atomic Rules approach. They are ultimately aimed at enabling more effective black-box testing. GQASV is described first (Section 3.10.1), followed by SMT (Section 3.10.2). A demonstration of both approaches is also provided, for an online, real-world foreign currency calculator (Section 3.10.3).

3.10.1 Goal/Question/Answer/Specify/Verify (GQASV)

An unstated requirement for the effective use of black-box testing methods is that program input fields are completely specified. As discussed in Chapter 2, a survey of software testing practices in Australia found that of sixty-five organisations interviewed, over half reported that 20-59% of their program defects were related to specification defects (Ng et al. 2004). For a specific example of field incompleteness, consider the following real-life scenario. A business analyst (BA) working on a requirements specification for financial software discovered that the program under development needed to validate credit card numbers. The BA assumed that all members of the project team were familiar with valid credit card number

formats and thus omitted specifying their input data format explicitly. Consequently, during testing, the system tester was unable to derive a complete or effective black-box test set for that requirement and thus was unable to verify whether that part of the program satisfied its end-user needs. The input field definitions were also difficult to extract from program requirements, partly as their data and behaviour were specified in multiple places. As Abbot observed (1986), a program's functionality, inputs, outputs and error checking are often discussed over various (separate) sections of a specification and can even appear in completely separate specifications. Consequently, there exists a need for a requirement elicitation procedure that ensures program input/output fields are completely specified, thus enabling effective black-box testing.

3.10.1.1 The GQASV Process

GQASV is proposed as a simple yet readily applied requirement elicitation procedure that can be used by business analysts or testers to develop precise definitions of program input and output fields, as well as to gather domain knowledge that is useful for conducting effective black-box testing (Murnane, Reed & Hall 2006). GQASV is a modest extension of the well-known Goal/Question/Metric paradigm (Basili & Selby 1984, Basili & Weiss 1984, Basili 1991, Basili 1992, Basili et al. 1994) to support the analysis of specification completeness and the collection of domain knowledge, specifically in support of black-box testing.

The input and output field definitions developed through GQASV specify the datatype, set type and size of each field in terms of minimum and maximum lengths, whether the field is mandatory or repeats, and the valid data set the program should accept and (ideally) the invalid data set it should reject. While the last two items are essential for selecting valid and invalid test cases, generic tests can be constructed using only datatype, set type and size, as they define the minimum amount of information required for proper application of the Atomic Rules approach. If the valid input data set is defined, then the datatype, set type and size of each field can be deduced and can act as an error checking mechanism. GQASV also facilitates the capture of domain knowledge that is utilised by testers during the specification process, allowing this information to be reused and shared with other novice and experienced testers when learning about the application domain of the program under test.

Each application of the technique results in one GQASV instance (i.e. one for each field specified). GQASV comprises the five steps, as follows[17].

1. As a goal, state that a particular field is going to be specified for the purpose of conducting effective black-box testing.
2. Consider the following questions:
    a. What is the field's datatype? [Integer | Real | Single Alpha | Multiple Alpha | Multiple Alphanumeric | Single Non-Alphanumeric | Multiple Non-Alphanumeric]
    b. What is the field's set type? [Range | List]

[17] The verification step of this process (i.e. step 5) is new; thus, it was not defined in the original publication of this method in (Murnane, Reed & Hall 2006).

c. For ranges, what are the minimum and maximum values? For Lists, what is the minimum and maximum length of valid data?

d. What valid data set should the program accept and what invalid data should it reject?

e. Is the field mandatory? [Yes | No]

f. Does the field repeat? [Yes | No]; if Yes, what are the minimum and maximum number of repetitions?

3. Seek and record answers to these questions by searching for domain knowledge in textbooks, standards, papers, websites or by consulting with domain experts (e.g. consultants working in the domain, business analysts, developers, clients and experienced testers). Each answer should state how it was obtained, as this can be useful for the testing, development or maintenance of the program being specified and for future software development in the same or similar domains. Furthermore, the correctness of domain knowledge obtained from websites should be verified by a domain expert before it is relied upon.

4. Specify the field using a formal notation (e.g. Backus-Naur Form), including valid and invalid data sets, if available.

5. Verify the correctness of the field definition by verifying its completeness with a domain expert (e.g. a client or business analyst who understands the client's needs) and (ideally) with a system developer, who can verify whether the implementation of this field is the same as its specification (see footnote in Section 3.10.2.1 for an example of the importance of this step).

Subsequently, a collection of Atomic Rules that match the characteristics of the newly defined field can be chosen using Systematic Method Tailoring. This step is analogous to the metrics selection step in GQM. However, a fundamental difference is that while question identification in GQM requires some undefined knowledge and expertise that is not apparent from the method, GQASV has a fixed set of questions that are always required when specifying any input or output field, making it possible to work with the approach when one has limited application domain knowledge.

An analysis of the literature suggests that while GQM and a number of other goal-oriented requirement engineering approaches have been used for requirement elicitation (e.g. see (Bonifati et al. 2001, Dubois, Yu & Petit 1998, Lamsweerde & Willemet 1998, Letier & Lamsweerde 2004, Sommerville et al. 1998)), this appears to be the first use of GQM for the analysis of specification completeness and the identification and documentation of domain knowledge, specifically in support of more effective black-box testing.

A demonstration of GQASV is provided in Section 3.10.3, along with a demonstration of Systematic Method Tailoring, which is introduced in the next section.

3.10.2 Systematic Method Tailoring (SMT)

Non-prescriptive approaches to black-box testing, such as Error Guessing and Exploratory Testing, compliment prescriptive black-box testing (Craig & Jaskiel 2002) and are seen by many practitioners as an important aspect of the testing process. Yet there are currently no techniques available to guide testers in the capture of test case design rules that are used during non-prescriptive testing, other than to create lists of

"error prone situations" (Myers 1979) that have not previously been documented by testers. As Jorgensen (1995) claims, "special values testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform… There are no guidelines, other than to use 'best engineering judgement'. As a result, special value testing is very dependent on the abilities of the tester". On the other hand, Craig and Jaskiel (2002) maintain that "Good exploratory testers often keep notes or checklists of tests that appear to be useful, to reuse on future releases. These 'notes' may (or may not) even look a lot like test scripts." For example, Test Matrices and Test Catalogues (see Chapter 2, Section 2.7). can be used to create repositories of "special values" (Kaner et al. 2001, Marick 1995). However, Test Matrices and Catalogues do not provide testers with any guidance on how to document the generic test case design rules that are used to derive special values. They also do not provide guidance on how to construct 'tailored' black-box testing methods from the test case design rules that are collected during Error Guessing and Exploratory Testing.

As an enhancement to the Atomic Rules approach, three new procedures for the Systematic Method Tailoring of black-box testing are proposed (Murnane, Reed & Hall 2006). These new approaches support the capture of domain-knowledge based test case design rules that are used during non-prescriptive testing and can be used to construct new black-box testing methods. They are:

1. selection-based tailoring (Section 3.10.2.1);
2. creation-based tailoring (Section 3.10.2.2);
3. creation-based tailoring via selection, using:
    3a. all combinations (Section 3.10.2.3);
    3b. paired combinations (Section 3.10.2.3); and
    3c. selective combinations (Section 3.10.2.3).

Although each approach is discussed independently, in practice, a combination of all three approaches may be used. A demonstration of SMT is provided in Section 3.10.3.

3.10.2.1 Selection-Based Tailoring

In *selection-based tailoring*, new black-box methods can be defined by 'selecting' existing Atomic Rules that match the Set Type (i.e. Range or List) of an Atomic Rule against the Set Type of valid data for each field under test, allowing new black-box testing methods to be defined that suit the unique testing needs of each program under test. In other words, a new Atomic Rule-based black-box testing method can be constructed by either selecting a set of Atomic Rules from the complete set of existing rules, or by selecting rules from a pre-existing Atomic Rule-based black-box testing method. This can be done at any level of the Atomic Rules schema. This is a bottom-up approach that is based on the approach taken in traditional black-box testing methods. For example, Myers (1979) provides different guidelines for selecting test data for range-based and list-based fields.

Consider the following example. Atomic Rules that could be selected for testing a range-based input field *<age> ::= [0 – 150]* include test selection Rules *BVA1: lower boundary – 1 selection* and *BVA6: upper boundary + selection* (see Appendix B for rule definitions). These could not be applied to a list-based field defined as *<colour> ::= [blue | green | red]*, as this field is a nominal set and it is impossible to predict what comes before or after its lower and upper boundaries. An interesting point to consider is, had the application programmer treated this field as a contiguous range-based field, where each colour is defined by a numerical range in the colour spectrum, e.g. *<colour_wavelength> ::= [450 - 495 nm, 495 - 570 nm, 620 - 750 nm]*, then Atomic Rules *BVA1* and *BVA6* would be applicable. This highlights the importance of the verification step in GQASV (see Section 3.10.1.1), as it ensures that each input and output field is specified and tested in an appropriate manner.

3.10.2.2 Creation-Based Tailoring

In *creation-based tailoring*, new Atomic Rules that have not been defined in existing black-box testing methods are defined. This is useful when testers suspect that a specific input may be effective for testing a particular field, and is similar to Error Guessing (Myers 1979). The benefit is that as each rule is defined using the Atomic Rules schema, it is described in a more prescriptive manner that makes the rule available for future reuse. Thus, a new Atomic Rule r_{i+1} that is not currently in the set of existing Rules R could be defined, $\{r_{i+1} : r_{i+1} \notin R\}$. Examples of new Atomic Rules that could be defined include the following:

1. Variations of ST Rules that have not been defined in existing literature, such as r_{i+1}*: first character selection*, which selects the first character of an input value (Table 3-23).
2. Rules that select specific input values, such as r_{i+2}*: select 0*, to test for divide by zero errors.
3. Rules to select sets of input values, such as r_{i+3}*: select all ASCII symbols*, to select a string of special characters from the ASCII table.
4. Rules that select Unicode characters (Aliprand et al. 2003), such as r_{i+4}*: Unicode U+00FC (ü) replacement*, which would be useful for performing Internationalisation Testing of programs that must support international languages.
5. Rules for testing programs with Graphical User Interfaces, for example:
    a. r_{i+5}*: maximum character selection*, which could add characters to a text field until no more characters will fit. This could be useful for testing for buffer overflow faults.
    b. r_{i+6}*: minimum – 1 list position selection*, which could attempt to select a record before the start of a record list to determine whether the program will run off the end of the record list. A similar rule could attempt to access a record beyond the end of a list.
6. Rules that select escape characters and keywords that are part of the programming or database query language of the system under test. For example, for HTML programs:
    a. r_{i+7}*: HTML tag character selection*, which could attempt to test with non-alphanumeric characters < and >, which are part of the HTML syntax.

Each new rule can be defined as an instance of the Atomic Rules schema (e.g. see Table 3-23).

Table 3-23: Creation of a new Atomic Rule defined through Systematic Method Tailoring.

Attribute	Values
Test Method	Syntax Testing
Number	r_{i+1}
Name	First Character Selection
Description	Select the first character of an input string
Source	N/A
Rule Type	DISR
Set Type	List or Range
Valid or Invalid	Invalid
Original Datatype	Multiple: Integer, Real, Single Alpha, Multiple Alpha, Multiple Alphanumeric, Single Non-Alphanumeric, Multiple Non-Alphanumeric
Test Datatype	Same as original
Test Data Length	1
# Fields Populated	1
# Tests Derived	0

3.10.2.3 Creation-Based Tailoring via Selection

In *creation-based tailoring via selection*, existing Atomic Rules are combined to create new Rules. There are three types of tailoring within this class:

1. all combinations;
2. paired combinations; and
3. selective combinations.

In each of these procedures, which are defined below, new Atomic Rules are defined by creating new instances of the Atomic Rules schema.

In *all combinations*, the set of all Atomic Rules $\{r_1, \ldots, r_n\}$ are combined, resulting in the n-ary Cartesian product $R_1 \times \ldots \times R_n = \{(r_1, \ldots, r_n) \mid r_1 \in R_1 \wedge \ldots \wedge r_n \in R_n\}$. However, this results in $\prod_{i=1}^{n} R_i$ combinations, where n is the number of existing Atomic Rules that have been identified for black-box testing. Thus, this may be only useful for experimentally locating combinations not found through other tailoring procedures.

In *paired combinations*, each Atomic Rule is paired with every other rule, resulting in the binary Cartesian product $R_i \times R_j = \{(r_m, r_n) \mid r_m \in R_i \wedge r_n \in R_j\}$, where each pairing creates a new rule. Some examples are:

1. r_{n+1}: *uppercase first item = BVA7: first list item selection* × *ST7: uppercase a lowercase letter*.

2. r_{n+2}*: smallest integer replacement = EP4: integer replacement × BVA2: lower boundary selection*.
3. r_{n+3}*: alphabetic letter Z or z replacement = EP6: single alpha replacement × BVA5: upper boundary selection*.

In *selective combinations*, rule amalgamation is based on a tester's 'intuition' that certain combinations may cause program failure (for a discussion on tester 'intuition,' see Chapter 6, Section 6.6.3). Again, this is similar to Error Guessing. For example, if a tester suspects that a program does not place an upper limit on the number of digits that can be input into a numerical field, a new rule r_{n+4}*: largest integer/real number replacement = EP4: integer replacement × EP5: real number replacement ×* r_{i+5}*: maximum character selection* could be defined.

Some combinations create Rules that already exist. In the current set of Atomic Rules for EP, BVA and ST, those combinations are:

1. *EP10: multiple alphanumeric replacement = EP4: integer replacement × EP7: multiple alpha replacement.*
2. *ST9: null all input = EP16: invalid test case constructor (minimised) × (EP11: null item replacement | BVA9: null item replacement).*

Also, some Rules are contradictory. EP replacement Rules *EP4* to *EP11* cannot be combined with *EP12: valid list selection* as the replacement Rules selects invalid data while *EP12* selects valid data. This is similar to the identification of contradictory test frames in CPM (Ostrand & Balcer 1988).

3.10.3 Demonstration of GQASV and SMT

As a preliminary proof of concept, GQASV and SMT are applied to an online Foreign Currency Exchange Calculator (Figure 3-16) (Murnane, Reed & Hall 2006). To assess their effectiveness, the results of applying a set of EP and BVA Atomic Rules to those selected by a new method derived by SMT are compared. To limit the scope of the example, only the "Foreign Currency" field of this program shall be tested. Settings for fields "*I wish to*," "*Select the foreign currency*" and "*Select the currency type*" are shown in Figure 3-16. As the program specification is not accessible, GQASV is applied to obtain a definition of the 'Foreign Currency' field.

1. ***Goal***: to specify the Foreign Currency field of the Foreign Exchange Calculator, in order to enable more effective black-box testing.
2. ***Questions***:
    a. What is the field's datatype?
    b. What is the field's set type?
    c. Ranges: minimum and maximum values; Lists: what is the minimum and maximum length of valid data?
    d. What valid data set should the program accept, and what invalid data should it reject?
    e. Is the field mandatory?

f. Does the field repeat? Minimum/maximum repetitions?

3. ***Answers***:

   a. *Datatype*: based on experience with international money transfers, acceptable datatypes are Integer and Real (i.e. non floating-point numbers).

   b. *Set type*: based on experience with banking systems, it is reasonable to assume that the interval of allowable values is continuous, thus set type is Range.

   c. *Minimum/maximum range values*: various searches were used to discover this. They are included here to give the reader an understanding of the process followed. First, a search of the St George Bank website for the term "international transfer" located on the page "Foreign Exchange Services" (St George Foreign Exchange Services), which included a telephone number. When called, the operator reported that there were no minimum or maximum limits placed on currency exchanges. However, through GUI-based application domain knowledge, it is known that unlimited input lengths can cause buffer overflow and conversion exceptions in internet-based applications. The next search determined the financial worth of the richest person on Earth, Bill Gates (at the time of the case study), which may be a sensible value to use. According the Forbes this was US$46.5 billion (Kroll & Gildman 2005). However, if the top twenty-five billionaires saved their money with the same bank, their total financial worth could be more sensible. According to Forbes, this was US$496.8 billion (Kroll & Gildman 2005). Taking this even further, one may consider the GDP of the largest economy in the world, the USA, US$10.8 trillion (Special Broadcasting Service 2003). These answers provide "application solution domain" (Reed 1990) data that is potentially sensible for defining this field. However, the maximum variable size of the programming language used, and combined implementation and runtime domain issues, could be considered. The client-side application was written in JavaScript (discovered by viewing the source code), which is capable of representing numbers in the range[18] $\pm 1.7976931348623157 \times 10^{308}$ (Flanagan 2002). For this application, this limit would be the maximum output value when converting to a particular currency or when an input is represented internally[19]. Thus, this figure needs to be divided by the largest possible exchange rate, which are available real-time on the Reserve Bank of Australia website (Reserve Bank 2005). Plausible values are 0.1 to 1000. Thus, sensible minimum and maximum range values could be $\pm 1.7976931348623157 \times 10^{305}$.

   d. *Valid data set*: as described in step c.

   e. *Is the field mandatory?* Yes.

   f. *Does the field repeat*? No.

4. ***Specify***: <foreign_currency> ::= [$-1.7976931348623157 \times 10^{305}$ $-1.7976931348623157 \times 10^{305}$]

5. ***Verify***: in a real-world software testing scenario, a domain expert (e.g. a client or business analyst) would be available to verity the completeness of this field definition. As this was carried out for demonstration purposes only and a domain expert was not available, this definition could not be verified.

[18] For the purposes of this discussion, we only consider exponents > 0.
[19] In Reed's KABASPP model (Reed 1990), this would represent knowledge gained from the development domain of the program.

Atomic Rules from EP and BVA can now be applied to generate test data (Table 3-24), followed by the selection of test cases by a tailored method (Table 3-25). Although the input field permits more numbers to be added, for test 15 (Table 3-25), an arbitrarily large number chosen to represent the maximum possible digits is 120,000. Although floating point representations are used throughout this discussion, when providing input to the program, an integer or fixed point decimal value containing 309 digits to the left of the decimal point was used. Thus, the inputs specified by test cases 8 to 13 (Table 3-25) contain 309 digits, the first section of which is the 17 digit mantissa of the resultant value. For example, in the case of test case 9, the number input to the program is 17976931348623157 followed by 297 "9"'s.

In fact, two different sets of values could have been used in this exercise, depending upon whether implementation "domain/run-time" issues (i.e. variable storage limits) or "application domain" issues (i.e. sensible values for maximum amounts) were being tested (Reed 1990). While it could be more sensible to derive test cases based on the latter, for the purposes of this proof of concept exercise, in this example implementation domain issues have been chosen as the focal point.

As Table 3-24 and Table 3-25 show, the tailored method detects a suspected fault with test case 14 (Figure 3-19) that is not detected by EP or BVA. Further examination revealed that inputting the string **"**<> followed by any other symbol and clicking the *Calculate* button causes those symbols to be printed to the right of the input field. Both EP and the tailored method detect that the program does not limit input data lengths (Table 3-24, test cases 1 and 2; Table 3, test case 15), causing a suspected buffer overflow (Figure 3-17). The resulting screen does not specify what was wrong with the input. BVA did not detect this as the exchange rate used was overestimated.

This example demonstrates that the use of GQASV and SMT can result in more effective black-box testing. It also shows how recording Atomic Rule numbers that have been applied against each input field during testing can simplify the process of assessing test set completeness (Table 3-24). Interestingly, a re-test of this scenario approximately three months after this initial proof of concept testing was carried out revealed that the fault detected by test cases 1, 2, and 15 had been repaired. However, it was not possible to determine whether it was fixed due to the testing that had been carried out during this proof of concept. The fault identified by test case 14 is still evident in the software as of 20th July 2008.

Table 3-24: Equivalence Partitioning and Boundary Value Analysis test cases for the *Foreign Currency* field of the St George Bank Foreign Currency Calculator.

#	Rule	Test Data	Result
1	EP1: < lower boundary selection	$-1.7976931348623157 \times 10^{305}$	Suspected buffer overflow (Figure 3-19)
2	EP2: > upper boundary selection	$+1.7976931348623157 \times 10^{305}$	Suspected buffer overflow (Figure 3-19)
3	EP3: lower to upper boundary selection	50000	Correct result output (Figure 3-16)
4	EP6: single alpha replacement	G	Input rejected, validation message shown (Figure 3-18)
5	EP7: multiple alphanumeric replacement	G55f	Input rejected, validation message shown (Figure 3-18)
6	EP8: single non-alphanumeric replacement	*	Input rejected, validation message shown (Figure 3-18)
7	EP11/BVA9: null item replacement		Input rejected, validation message shown (Figure 3-18)
8	BVA1: lower boundary – selection	$-1.7976931348623158 \times 10^{305} - 1$	Correct result output (Figure 3-16)
9	BVA2: lower boundary selection	$-1.7976931348623158 \times 10^{305}$	Correct result output (Figure 3-16)
10	BVA3: lower boundary + selection	$-1.7976931348623158 \times 10^{305} + 1$	Correct result output (Figure 3-16)
11	BVA4: upper boundary – selection	$1.7976931348623158 \times 10^{305} - 1$	Correct result output (Figure 3-16)
12	BVA5: upper boundary selection	$1.7976931348623158 \times 10^{305}$	Correct result output (Figure 3-16)
13	BVA6: upper boundary + selection	$1.7976931348623158 \times 10^{305} + 1$	Correct result output (Figure 3-16)

Table 3-25: Test cases of a tailored black-box method derived through SMT for the *Foreign Currency* field of the Foreign Currency Exchange Calculator (rules defined in Section 3.10.2).

#	Rule	Test Data	Result
14	r_{i+3}: select all ASCII symbols	!@#$%^&*()_+{}|:"<>?[]\;',./~`	Input rejected, validation message shown (Figure 3-18). Symbols output to the right of the Foreign Currency Field (Figure 3-19)
15	r_{n+4} largest integer/real replacement	120000 9's	Suspected buffer overflow (Figure 3-19)
16	r_{i+4}: Unicode U+00FC (ü) replacement	Ü	Input rejected, validation message shown (Figure 3-18)

Figure 3-15: St George Bank's online Foreign Currency Exchange Calculator (St George Calculator 2005).

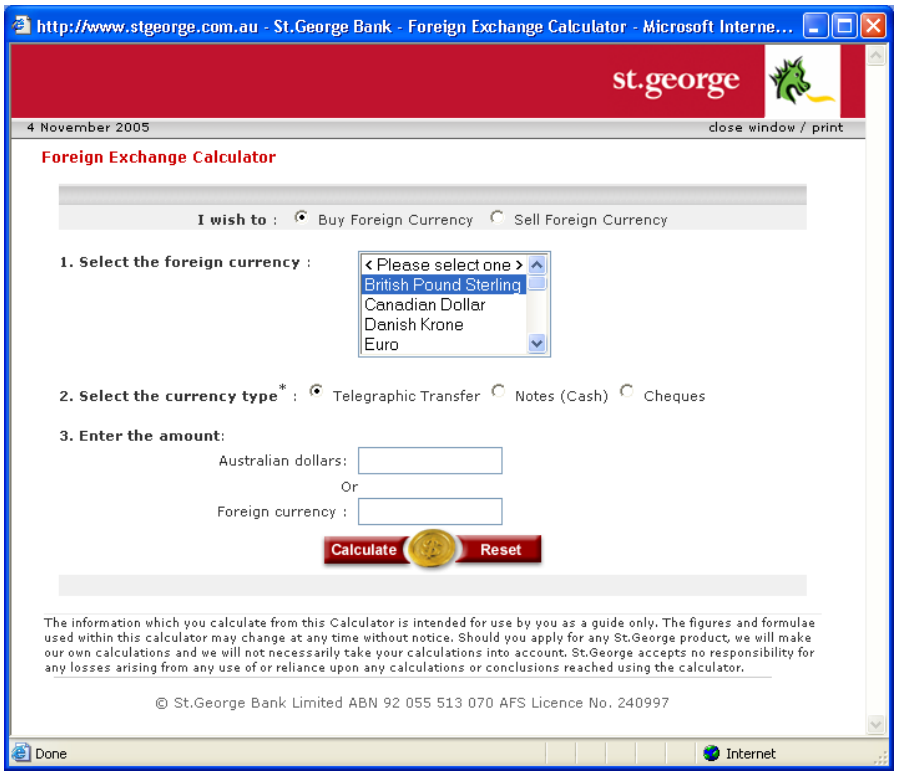


Figure 3-16: Result of executing the Foreign Currency Exchange Calculator with valid values.

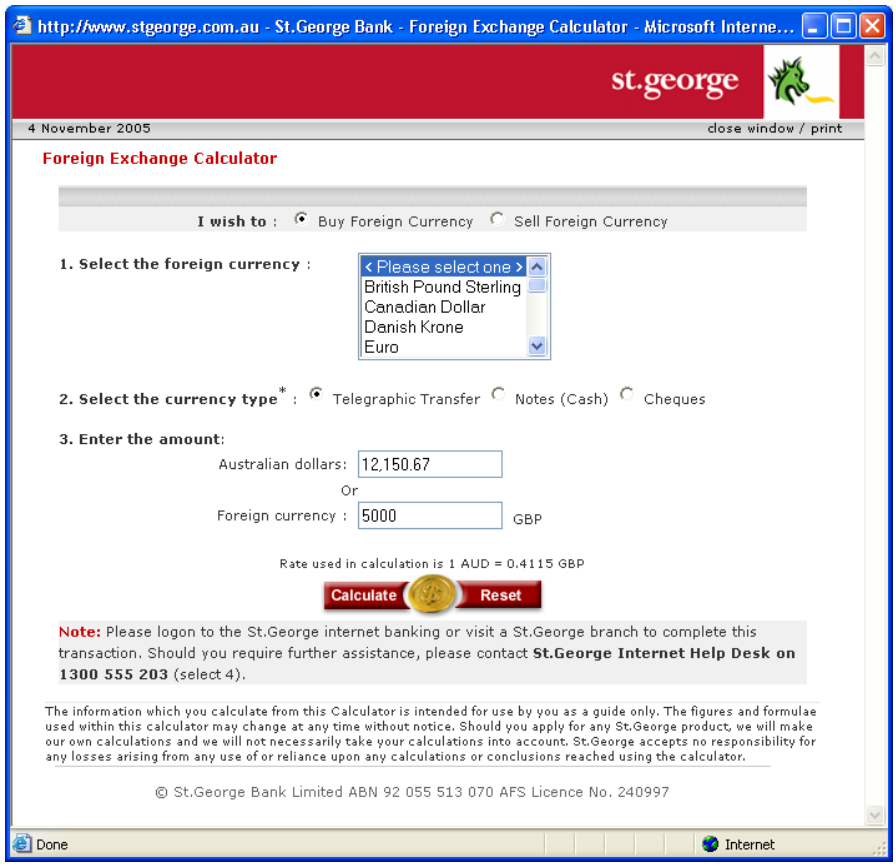


Figure 3-17: Result of testing the Foreign Currency Exchange Calculator with very large input, causing a suspected buffer overflow failure.

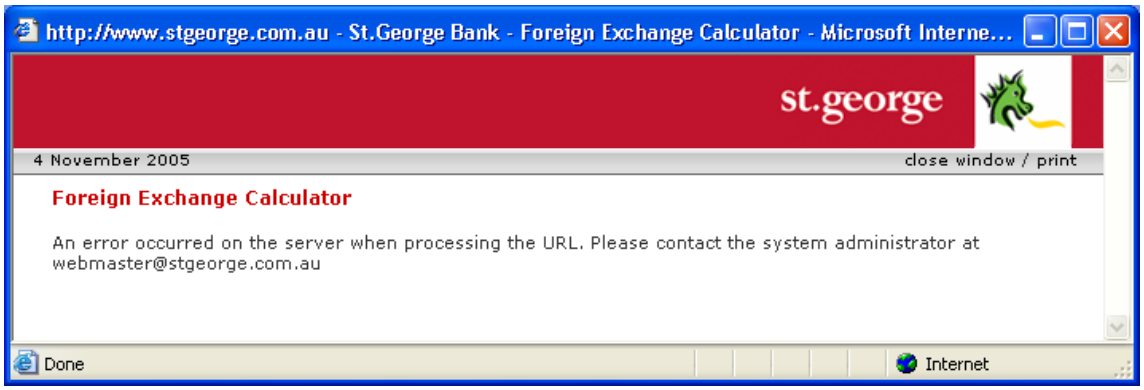
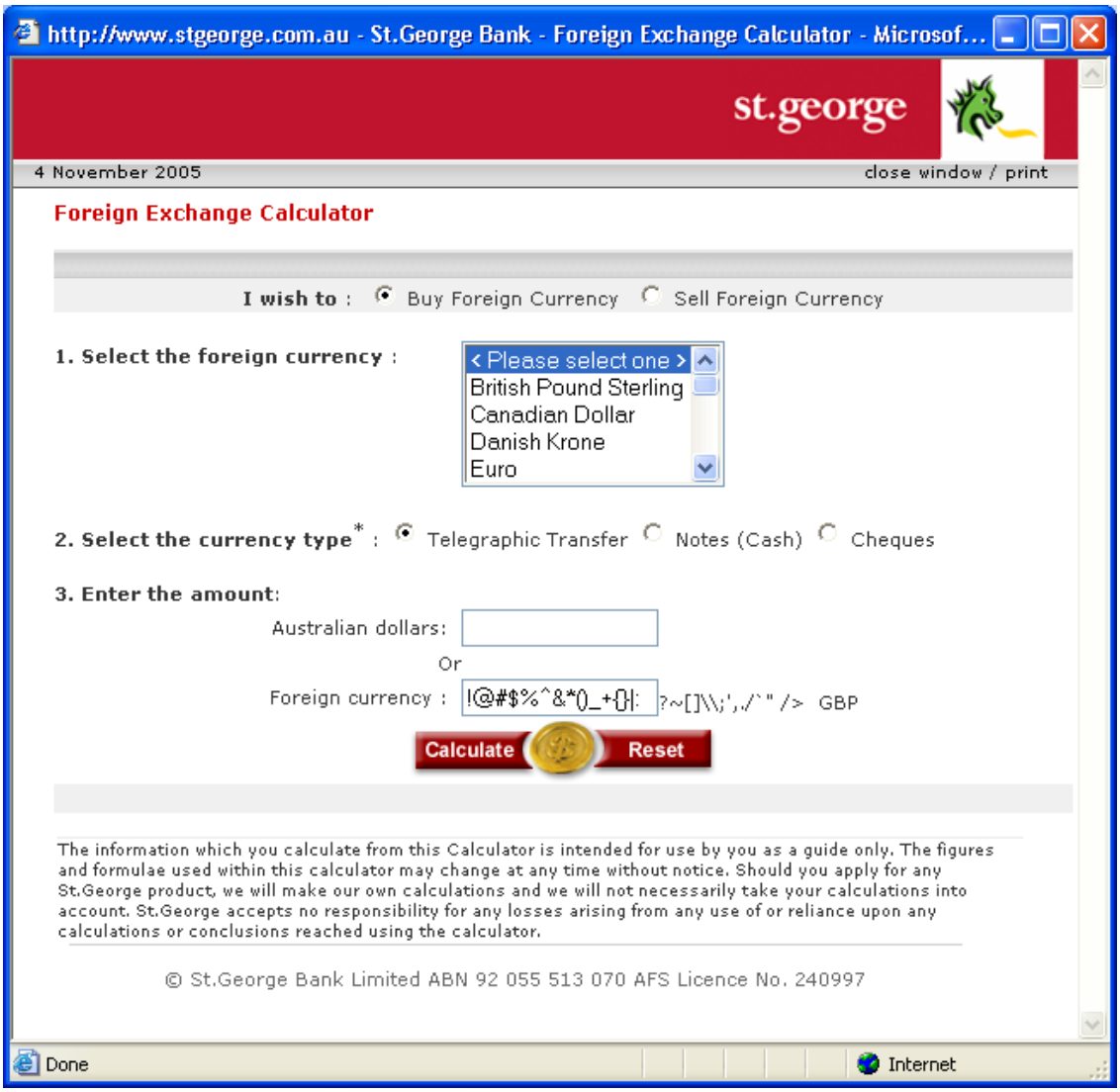


Figure 3-18: Validation message displayed when the Foreign Currency Exchange Calculator is tested with an invalid datatype.

Figure 3-19: Demonstration of symbols that are output to the right of the *Foreign Currency* field on the Foreign Currency Exchange Calculator, when test case 14 of Table 3-24 is applied.



3.11 Summary

The Atomic Rules approach, Goal/Question/Answer/Specify/Verify and Systematic Method Tailoring have been presented as solutions to seven problems with existing black-box testing methods (definition by exclusion, multiple versions, method overlap, notational and terminological differences and reliance on domain knowledge), with the ultimate aim of improving the usability and failure-detection effectiveness of the methods.

Definition by exclusion was resolved by defining explicit datatypes that delineate the scope of valid and invalid equivalence classes that are selected by each Atomic Rule. This also partially resolved ***reliance on domain knowledge***, by defining Atomic Rules to a level of detail that facilitates the development of effective and predictable test sets, regardless of each tester's domain knowledge and experience. Systematic Method Tailoring (SMT) further resolved ***reliance on domain knowledge*** by facilitating the definition of new Atomic Rules and new black-box testing methods through the capture of domain knowledge that is

used by professional testers during non-prescriptive testing. ***Reliance on domain knowledge*** was further resolved by GQASV, which supports testers in developing precise specifications of input program input and output fields, enabling more effective black-box testing. Domain knowledge utilised during the specification process is also captured by GQASV, allowing it to be reused and shared with other testers.

Notational and terminological differences were resolved by creating a characterisation schema that defines the characteristics of each black-box test case design rule in a uniform way and by developing a four-step test case design process that is common to all black-box testing methods. ***Method overlap*** was resolved by identifying Atomic Rules that overlap, both within and between different black-box testing methods. ***Multiple versions*** of the same method were resolved by developing one set of Atomic Rules that cover the test case design rules of all published versions of that method. This makes the methods ***easier to audit***, since it provides one set of Atomic Rules that cover all versions of each method. The process of auditing the completeness of black-box test sets was demonstrated by creating a Test Matrix to track the set of Atomic Rules that were (or were not) applied to a program.

The prescriptive nature of the Atomic Rules approach, GQASV and SMT makes black-box test case generation ***easier to automate***. A prototype called the Atomic Rules Testing Tool, which automates these concepts, is presented in Chapter 4. Currently, ARTT can automatically generate black-box test data from specifications that are input through a graphical user interface, record domain knowledge gained through GQASV and define new Atomic Rules through SMT. The ultimate aim of ARTT is to make black-box test case design even more efficient and precise.

An additional benefit of the Atomic Rules approach, which was presented in this chapter, is that Atomic Rules from EP, BVA and ST can aid test data selection for methods like State Transition Testing, Use Case Testing and the Category Partition Method.

Two limitation of this approach were presented in this chapter. Currently, the Atomic Rules approach cannot be used to test field dependencies, where the value chosen for one field of a test case depends on the value chosen for another. Atomic Rules also cannot currently be used for selecting input test data values for testing output field partitions. On the other hand, future research in this area may enable support of these two aspects of black-box test case design (see Chapter 7).

Chapter 4

Automating the Atomic Rules Approach

"There are only two industries that refer to their customers as 'users'."

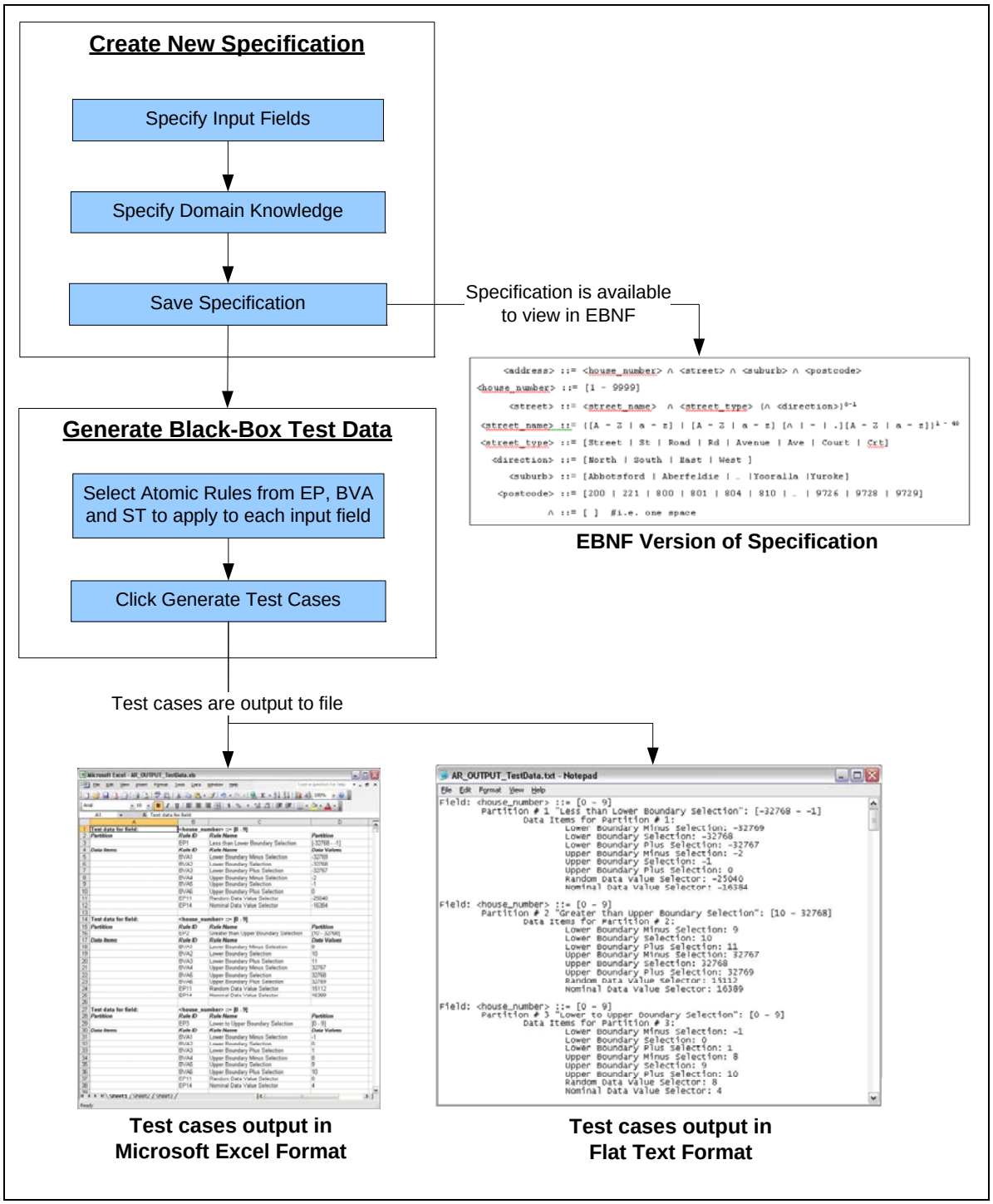
Edward Tufte

4.1 Overview

The Atomic Rules approach, GQASV and SMT were developed to improve the usability and failure-detection effectiveness of black-box testing. Since these approaches exhibit a high degree of regularity coupled with considerable fine-grained detail, it was considered that automation was both possible and desirable. It was also felt that an appropriate tool could support and record the decision-making process, providing reusable records of, for example, the actual knowledge sources used and outcomes obtained when applying GQASV. While it was not originally intended, such a tool could be considered to be an application domain specific Design Reasoning Recording (DRR) system (Potts & Bruns 1988), albeit with some limitations in the DRR sense (Potts & Bruns (1988) is cited as a pioneering and classic description of this class of system).

In this chapter, a prototype called the Atomic Rules Testing Tool, which provides automation support to the Atomic Rules approach, GQASV and SMT, is presented. The aim of ARTT is to improve the efficiency and accuracy to black-box test data generation. ARTT automates test data generation as follows (Figure 4-1). A graphical user interface allows the user to create specifications of program input fields. The user can then choose to apply a subset of Atomic Rules from EP, BVA and ST to each input field. ARTT then applies the chosen set of Atomic Rules to the specified input fields to generate black-box test data values, based on the Atomic Rule application order prescribed in the four-step test case design process (see Chapter 3, Section 3.2.1). ARTT enables the capture of domain knowledge that is collected during the specification process through GQASV and, in addition, the definition of new Atomic Rules through creation-based SMT. ARTT supports any level of testing, although it is particularly useful during Unit, Integration and System Testing, during which input field validation testing is typically performed.

Table 4-1: Process of creating a specification and generating test data in the Atomic Rules Testing Tool.



ARTT currently implements Data-Set Selection Rules, Data-Item Selection Rules and Data-Item Manipulation Rules. Future work will include implementation of Test Case Construction Rules (see Section 4.10), as well as the automatic generation of program source code for input data validation in various programming languages. For example, if an input field should only accept integers within a certain range, then ARTT could apply a set of Atomic Rules to automatically generate source code that accepts integers within that range and rejects all other inputs. This would reduce the need for rigorous input/output testing.

This chapter is structured as follows. An overview of the screens and navigation in ARTT is provided in Section 4.2, while the architecture is discussed in Section 4.3. The approach to test data generation is described in Section 4.2. Specification creation in ARTT is discussed in Section 4.4.1. The application of Atomic Rules to specifications is covered in Section 4.4.2. This is followed by an example of test data generation in Section 4.4.3. The implementation of GQASV and SMT are covered in Sections 4.5 and 4.6. The specification notation used in ARTT is described in Section 4.7. Benefits and limitations are discussed in Sections 4.8 and 4.9 and future improvements in Section 4.10. A chapter summary is given in Section 4.11. Detailed functional specifications for ARTT are provided in Appendix F.

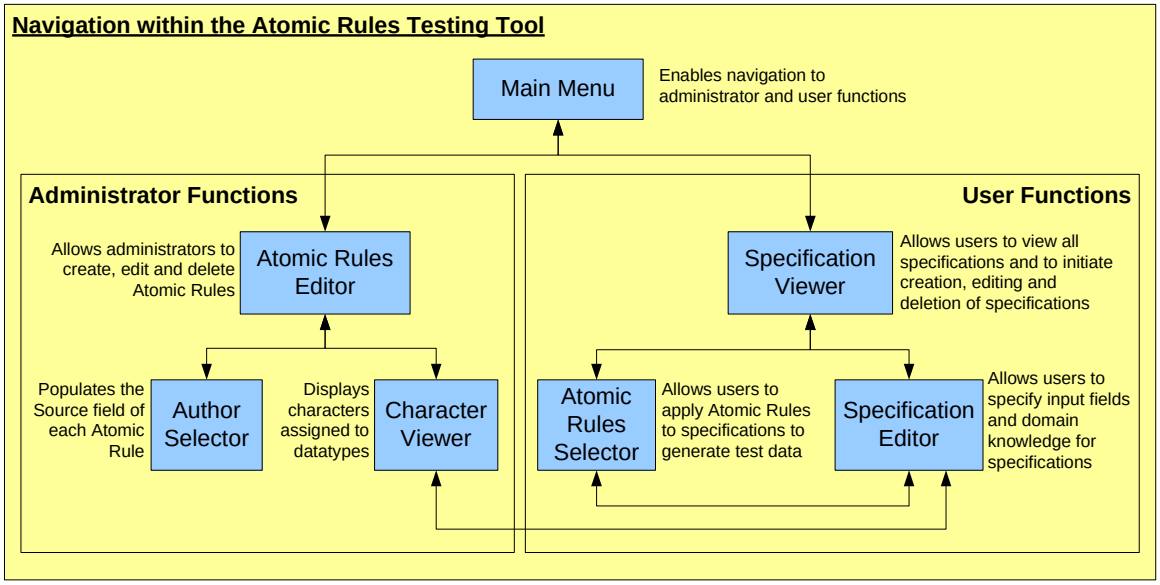
4.2 Screens and Navigation

ARTT functionality is divided into two main functional areas: administrator and user functions. These are implemented within seven screens, as follows .

1. **Main Menu:** this screen supports navigation to the user and administration functions (see Figure 4-1 and Appendix F, Section F.4.1).
2. **Atomic Rules Editor:** this allows administrators to create, edit and delete Atomic Rules (see Appendix F, Section F.4.2).
3. **Author Selector:** this allows administrators to populate the Source field of Atomic Rules (see Section 3.2.2), and is accessible from the Atomic Rules Editor (see Appendix F, Section F.4.3).
4. **Character Viewer:** this screen allows administrators and users to view the individual characters that are included within the datatypes (e.g. Integer, Real, Alpha) that are supported by ARTT. This screen can be accessed from the Atomic Rules Editor to view the contents of the Original Datatype and Test Datatype of each Atomic Rule, and from the Specification Editor to view the contents of input fields that are specified by datatype (see Appendix F, Section F.4.4).
5. **Specification Viewer:** this allows users to view all specifications that have been created, and to initiate the creation, editing and deletion of specifications (see Appendix F, Section F.4.5).
6. **Specification Editor:** this allows users to create specifications, by creating, editing and deleting input fields, assigning domain knowledge to each input field and attaching files to the specification. Users can also view an automatically generated EBNF representation of their specification on this screen (see Appendix F, Section F.4.6).
7. **Atomic Rules Selector:** this allows users to apply a set of Atomic Rules from EP, BVA and ST to a specification, to automatically generate black-box test data (see Appendix F, Section F.4.7).

Although there are user and administrator functions within ARTT, there is no login screen, since data security is not a risk.

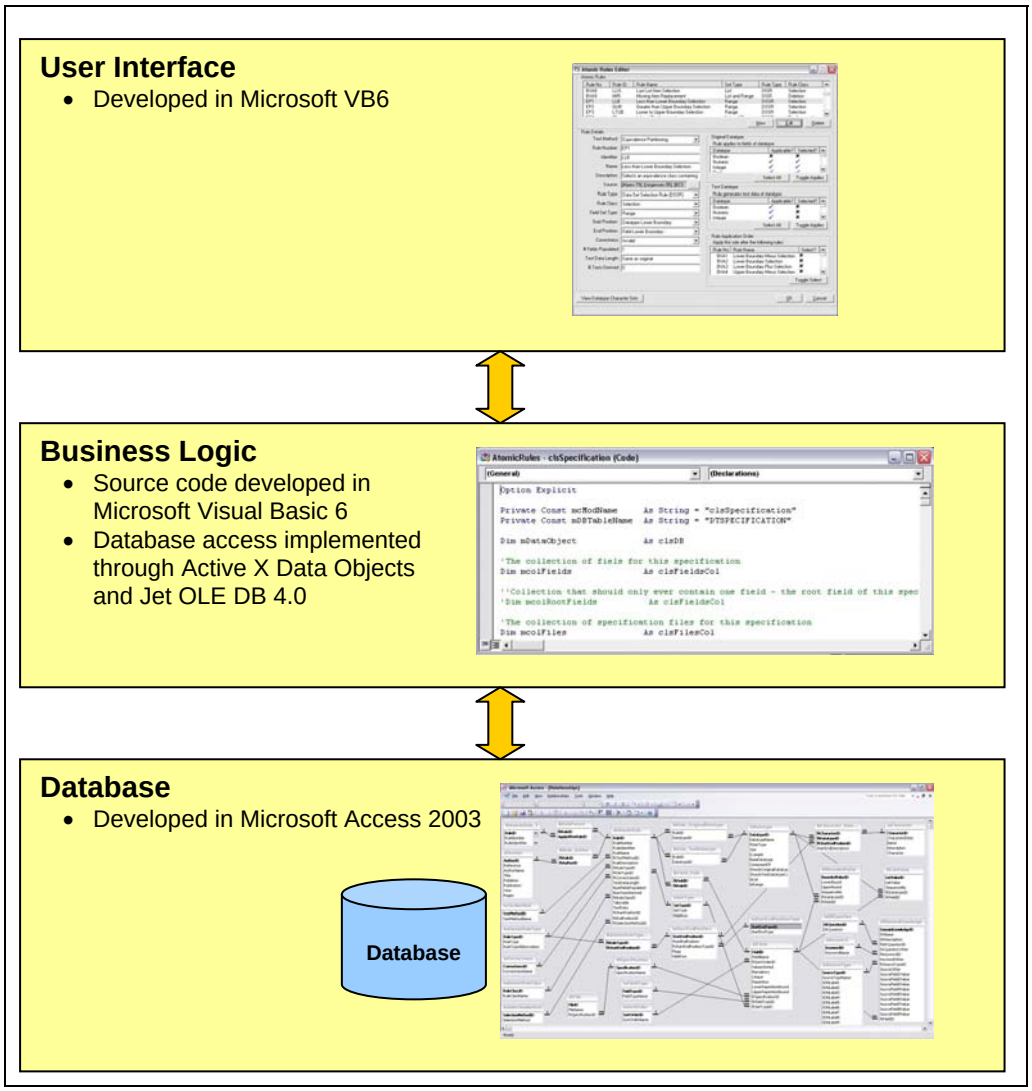
Figure 4-1: Screens and navigation within the Atomic Rules Testing Tool.



4.3 Architecture

ARTT was developed in Microsoft Visual Basic 6.0 (VB6) in a Windows XP environment. It has a three-tiered architecture with the GUI and business logic being implemented in VB6 and the database developed in Microsoft Access 2003 (Figure 4-2). The application tier communicates with the database via an ODBC Data Source that utilises ActivieX Data Objects and Jet OLE DB 4.0.

Figure 4-2: High-level architecture of the Atomic Rules Testing Tool.



4.4 Test Data Generation

The process of generating black-box test data values through ARTT is as follows.

1. A user specifies the characteristics of program input fields through the *Specification Editor* screen (see Section 4.4.1).
2. The user then selects a set of Atomic Rules to automatically apply to the specification through the *Atomic Rules Selector* screen (see Section 4.4.2).
3. ARTT applies the chosen set of DSSRs, DISRs and DIMRs against the specified input fields to automatically generate equivalence classes, test data values and manipulated test data values. This is done by mapping the *Set Type* and *Datatype* of each input field to the *Set Type* and *Original Datatype* of each Atomic Rule. Equivalence classes and test data values are output to file in both plain text and Microsoft Excel format.

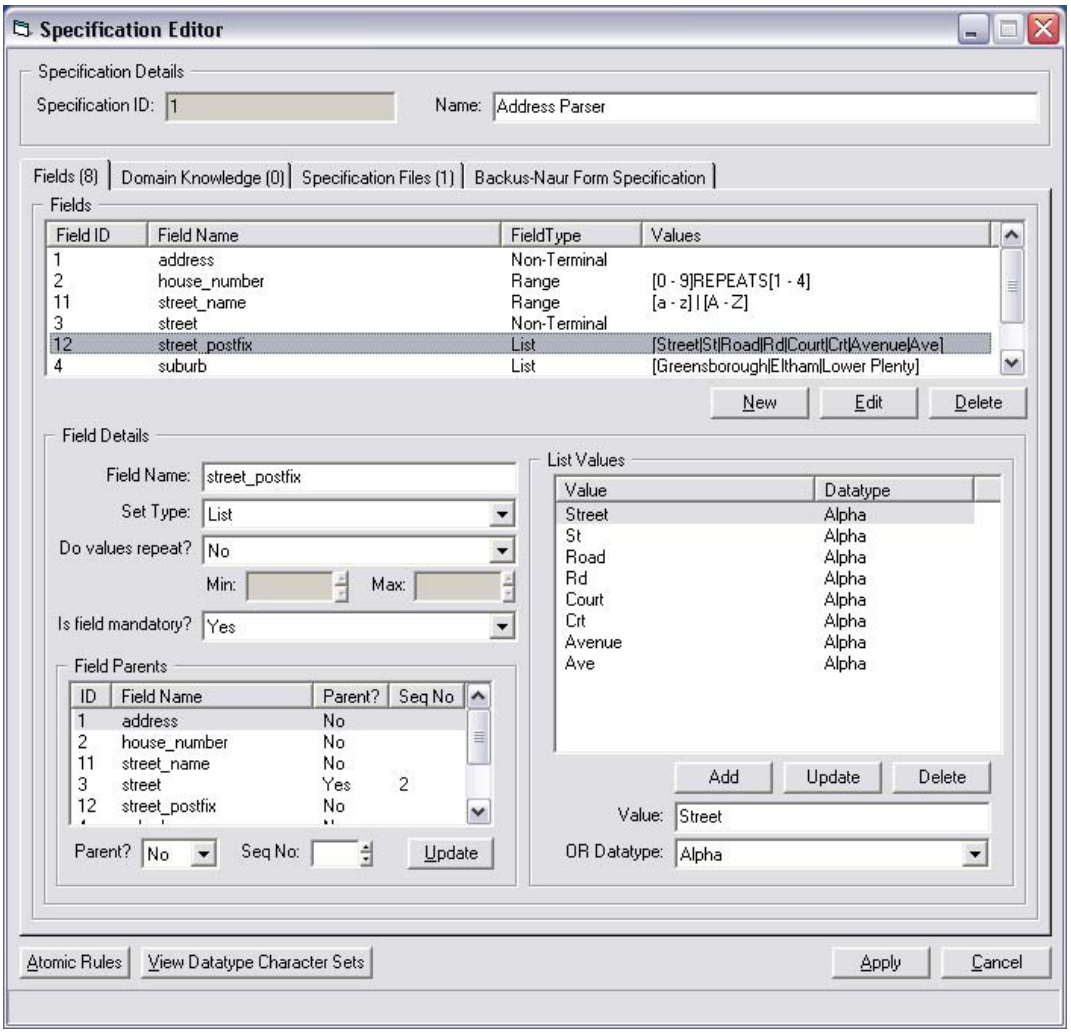
This process is described in detail in the subsections below.

4.4.1 Specification Creation

The input fields that test data is going to be generated from can be specified through the Specification Editor (Figure 4-3 and Appendix F Section F.3.6). The following information is recorded for each field.

1. Field name
2. Set type (Range or List)
3. For *list-based fields*, the individual values that are stored in the list or a list of datatypes that are accepted by the field as valid. For *range-based fields*, the minimum and maximum values of contiguous data that is allowed in the field or a contiguous datatype
4. The minimum and maximum number of times the field repeats (i.e. zero or more)
5. Whether the field is mandatory or optional
6. Parent fields of the field, which allows the hierarchy of the specification to be defined (e.g. see Figure 4-4)

Figure 4-3: Specifying the input fields of a program in the Atomic Rules Testing Tool.



For example, consider the specification for the Address Parser program provided in Chapter 3 (Figure 3-9). For the list field *<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]*, the following information would be recorded (e.g. see Appendix F, Section 3.6.1).

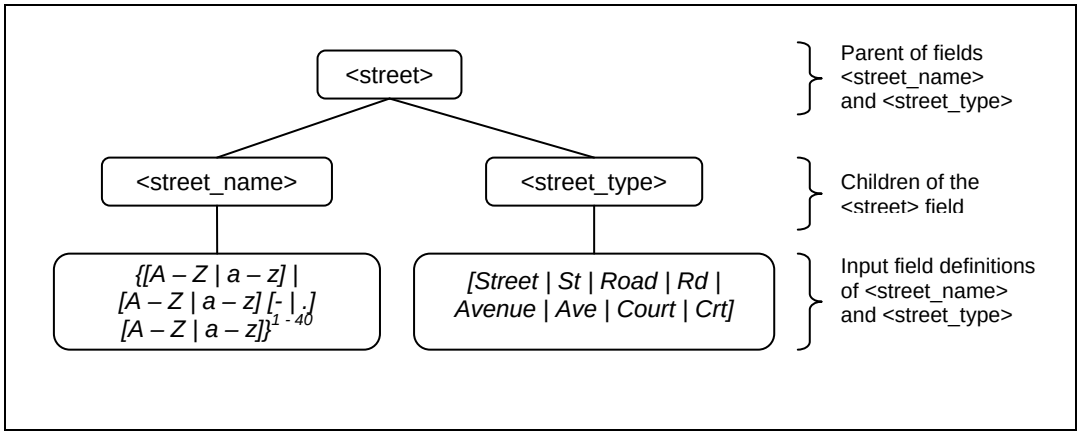
1. Field name: street_type
2. Set type: list
3. Individual values stored in the list: Street, St, Road, Rd, Avenue, Ave, Court, Crt
4. Minimum and maximum number of times the field repeats: zero (field does not repeat)
5. Mandatory or optional: mandatory
6. Parent field: street_name (see Figure 4-4)

Similarly, for the range-based field *<house_number> ::= [1 – 9999]*, the following information would be recorded (e.g. see Appendix F, Section 3.6.2).

1. Field name: house_number
2. Set type: range
3. Contiguous data allowed in the field: 1 – 9999
4. Minimum and maximum number of times the field repeats: zero (field does not repeat)
5. Mandatory or optional: mandatory
6. Parent field: address

Thus, the Specification Editor screen allows the individual fields of a specification to be defined in a systematic format and also allows the hierarchy of the specification to be defined.

Figure 4-4: Abstract Syntax Tree depicting example parent/child relationships in a (hierarchical) Address Parser specification.



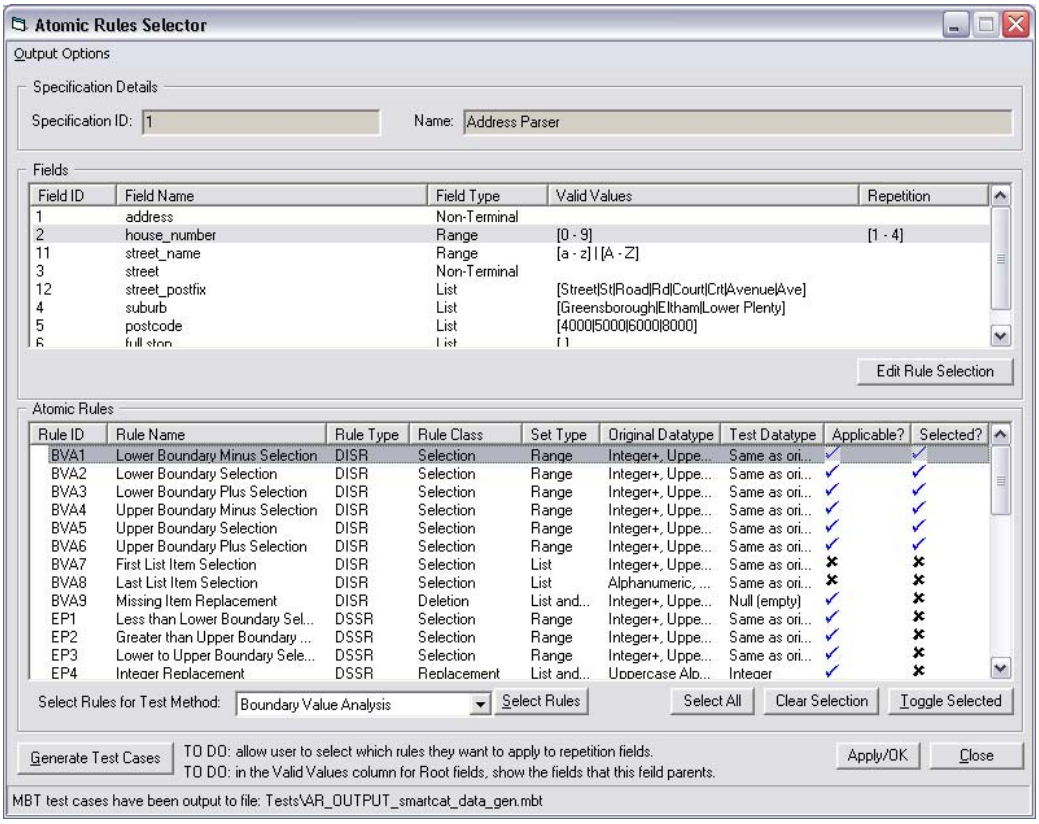
4.4.2 Atomic Rules Selection

Once the program input fields have been specified, the Atomic Rules Selector allows the user to select a set of Atomic Rules that will be applied to each input field (Figure 4-5 and Appendix F Section F.3.7).

ARTT only allows a user to apply Atomic Rules that match the *Set Type* and *Datatype* of the input field. For example, if test data was being generated for a field defined as *<house_number> ::= [0 – 9]*$^{1-4}$ (which is an alternate definition of the house number field), then rules *BVA1* to *BVA6* could be applied since they are applicable to range-based fields and contiguous datatypes, whereas *BVA8* and *BVA9* could not be applied as these only apply to list-base fields (e.g. see Figure 4-5, 'Applicable' column).

From the list of 'applicable' rules, the user can choose to apply a set of Atomic Rules EP, BVA, ST or can choose to apply all applicable rules from these methods (e.g. see Figure 4-5, 'Selected' column).

Figure 4-5: Selecting Atomic Rules to apply to an example specification.



4.4.3 Test Data Generation

Once the user has selected a set of Atomic Rules to apply to each input field, clicking the 'Generate Test Cases' button on the Atomic Rules Selector screen (see Figure 4-5) activates the automated test data generation process. ARTT generates test data by applying each Atomic Rule to each field of a specification, according to the four-step test selection process, which is implemented in four main algorithms:

1. Select Partition
2. Select Data Item
3. Manipulate Data Item
4. Construct Test Cases

Currently, only the first three functions of this process are implemented. Future work on ARTT will include development of algorithms that construct test cases (see section 4.10).

Consider a field *<house_number> ::= [1 – 9999]*. If the user applies Data-Set Selection Rules *EP1*, *EP2*, *EP3*, and *EP5* to this field, ARTT would automatically generate the following equivalence classes.

1. EP1: [-32768 – 0]
2. EP2: [10000 – 32767]
3. EP3: [1 – 9999]
4. EP5: real numbers in the range [-32768.00 – 32767.00]

Further, if the user applies Data-Item Selection Rules *EP13* and *BVA1* to *BVA6* to this field, the following test data values would be automatically generated from these equivalence classes. One of the limitations of ARTT is it does not yet automatically identify test data values that overlap (e.g. see 1f and 3b in the list below). This is planned as a future feature of this tool.

1. EP1: [-32768 – 0]
    a. EP13: select a random value, such as -25040
    b. BVA1: -32767
    c. BVA2: -32768
    d. BVA3: -32769
    e. BVA4: -1
    f. BVA5: 0
    g. BVA6: 1
2. EP2: [10000 – 32767]
    a. EP13: select a random value, such as 10936
    b. BVA4: 9999
    c. BVA5: 10000
    d. BVA6: 10001
    e. BVA1: 32766
    f. BVA2: 32767
    g. BVA3: 32768
3. EP3: [1 – 9999]

   a. EP13: select a random value, such as 8723
   b. BVA1: 0 (already covered by 1g)
   c. BVA2: 1 (already covered by 1h)
   d. BVA3: 2
   e. BVA4: 9998
   f. BVA5: 9999 (already covered by 1c)
   g. BVA6: 10000 (already covered by 1d)
4. EP5: real numbers in the range [-32768.00 – 32767.00]
   a. EP13: select a random value, such as 18732.97
   b. BVA1: -32767.99
   c. BVA2: -32768.00
   d. BVA3: -32768.01
   e. BVA4: 32766.99
   f. BVA5: 32767.00
   g. BVA6: 32767.01

Then, if the user applies the Data-Item Manipulation Rule *ST2: Replace Last Character* to this field, the following 'mutated' test data values are generated from the list of test data values above. In each of the mutated data items below, the last character is replaced by another character that is randomly chosen from the ASCII table. This produces many test data values that overlap (e.g. items 1c and 1d belong to the same equivalence class). Future implementations of ARTT will remove these types of redundant test data values.

1. EP1: [-32768 – 0]
   a. EP13: select a random value, such as -2504D
   b. BVA1: -3276k
   c. BVA2: -3276'
   d. BVA3: -3276[
   e. BVA4: -A
   f. BVA5: @
   g. BVA6: !
2. EP2: [10000 – 32767]
   a. EP13: select a random value, such as 1093p
   b. BVA4: 999.
   c. BVA5: 1000m
   d. BVA6: 1000Q
   e. BVA1: 3276'

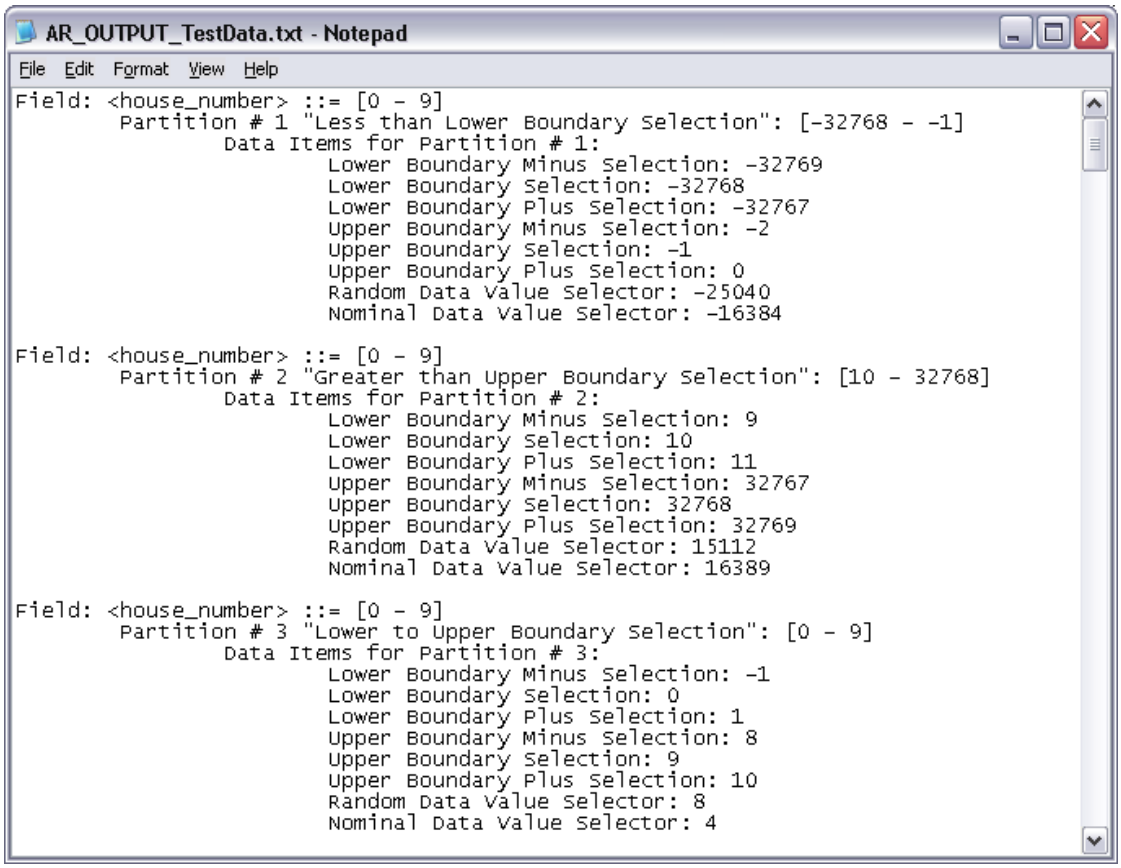
f. BVA2: 3276b

g. BVA3: 3276R

3. EP3: [1 – 9999]
   a. EP13: select a random value, such as 8728
   b. BVA1: #
   c. BVA2: +
   d. BVA3: ;
   e. BVA4: 999S
   f. BVA5: 999v
   g. BVA6: 1000l
4. EP5: real numbers in the range [-32768.00 – 32767.00]
   a. EP13: select a random value, such as 18732.9M
   b. BVA1: -32767.9(
   c. BVA2: -32768.0$
   d. BVA3: -32768.0j
   e. BVA4: 32766.9.
   f. BVA5: 32767.0]
   g. BVA6: 32767.0+

In a matter of seconds, ARTT will output the set of equivalence classes, test data values and manipulated test data values that are generated to a text file (Figure 4-6) and to a Microsoft Excel spreadsheet (Figure 4-7). This completes the process of test data generation. Some names and numbers of Atomic Rule have changed since the prototype of the Atomic Rules Testing was first developed. For example, *EP11: Random Test Data Selector* and *EP14: Nominal Value Selector* are illustrated in this figure but are now referred to as *EP13: Random Data Value Selector* and *EP17: Nominal Data Value Selector*. Thus, the names and numbers of some of rules that appear in the following two diagrams differ from those defined in the Appendix.

Figure 4-6: Example of test data values output to a text file by the Atomic Rules Testing Tool.



When test data is output to Microsoft Excel spreadsheets, it includes the Rule Identifiers (Rule ID) that were used in the test data generation process (Figure 4-7).

Figure 4-7: Example of test data values output to a Microsoft Excel spreadsheet by the Atomic Rules Testing Tool.

Microsoft Excel - AR_OUTPUT_TestData.xls

File Edit View Insert Format Tools Data Window Help

A1 Test data for field:

	A	B	C	D
1	**Test data for field:**	**<house_number> ::= [0 - 9]**		
2	***Partition***	***Rule ID***	***Rule Name***	***Partition***
3		EP1	Less than Lower Boundary Selection	[-32768 - -1]
4	***Data Items***	***Rule ID***	***Rule Name***	***Data Values***
5		BVA1	Lower Boundary Minus Selection	-32769
6		BVA2	Lower Boundary Selection	-32768
7		BVA3	Lower Boundary Plus Selection	-32767
8		BVA4	Upper Boundary Minus Selection	-2
9		BVA5	Upper Boundary Selection	-1
10		BVA6	Upper Boundary Plus Selection	0
11		EP11	Random Data Value Selector	-25040
12		EP14	Nominal Data Value Selector	-16384
13				
14	**Test data for field:**	**<house_number> ::= [0 - 9]**		
15	***Partition***	***Rule ID***	***Rule Name***	***Partition***
16		EP2	Greater than Upper Boundary Selection	[10 - 32768]
17	***Data Items***	***Rule ID***	***Rule Name***	***Data Values***
18		BVA1	Lower Boundary Minus Selection	9
19		BVA2	Lower Boundary Selection	10
20		BVA3	Lower Boundary Plus Selection	11
21		BVA4	Upper Boundary Minus Selection	32767
22		BVA5	Upper Boundary Selection	32768
23		BVA6	Upper Boundary Plus Selection	32769
24		EP11	Random Data Value Selector	15112
25		EP14	Nominal Data Value Selector	16389
26				
27	**Test data for field:**	**<house_number> ::= [0 - 9]**		
28	***Partition***	***Rule ID***	***Rule Name***	***Partition***
29		EP3	Lower to Upper Boundary Selection	[0 - 9]
30	***Data Items***	***Rule ID***	***Rule Name***	***Data Values***
31		BVA1	Lower Boundary Minus Selection	-1
32		BVA2	Lower Boundary Selection	0
33		BVA3	Lower Boundary Plus Selection	1
34		BVA4	Upper Boundary Minus Selection	8
35		BVA5	Upper Boundary Selection	9
36		BVA6	Upper Boundary Plus Selection	10
37		EP11	Random Data Value Selector	8
38		EP14	Nominal Data Value Selector	4

Sheet1 Sheet2 Sheet3

Ready

4.5 Implementation of Goal/Question/Answer/Specify/Verify

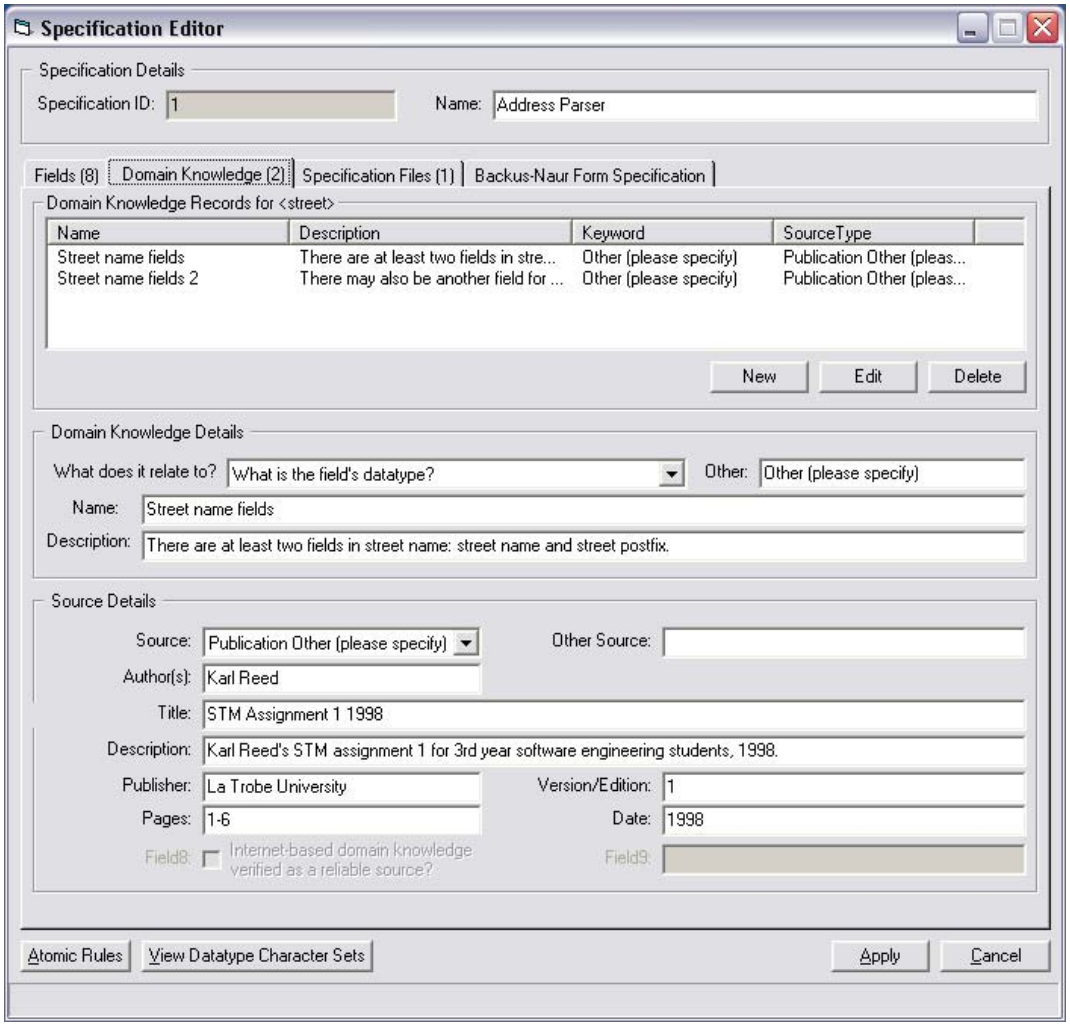
ARTT also provides support for GQASV. When the user enters the specification for each input field through the Specification Editor screen (see Section 4.4.1), they can also choose to record the source of any domain knowledge they may have consulted during the specification process (Figure 4-8). For example, if the user consulted a programming textbook for the definition of the datatype of an input field, then they could record the name of that textbook against the field. In the current version, only the name of the knowledge source can be recorded. More precise identification of the source, such as page number and a navigation tool for online sources, will be included in future developments of the tool.

ARTT collects the following information on the source of the domain knowledge.

1. A question asking the user “What does [the domain-knowledge] relates to?” Predefined questions the user can choose from are as follows.
    a. What is the field's datatype?
    b. What is the field's set type?
    c. For Set Type = Range: what are min and max values?
    d. For Set Type = List: what are min and max valid data lengths?
    e. Is the field mandatory?
    f. Does the field repeat?
    g. If field repeats, what are min and max repetitions?
    h. Other (please specify). If the user chooses other, then they can specify their own question in the ‘Other’ field.
2. A Name and Description to uniquely identify the domain knowledge. This will appear in the Domain Knowledge list when the field is saved (see Figure 4-8).
3. The Source (type) of the domain knowledge, which can be chosen from the following list:
    a. Personal Knowledge
    b. Personal Experience
    c. Book
    d. Textbook
    e. Standard
    f. Conference Paper
    g. Journal Paper
    h. White Paper
    i. Magazine
    j. Newspaper Article
    k. Technical Report
    l. Web Site
    m. Domain Expert
    n. Source Code
    o. Publication Other (please specify)
    p. Other (please specify). If the user chooses other, then they can specify their own question in the ‘Other’ field.
4. The user is then able to enter the details of the source. The fields that appear in the ‘Source’ frame (see Figure 4-8) depend on the particular ‘source type’ that was chosen in the previous step. The fields that appear can include the author, title, description and publisher of the source (see Appendix F Section F3.6.3 for more information).

This allows the user to record the source location of all information they utilised when they were specifying the input fields of the program under test.

Figure 4-8: Recording domain-knowledge information gained for each input field being specified.



4.6 Implementation of Systematic Method Tailoring

ARTT automates two forms of SMT: selection-based tailoring and creation-based tailoring. *Selection-based tailoring*, which was demonstrated in the previous section, allows users to apply Atomic Rules from various different black-box testing methods in combination to automatically generate 'novel' test data.

ARTT enables *creation-based tailoring* via the definition of new Atomic Rules through the Atomic Rules Editor (Figure 4-9). On this screen the user can view the characteristics of each Atomic Rule and define new rules. The screen includes all attributes of the Atomic Rules schema, as well as a number of additional fields that are required for automation of this approach. These are as follows (attributes that correspond to fields of the Atomic Rules schema are marked with an asterisk *).

1. **Test Method***. Name of the test method the rule is defined for. Options are Equivalence Partitioning, Boundary Value Analysis and Syntax Testing.

2. **Rule Number***. Abbreviation of the test method name followed by an incremental number.
3. **Identifier***. Abbreviation of the Name field.
4. **Name***. The name of the Atomic Rule.
5. **Description***. A brief description of what the rule does.
6. **Source***. The reference from which the rule was derived (this can be selected from the Author Selector screen; see Appendix F, Section F.3.3).
7. **Rule Type***. Corresponds to the step of the four-step test case design process that this rule implements. Options are Data-Set Selection Rule, Data-Item Selection Rule, Data-Item Manipulation Rule and Test Case Construction Rule.
8. **Rule Class**. There are five classes of rules implemented in ARTT, as follows.
    a. *Selection* rules select equivalence classes (e.g. *EP1* to *EP3*) and test data values (e.g. *BVA1*).
    b. *Insertion* rules add individual test data values into equivalence classes and to other test data values (e.g. *ST3* and *ST6*).
    c. *Deletion* rules remove values from equivalence classes and test data values (e.g. *ST1* and *ST4*)
    d. *Replacement* rules replace equivalence classes with invalid partitions (e.g. *EP4* to *EP11*) and values within test data items with invalid values (e.g. *ST2* and *ST5*).
    e. *Combinatorial* rules construct test cases (e.g. *EP14* to *EP16*). ARTT currently does not implement this class of rules (see future work, see Section 4.10).
9. **Field Set Type***. Set type of the field that the rule can be applied to. Options are List, Range, List and Range or Neither (e.g., non-terminal fields like the <*address*> field of the Address Parser specification could be tested by Syntax Testing rules that substitute non-terminal fields for other terminal or non-terminal fields).
10. **Start Position** and **End Position**. These define the start and end points between which equivalence class and test data value are selected from fields under test. The user can choose from six types of start and end positions, as follows.
    a. **Datatype start and end positions** select equivalence classes between two boundaries of the datatype of the field under test. For example, if the field under test is an integer field, then a DSSR could be defined to select an equivalence class between the lower and upper boundaries of the datatype selecting the class [-32768 – 32767].
    b. **Field start and end positions** can be used by DSSRs to select an equivalence class from a range-based field. For example, they could select values between the lower and upper boundaries of a field <*house_number*> *::= [1 – 9999]*, which would select the valid partition [1 – 9999]. They can also be used by DISRs to select one data value from a partition, such as selecting the upper boundary of the <*house_number*> field, selecting the value *9999*.

c. **List value start and end positions** can be used by DSSRs to select an equivalence class from a list-based field. For example, they could be applied to the field *<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt]* to select the valid partition *[Street | St | Road | Rd | Avenue | Ave | Court | Crt]*. They can also be used by DISRs to select one data value from a list, such as selecting the lower boundary value *Street* from this partition.

d. **Nominal start and end positions** can be used by DSSRs, DISRs and DIMRs to select the mid-point value from a field, equivalence class (e.g. selecting the keyword '*Rd*' from the <*street_type*> field) or test data value (e.g. selecting the middle character '*u*' from the keyword '*Court*').

e. **Random start and end positions** can be used by DSSRs, DISRs or DIMRs to select a randomly chosen value from a field, partition or data value.

f. **First and last character start and end positions** can be used by DIRMs to alter a single test data value, such as selecting the letter '*R*' from the keyword '*Road*' or selecting '*reet*' from the keyword '*Street*'.

Therefore, the values that appear in the Start and End Position fields depend on the value of Rule Type, as follows.

If Rule Type = DSSR then rule selects a partition. Start and End Positions are:

i. Datatype Lower Boundary – (e.g. ASCII A – 1 = @)
ii. Datatype Lower Boundary (e.g. ASCII A = A)
iii. Datatype Lower Boundary + (e.g. ASCII A + 1 = B)
iv. Datatype Upper Boundary – (e.g. ASCII Z – 1 = Y)
v. Datatype Upper Boundary (e.g. ASCII Z = Z)
vi. Datatype Upper Boundary + (e.g. ASCII Z + 1 = [ )
vii. Field Lower Boundary – (just below lower boundary of a range)
viii. Field Lower Boundary (on the lower boundary of a range)
ix. Field Lower Boundary + (just above the lower boundary of a range)
x. Field Upper Boundary – (just below the upper boundary of a range)
xi. Field Upper Boundary (on the upper boundary of a range)
xii. Field Upper Boundary + (just above the upper boundary of a range)
xiii. First Field Value (first value in a list)
xiv. Last Field Value (last value in a list)
xv. Nominal Value (middle value of a range or list)
xvi. Random Value (random value from a range or list)

If Rule Type = DISR then rule selects a test data value, so End Position will be disabled. Start Positions are:

i. Field Lower Boundary –
ii. Field Lower Boundary
iii. Field Lower Boundary +
iv. Field Upper Boundary –
v. Field Upper Boundary
vi. Field Upper Boundary +
vii. First Field Value
viii. Second Field Value
ix. Second Last Field Value
x. Last Field Value
xi. Nominal Value
xii. Random Value
xiii. Not Applicable

If Rule Type = DIMR then the rule manipulates individual test data values. Start and End Positions are:

i. Nominal Value
ii. Random Value
iii. First Character – – (e.g. add two chars to start of a data value)
iv. First Character – (e.g. add one char to start of a data value)
v. First Character (e.g. mutate first character of a data value)
vi. First Character + (e.g. mutate second character of a data value)
vii. First Character ++ (e.g. mutate third character of a data value)
viii. Last Character – – (e.g. mutate third last character of a data value)
ix. Last Character – (e.g. mutate second last character of a data value)
x. Last Character (e.g. mutate last character of a data value)
xi. Last Character + (e.g. add one char to end of a data value)
xii. Last Character ++ (e.g. add two chars to end of a data value)
xiii. Not Applicable (does not select a particular character or value)

If Rule Type = TCCR then these fields will be empty, because TCCRs do not select test data, they create test cases, so this field is disabled.

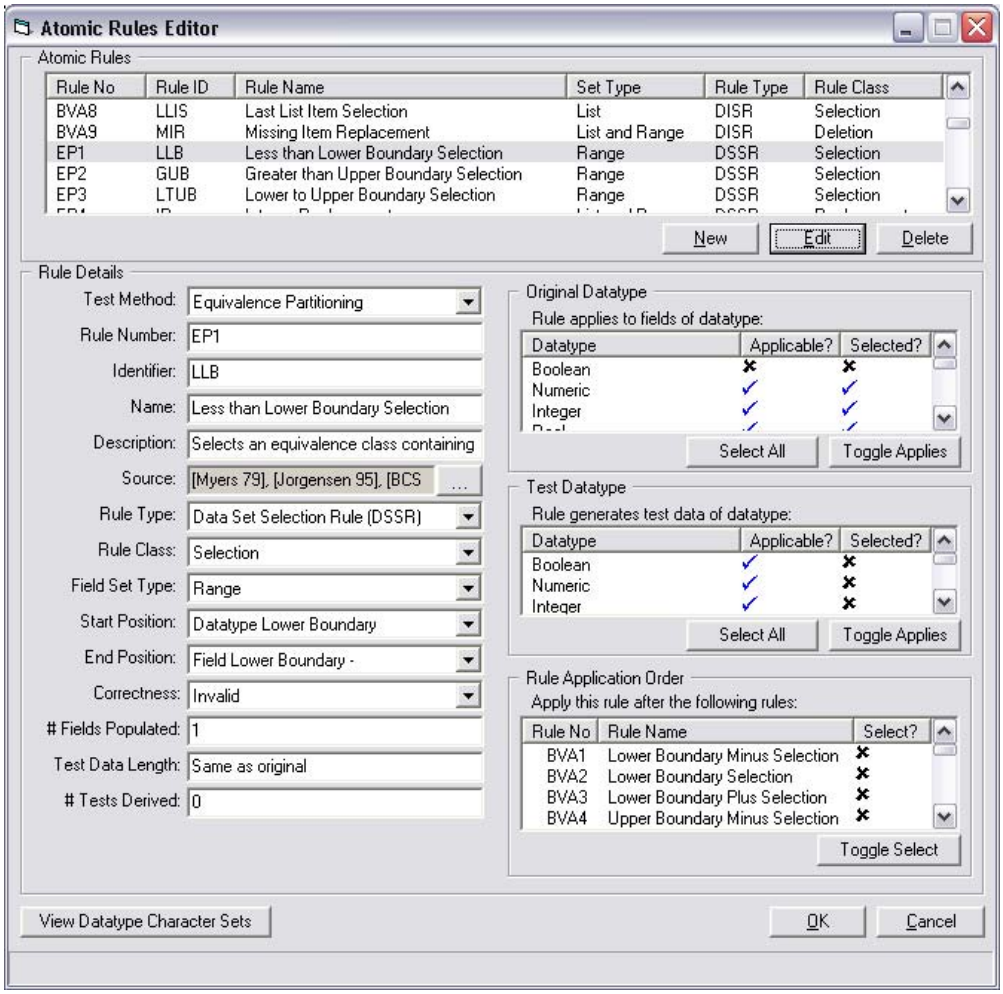
11. **Correctness***. Specifies whether the rule will select valid or invalid data in terms of what the program should accept and reject respectively.
12. **# Fields Populated***. This specifies the number of fields that will be populated with test data when the rule is applied.

13. **Test Data Length***. This specifies the length of the test data that will be derived by the rule.
14. **# Tests Derived***. For Test Case Construction Rules, this field contains an equation of the number of test cases derived.
15. **Original Datatype** and **Test Datatype***. These attributes allow the user to choose the datatypes the rule can be applied to (Original Datatype) and the datatypes the rule generates (Test Datatype). For example, *BVA1: Lower Boundary – Selection* and *BVA6: Upper Boundary + Selection* can only be applied to range-based datatypes such as 'integer'; they cannot be applied to list-based datatypes like 'alphanumeric' as there is no way to choose an outside boundary value from this datatype. The original and test datatypes implemented in ARTT extend the base set defined for EP, BVA and ST (see Section 3.2.2), as follows (see Appendix G for complete datatype definitions):
    a. Integer (all integers from -32768 to 32767)
    b. Integer+ (all positive integers from 0 to 32767)
    c. Integer- (all negative integers from -32768 to -1)
    d. Boolean (i.e. 1 and 0)
    e. Numeric (i.e. ASCII 48 to ASCII 57)
    f. Real (all Reals from -32768.00 to 32767.00)
    g. Real+ (all positive Reals from 0.00 to 32767.00)
    h. Real- (all negative Reals from -32768.00 to -1.00)
    i. Alpha (all alphabetical characters from A-Z and a-z)
    j. Lowercase Alpha (all lowercase alphas from ASCII 97 to ASCII 122)
    k. Uppercase Alpha (all uppercase alphas from ASCII 65 to ASCII 90)
    l. Alphanumeric (Alpha ∪ Numeric)
    m. Control Character (all control characters from ASCII 1 to ASCII 31)
    n. Symbol (Set 1) (all special characters from ASCII 32 to ASCII 47)
    o. Symbol (Set 2) (all special characters from ASCII 58 to ASCII 64)
    p. Symbol (Set 3) (all special characters from ASCII 91 to ASCII 96)
    q. Symbol (Set 4) (all special characters from ASCII 123 to ASCII 127)
    r. Symbol (Symbol 1 ∪ Symbol 2 ∪ Symbol 3 ∪ Symbol 4)
    s. Null (empty) (ASCII 0)
    t. Non-Alphanumeric (Symbol 1 ∪ Symbol 2 ∪ Symbol 3 ∪ Symbol 4)
    u. ASCII (all characters in the ASCII table)
    v. Same as original (applies to Test Datatype only)
16. **Rule Application Order**. This allows the user to specify which Atomic Rules the rule can be applied after. For example, DIMRs like *BVA1* to *BVA9* can be applied to a field after

DSSRs *EP1* to *EP11* have been applied to select partitions. Thus, this corresponds to the rule application order specified in the four-step test case design process.

Example pseudo code that was used to implement the application of a Data-Set Selection Rule to list-based and range-based fields to select an equivalence classes is given in Appendix F, Section F.4.

Figure 4-9: The Atomic Rules Editor.



4.6.1 Example of a New Atomic Rule

The following example illustrates how the fields of the Atomic Rules Editor (Figure 4-9) can be populated to define a new Atomic Rule. The example is based on the new rule *BVA12: Second List Item Selection*, which was identified during the industry evaluation of the Atomic Rules approach (see Chapter 6 Section 6.4.11 and Appendix B Section B.2).

1. **Test Method**. Boundary Value Analysis.
2. **Rule Number**. BVA12.
3. **Identifier**. SLIS.
4. **Name**. Second List Item Selection.

5. **Description**. Selects the second item in a list.
6. **Source**. N/A (rule was defined as a part of this research).
7. **Rule Type**. Data-Item Selection Rule (DISR).
8. **Rule Class**. Selection
9. **Field Set Type**. List.
10. **Start Position** and **End Position**.
    a. **Start Position**. Second field value.
    b. **End Position**: Second last field value.
11. **Correctness**. Valid (assuming the equivalence class contains only valid values).
12. **# Fields Populated**. 1.
13. **Test Data Length**. Same as original.
14. **# Tests Derived**. 0.
15. **Original Datatype** and **Test Datatype**.
    a. **Original Datatype.** Applies to all datatypes except Boolean and Null, as all other datatypes have a second and second-last item.
        i. Numeric (i.e. ASCII 48 to ASCII 57)
        ii. Real (all Reals from -32768.00 to 32767.00)
        iii. Real+ (all positive Reals from 0.00 to 32767.00)
        iv. Real- (all negative Reals from -32768.00 to -1.00)
        v. Alpha (all alphabetical characters from A-Z and a-z)
        vi. Lowercase Alpha (all lowercase alphas from ASCII 97 to ASCII 122)
        vii. Uppercase Alpha (all uppercase alphas from ASCII 65 to ASCII 90)
        viii. Alphanumeric (Alpha $\cup$ Numeric)
        ix. Control Character (all control characters from ASCII 1 to ASCII 31)
        x. Symbol (Set 1) (all special characters from ASCII 32 to ASCII 47)
        xi. Symbol (Set 2) (all special characters from ASCII 58 to ASCII 64)
        xii. Symbol (Set 3) (all special characters from ASCII 91 to ASCII 96)
        xiii. Symbol (Set 4) (all special characters from ASCII 123 to ASCII 127)
        xiv. Symbol (Symbol 1 $\cup$ Symbol 2 $\cup$ Symbol 3 $\cup$ Symbol 4)
        xv. Non-Alphanumeric (Symbol 1 $\cup$ Symbol 2 $\cup$ Symbol 3 $\cup$ Symbol 4)
        xvi. ASCII (all characters in the ASCII table)
    b. Test Datatype. Same as original, as the rule does not alter the datatype of the original field.
16. **Rule Application Order**. Can be applied after *EP1* to *EP10*.

Thus, this demonstrates how the attributes of the new rule *BVA12: Second List Item Selection* can be assigned in ARTT to define a new Atomic Rule.

4.7 Specification Notation

ARTT outputs specifications in a language that uses a combination of extended BNF (EBNF) (see (Knuth 1964) for early definition of BNF) and PL/I (Barnes 1979) syntax (Table 4-2). One of the unique aspects of this notation is that it uses a 'REPEATS' tag to specify the minimum and maximum number of times a field can repeats (see Table 4-2 row 4, Figure 4-2 and Figure 4-10).

Table 4-2: The EBNF language to be used by the SBSMT simulator.

Character	Meaning
<c>	<c> is a terminal field
<d> ::= <c>	<d> is a non-terminal composed of <c>
<c>REPEATS[MIN-MAX]	<c> repeats between MIN and MAX times
<c>?	<c> is optional
<c> | <d>	Select one of the terminals <c> or <d>
[a – d]	A range of lowercase alpha characters from a to d
[A | B | C | D]	A list of uppercase alpha characters

Figure 4-10: Example of the EBNF representation of a specification stored in ARTT.

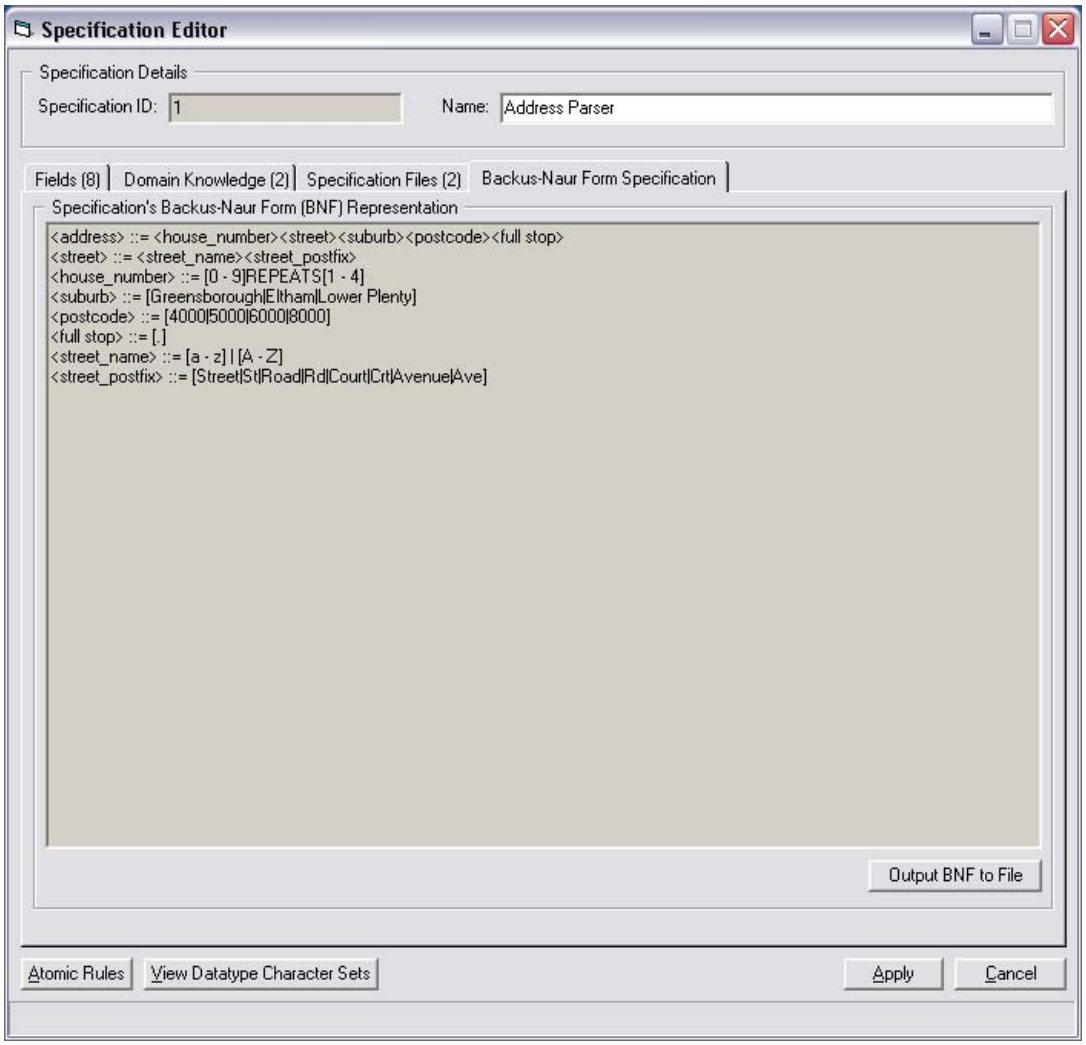
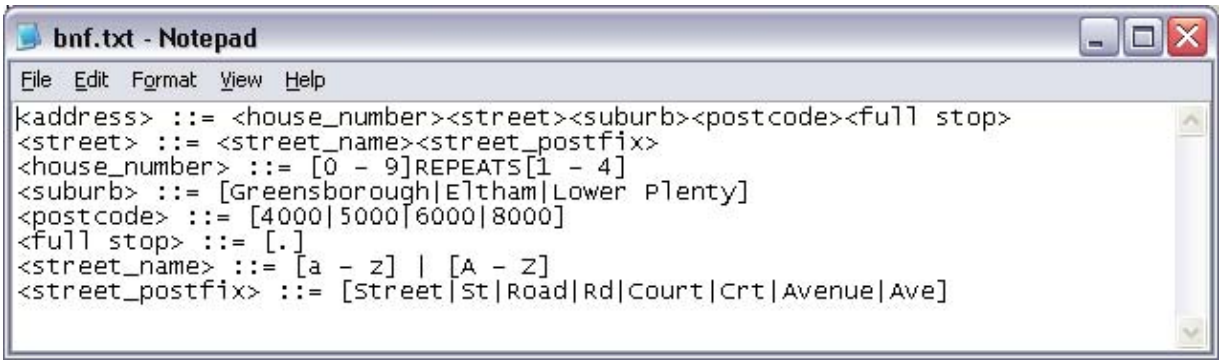


Figure 4-11: Example of an EBNF specification output by ARTT.



4.8 Benefits

ARTT presents a number of important benefits to the software testing community, as follows.

a) ARTT improves the efficiency of black-box test data generation (compared to manual testing). For example, manual design of test data values using EP, BVA and ST for one input field could take up to an hour or more, compared to a matter of seconds in ARTT.

b) Reducing the amount of manual testing effort required during testing allows testers to allocate more time to complex testing issues that cannot be solved through automation.

c) ARTT ensures accuracy and repeatability in the black-box test data generation, as the same algorithm is used every time the tool is executed.

d) ARTT facilitates the automatic generation of novel test data by applying new combinations of Atomic Rules from EP, BVA and ST to input fields to select new types of test data.

e) Each Atomic Rule is documented with a list of reference sources (e.g. textbooks, standards, journals), which allows testers to locate additional information on each rule.

f) ARTT forces the characteristics of input fields to be properly specified, ensuring that thorough documentation for each field is stored and allowing it to be reused.

g) ARTT supports the capture of domain knowledge utilised during the specification process, allowing this information to be reused and shared with other testers.

h) ARTT supports testers in the creation of new Atomic Rules, allowing new rules to be recorded, used and shared with other testers.

4.9 Limitations

The current limitations of ARTT are as follows.

a) ARTT currently only automates DSSRs, DISRs and DIMRs from EP, BVA and ST. Future work will include the development of algorithms for automating TCCRs, as well as implementation of Atomic Rules from other black-box testing methods.

b) Test data is only generated for input fields, not output fields. This would require identification of the conditions under which a particular output will be generated, most likely through implementation of algorithms (e.g. testing the output of a function that calculates total sales made requires the sum of prices paid for each item sold).

c) Input field specifications must be input through a GUI. Future implementations will include an automatic upload facility to input specifications in EBNF.

d) When a new input field is specified, it would be useful if ARTT could automatically identify any similar field definitions in the ARTT database. For example, if a new field <street> was specified, any existing fields with the same name could be presented to the user, with an option

of adopting the existing definition. This would reduce the time to specify input fields and would allow different testers to learn from each other.

e) ARTT does not support prioritisation of test data. It would be useful if the most effective Atomic Rules (and the test data they generate) could be flagged by a user, according to existing research on which black-box test case design rules are the most effective[1].

f) ARTT does not currently generate test data for testing field repetition. For example, if a field is specified as *<house_number> ::= [0-9]REPEATS[1-4]*, ARTT does not apply rules to test the repetition of the field (e.g. to select test data that contains too many digits).

4.10 Future Improvements

A number of improvements are planned for the Atomic Rules Testing Tool (ARTT), as follows.

- ARTT can be extended to automatically generate program source code for input data validation in various languages. For example, if a program input field should only accept integers within a certain range, then ARTT could apply Atomic Rules from EP and BVA to generate source code that accepts integers within the valid range and rejects all other input values. This would reduce the need for black-box testing and would ensure that programs only accept valid input data.

- Since ARTT currently only generates test data, not test cases, in future the tool will be extended to include implementation of black-box TCCRs, which will enable the automatic generation of complete black-box test cases.

- The Specification Editor will be enhanced to allow users to import EBNF specifications, enabling more efficient specification creation.

- The Specification Editor could be enhanced to provide users with guidance on the creation of input fields for new specifications, based on the names of previously defined input fields in existing specifications, which would enable reuse of specifications and domain knowledge. For example, if a user previously created a specification for an Address Parser program, which consisted of input fields like <street_name> and <suburb>, then any time a user creates a new specification with input fields of the same (or similar) names, ARTT could automatically suggest that the user utilise the previous definitions. ARTT could also be enhanced to recommend the use of any domain knowledge that was attached to the previously defined input fields, as well as the set of Atomic Rules that were applied to that specification.

- The Domain Knowledge tab of the Specification Editor will be enhanced to include more precise identification of the source of each item of domain knowledge, such as page number and a navigation tool to access, search and reference online sources.

[1] For example, Atomic Rule prioritisation could be based on the outcomes of the experiment reported in Chapter 6 of this thesis.

- The Domain Knowledge of the Specification Editor could also be enhanced to identify commonalities in the domain knowledge recorded against different specifications.

- ARTT will be extended to automatically produce an abstract syntax tree of specifications that are input by the user, providing them with a visual representation of the hierarchy in each specification.

- The implementation of Atomic Rules for BVA will be enhanced to support generation of test data values for fields that repeat.

- A function will be implemented to remove redundant test data generated by ARTT. For example, in Section 4.4.3, the same test data values are generated a number of times.

- Future work may also include the implementation of automated functions for generating test data values for testing output partitions (discussed in Chapter 3, Section 3.8).

- ARTT could also be enhanced to automatically construct 'unrestricted specifications' for the Category-Partition Method (CPM), by automatically combining test data values generated through EP, BVA and ST. ARTT could also allow testers to record the expected results of specific combinations of test data values, allowing complete CPM test scripts to be automatically generated.

- ARTT could be enhanced to integrate with unit testing tools like JUnit (JUnit) and code coverage analysis tools like JCover (Codework 2009), to gather information on the level of code coverage achieved by the test data and test cases that are generated when applying specific sets of Atomic Rules to particular program specifications.

4.11 Summary

In this chapter, the design and implementation of a prototype called the Atomic Rules Testing Tool (ARTT) was presented. The aim of developing ARTT was to improve the efficiency and accuracy of black-box test data generation. Currently, ARTT can be used to generate black-box test data by applying Atomic Rules from EP, BVA and ST to a specification of the input fields of a program. ARTT supports the capture of domain knowledge through GQASV and supports the definition of new Atomic Rules through SMT. In addition, ARTT is an example of a new approach to tool support for GQM-style processes.

Chapter 5

University Evaluation of the Atomic Rules Approach

"Not everything that can be counted counts, and not everything that counts can be counted."
Albert Einstein

5.1 Overview

As has already been pointed out, the Atomic Rules approach has been proposed to improve the usability and effectiveness of black-box testing methods by solving seven problems with method descriptions: *definition by exclusion*, *multiple versions*, *overlap*, *notational and terminological differences*, reliance on domain knowledge, *difficult to audit* and *difficult to automate*. In this chapter, the details of two classroom experiments that examined the usability of the Atomic Rules approach are presented. The experiments involved 32 undergraduate and graduate university students in 2004 (*n = 32*) and forty in 2005 (*n = 40*) who were enrolled in a combined third/fourth year subject on software testing at La Trobe University and who were considered to be novice software testers. The aim of the experiments was to compare the usability of the Atomic Rules representation of Equivalence Partitioning (EP) and Boundary Value Analysis (BVA) (Murnane, Hall & Reed 2005) to that of Myers' (1979) original definitions (see Chapter 2).

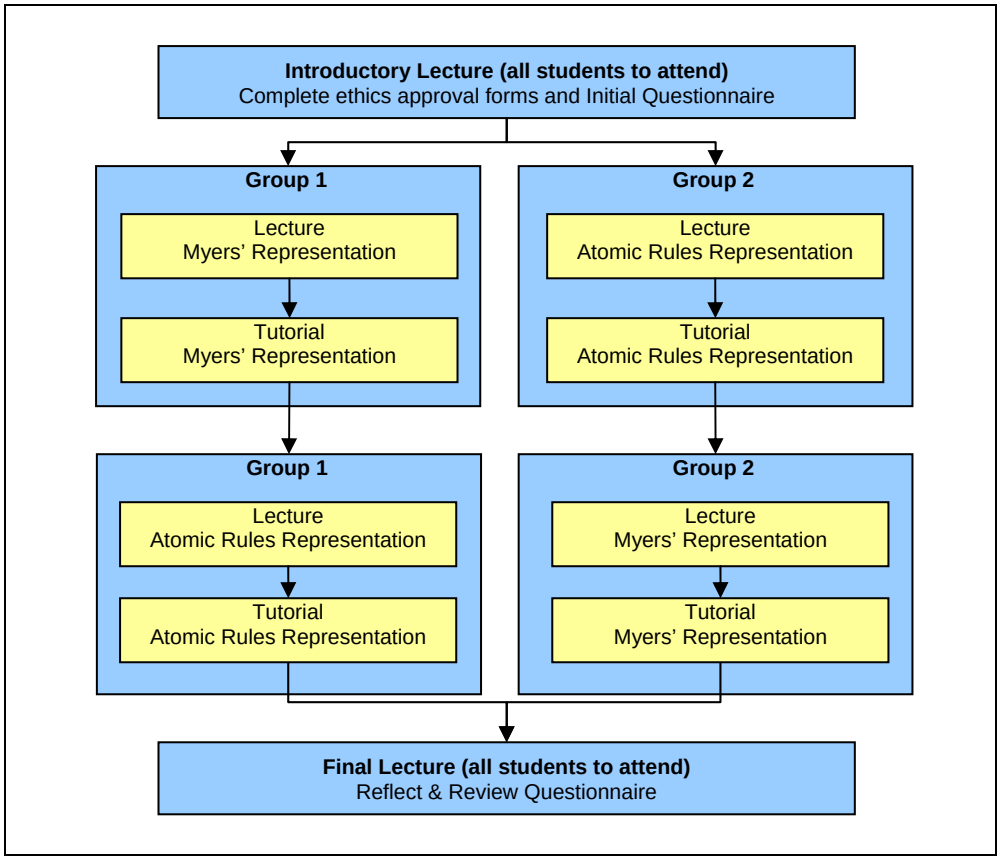
In the first week of the experiment (see Figure 5-1), all students attended an 'introductory' lecture, during which they were informed that an experiment would be taking place over a four week period and that they could chose to provide permission for their data to be included in the data analysis for this thesis if they wished (a requirement of ethics approval). They also completed an *Initial Questionnaire*, which was used to identify their current understanding of black-box testing methods and to gain an understanding of their prior programming and software testing experience.

In the second week of the experiment, the class was divided into two groups. While Group 1 was given a two-hour lecture on Myers' representation of EP and BVA, Group 2 was given a lecture on the corresponding Atomic Rules. During a subsequent tutorial, each group's comprehension of EP and BVA was assessed by asking them to derive black-box test cases using specially prepared 'toy' specifications.

In the third week, the groups were swapped and the process was repeated (i.e. Group 1 learnt Atomic Rules while Group 2 learnt Myers'). This was to ensure that all students had equal opportunity to learn the two representations in preparation for their assignment and exam, and to assess whether what they learnt about the representations in week 1 allowed them to improve the correctness of their test cases in week 2.

In the final week of the experiment, all students attended a one-hour lecture, which summarised the experiment aims. They were also asked to complete a *Reflect and Review Questionnaire*, to identify their initial and final understanding of EP and BVA and their preferred method representation. The students were not told what the aims of the experiment were until the end of the experiment (after completing the Reflect and Review questionnaire), to ensure that this knowledge did not bias their answers during the experiment.

Figure 5-1: Overview of the experiment process.



While the aims of the two experiments and the materials presented during lectures were the same for both years, three changes were made in 2005 that were significant enough for it to be considered to be a different experiment. As each group's lecture took place at the same time in different locations, two lecturers were required. In 2004, Ms. Murnane taught the Atomic Rules representation while Associate Professor Karl Reed taught Myers' representation. To eliminate the extraneous variable of student preference for lecturer, the lecturers were swapped in 2005. Also, the 2004 results suggested that students could handle more challenging specifications during tutorials. Thus, longer and more complex specifications were used in 2005. Lastly, to ensure students had enough time to complete their work, the tutorials were increased from one hour in 2004 to two hours in 2005. To avoid conflict with other university classes or commitments (a factor that could impact student performance (Carver et al. 2003)), all work for this part of the experiment was completed in class.

The remainder of this chapter is structured as follows. The experimental design is presented in Section 5.2, including hypotheses, group allocation, specifications used and threats to validity. Results are presented

in Section 5.3, followed by discussions of related research in Section 5.3.7 and experiment results in Section 5.5. The chapter is concluded with a summary in Section 5.6.

5.2 Experiment Design

The primary independent variable in these experiments was the first black-box testing method representation learnt by the students. The approach to manipulating the independent variable was the "type" technique (Johnson & Christensen 2004) in which the type of variable presented is varied over two separate treatments. One-tailed tests were used in all significance tests.

All tests for statistical significance were chosen by considering the particular class of experiment that was planned and the type of data it would generate, and then by consulting a series of textbooks on experimentation and statistical analysis (Anastasia & Urbina 1997, Carver et al. 2003, Christensen 2004, Creswell 2002, Healey 2005, Johnson & Christensen 2004, Klugh 1986, Vegas et al. 2003) and seeking advice from a statistician (Fielding 2004) and a PhD student in psychology (Barutchu 2004). Since there are typically more than one significance test that can be used for data analysis and hypothesis testing, this ensured that the specific tests that were chosen were appropriate. This included the t-test (Healey 2005, Klugh 1986, Barutchu 2004), chi-square test (Healey 2005, Klugh 1986, Barutchu 2004), Mann-Whitney U (Klugh 1986, Barutchu 2004), marginal homogeneity (Agresti 2007, Barutchu 2004), test of two proportions (Healey 2005, Fielding 2004) and cross-tabulation (Healey 2005, Barutchu 2004). The 'Results Coach' in SPSS (Statistical Package for the Social Sciences) was also used to support the choice of tests and to perform all calculations.

In the following subsections, the experiment hypotheses, group allocation and threats to validity are explored.

5.2.1 Hypotheses

Hypotheses for this experiment were based on the following definition of test method usability that was introduced in Chapter 1.

> **Test Method Usability. The extent to which a test case design method can be understood, learnt and used by software testers to achieve specified test case design goals effectively, efficiently and with satisfaction, within the context of applying software testing methods.**

Using the quantitative and qualitative attributes of this definition of usability that were defined in Chapter 1, the following null (H_{0X}) and scientific/alternate hypotheses (H_{1X}) (Christensen 2004) were identified.

Completeness (effectiveness):

> **H_{01}**: The completeness of the black-box test set derived by novice testers is independent of the representation used.

H_{11}: Novice testers using the Atomic Rules approach derive a more complete test set compared to those using Myers' representation.

Efficiency:

H_{02}: The efficiency of black-box test case derivation by novice testers is independent of the representation used.

H_{12}: Novice testers using the Atomic Rules approach derive test cases more efficiently compared to those using Myers' representation.

Errors Made (effectiveness – accuracy):

H_{03}: The number of errors made by novice testers during black-box test case derivation is independent of the representation used.

H_{13}: Novice testers using the Atomic Rules approach make fewer errors during test case derivation compared to those using Myers' representation.

Questions asked (learnability):

H_{04}: The number of questions asked by novice testers during black-box test case derivation is independent of the representation used.

H_{14}: Novice testers using the Atomic Rules approach ask fewer questions compared to those using Myers' representation.

Satisfaction:

H_{05}: The preference of novice testers towards the use of black-box testing methods is independent of the representation used.

H_{15}: Novice testers prefer to use the Atomic Rules approach for black-box test case design compared to the use of Myers' representation.

Understandability:

H_{06}: A tester's understanding of black-box testing methods is independent of the representation used.

H_{16}: Novice testers rate the Atomic Rules approach to black-box test case design as easier to understand than Myers' representation.

These hypotheses are similar to those used in an experiment by Vegas et al. (Vegas et al. 2003), which examined whether novice testers were able to select appropriate software testing methods from characterisation schema-based representations of black-box, white-box and fault-based testing methods, as compared to selecting appropriate techniques from textbook descriptions.

Attributes that have not been measured by the hypotheses above are as follows:

- **Operability** in terms of how easy a method is to use was not assessed, as data to measure this unfortunately not collected during the experiment.

- **Learnability** in terms of how long it takes a tester to become competent was also not assessed, as the university semester only permitted the experiment to be carried out over two weeks of classes and each class had a limited duration.

5.2.2 Group Allocation

In each year, the participants were divided into two comparison groups. To provide repetition, each group was divided into two subgroups, with each deriving test cases from a different specification (Table 2). A discussion of the affect of group allocation on experiment validity is provided in Section 5.2.4.1.

Table 5-1: Group allocation.

Year	Group	Myers	Atomic Rules
2004	subgroup 1	13	8
	subgroup 2	5	6
	Total	***18***	***14***
2005	subgroup 1	10	8
	subgroup 2	10	12
	Total	***20***	***20***

5.2.3 Input Data Specifications

The main requirement of the specifications used during tutorials was that they had to include at least one numerical range, one list of values and a various datatypes (e.g. alphas, numbers and non-alphanumeric) to ensure that students have the opportunity to derive test cases for a 'base' collection of set types and datatypes. The two fictional specifications used in 2004 were for a Personal Details Recording System (Figure 5-2) and an Office Location Recording System (Figure 5-3), while the 2005 specifications were for a Patient Record System (Figure 5-4) and a Book Referencing System (Figure 5-5). All specifications were written in a semi-formal notation and contained input fields defined using a combination of Backus-Naur Form (BNF) and natural language.

There were two primary differences between these specifications: length and complexity. In 2004, the top level non-terminal node contained five fields, including two white-space fields, whereas the corresponding node in 2005 contained fourteen fields. Thus, the 2005 specifications were substantially longer. Also, the 2005 specifications contained a recursive field definition, which made test case derivation more challenging; e.g., the <*level_digits*> field was recursive. To compensate for this change, the tutorials were extended from one hour in 2004 to two hours in 2005.

Figure 5-2: Specification for a personal details recording system.

Specification

<personal_details>	::= <id_number> <s> <surname> <s> <gender>
<id_number>	::= [100 – 999]
<surname>	::= 1 to 100 characters from the sets alpha and non-alphanumeric (i.e. letters and/or symbols)
<gender>	::= [Male | Female]
<s>	::= one to seven single spaces

Example Record

555 Smith Male

Figure 5-3: Specification for an office location recording system.

Specification

<office_location>	::= <floor_number> <s> <room_name> <s> <desk_type>
<floor_number>	::= [001 – 200]
<room_name>	::= 1 to 100 characters from the sets alpha and non-alphanumeric (i.e. letters and/or symbols)
<desk_type>	::= [Desk | Cubicle]
<s>	::= one to seven single spaces

Example Record

100 The Blue Room Cubicle

Figure 5-4: Specification for a patient details record system.

Specification

<patient_record>	::= <name> ∧…"<ailment>"∧…<floor_no>,∧…<building>,∧…<patient_no>
<name>	::= 1 to 100 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)
<ailment>	::= 1 to 150 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)
<floor_no>	::= <level_no> ∧ Floor
<level_no>	::= <level_digits><level_postfix>
<building>	::= [Fredrick Building | John-Scott Memorial Ward | Mary House | Norman Building | … | Zane Square Building | Zoo Ward]
<patient_no>	::= <d><d><d><d>
<d>	::= [0 – 9]
<level_digits>	::= <d> | <level_digits> <d>
<level_postfix>	::= [nd | rd | st | th]
∧	::= one space
∧…	::= one or more spaces

Example Record

Joe Hamish Bloggs "Viral pneumonia, ear infection, and lower abdomen pain" 5th Floor, John-Scott Memorial Ward, 1234

Figure 5-5: Specification for a book referencing system.

Specification

<book_reference>	::= <author>∧..."<title>"∧…<edition >,∧...<publisher>,∧…<year>
<author>	::= 1 to 150 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)
<title>	::= 1 to 100 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)
<edition>	::= <edition_no> ∧ Edition
<edition_no>	::= <edition_digits><edition_postfix>
<publisher>	::= [Addison-Wesley | Artech House | Babbage Press | C & G Publishing Inc | ... | Zipper Press Inc | Zoo House Publishers Inc]
<year>	::= <d><d><d><d>
<d>	::= [0 – 9]
<edition_digits>	::= <d> | <edition_digits> <d>
<edition_postfix>	::= [nd | rd | st | th]
∧	::= one space
∧…	::= one or more spaces

Example Record

Walter Savitch "Problem Solving with C++ - The Object of Programming" 5th Edition, Addison-Wesley, 2005

5.2.4 Threats to Validity

5.2.4.1 Internal Threats to Validity

History. Experiment outcomes can be biased by time lapses between the application of treatment variables and measurement of dependent variables, between pre-test and post-test measurements (Christensen 2004) or if discussions take place between groups during that time (Creswell 2002). This posed a minimal threat in this case, since treatment took place during a lecture that was between two hours and three days prior to measurement. To combat this threat, participants were asked not to discuss the experiment with each other until after the final lecture. As there was no assignment or exam during the experiment, based on the material, the students were not expected to have a great need to hold discussions during the three weeks of the experiment.

Maturation. Changes or differences in a participant's internal condition (e.g. age, hunger, fatigue, boredom) (Christensen 2004), knowledge level (Creswell 2002), lecturer preference, or enthusiasm (Vegas et al. 2003) can bias results. For example, students who are excited about being involved in an experiment on a new technique may work harder on that technique. To ensure this did not bias results, Myers' representation was referred to as *Model 1* and the Atomic Rules approach as *Model 2*. Students were not told which was new until after the final lecture. Also, students were informed that there would be a gift for every member of the class at the end of the experiment whether they chose to participate or not, which was hoped to compensate them for any disruption they may have experienced during the experiment. A gift of chocolate, which was allowed by the university's ethics committee, was given to all students to thank them for their participation in the experiment. To combat boredom, students were reminded that the work they completed during tutorials would prepare them for their later assignments and exam in the subject and also for future work in industry. As lecture and tutorial attendance was not compulsory, bored students could choose not to attend class but all students were informed that the subject was an important part of their courses, in the hope that this would motivate them to attend. As the experiments were run over six separate classes, it was assumed that fatigue and hunger would not affect results. Selecting students from the same year levels should have negated the knowledge threat (Creswell 2002). To mitigate the potential lecturer preference bias threat, the lecturers were swapped in 2005.

Instrumentation. This threat relates to research observers becoming accustomed to experiment materials or increasing their experience in measuring data (Creswell 2002). To ensure the same standards were followed throughout analysis, standard measurement scales and analysis processes were followed. To standardise the analysis processes followed, one person was responsible for all data analysis.

Selection. Random group allocation can be used to mitigate the threat that the groups were biased; e.g. if one group has a higher mean intelligence level than the other (Creswell 2002). Conversely, if participants allocate themselves to groups, then the sample within each group is voluntary, not random, allocation (Berry & Tichy 2003). While random allocation was achieved in 2005 by drawing participant's names out of a hat, it was not achieved in 2004 due to a timetabling problem, which resulted in students allocating themselves to groups according to their chosen tutorial day/time. Subsequent analysis of the average grade

achieved within the two groups for the subject revealed no significant difference between the groups (see Section 5.2.4.3). Thus, this threat should not have biased experiment results.

Testing. Bias can occur if participants are given the same test more than once and become familiar with required responses (Creswell 2002). Although the experiment subjects derived test cases for the two representations over two weeks, results of the second week were not included in experiment analysis, as this would not measure their understanding of the representation learnt. Rather, it would test how well they adjusted to learning a second representation. This statistic could be analysed in future, in an assessment of whether industry testers would adjust to using Atomic Rules after having used different approaches as part of their jobs.

Reliability. This relates to the consistency of results being obtained from the same person with the same or equivalent tests on different occasions, allowing an error of measurement to be calculated (Anastasia & Urbina 1997). The simplest approach is to repeat the experiment on two separate occasions, where the error of measurement is a reliability coefficient which is the correlation between the two scores for each individual (Anastasia & Urbina 1997). Since these experiments took place during university semesters, there was not enough time to repeat the same test twice. However, within each group, the same test was repeated across two subgroups, which enabled testing of experiment reliability.

Population and sample. Validity can be affected if the sample is not representative of the entire population (Gorard 2001). Convenience sampling was used in these experiments, where participants were selected because they were easily accessible to the research team (Gorard 2001). As the resulting samples were not representative of all novice testers, program specifications or black-box testing methods (e.g. approaches such as Syntax Testing were not covered), the results are considered to be indicative.

Threats to internal validity, which were not applicable to these experiments, include diffusion of treatments, compensatory equalization, compensatory rivalry and resentful demoralization, as these only apply when using control groups (Creswell 2002). Control groups could not be used, as students in those groups would have been disadvantaged in their assignment and exam as a result of not learning the two representations. Also, in experiments involving students, participants sometimes work on tasks at home; thus copying is a threat (Vegas et al. 2003). In these experiments, all tasks were completed in class and every second student was given a different specification during tutorials so they could not copy from each other. A bias can also exist if participants do not follow the processes and procedures of the techniques prescribed (Vegas et al. 2003). Therefore, students were asked to show all workings during test case derivation, so that it was possible to verify that they followed the prescribed procedures of each test method, as this could be used to identify whether there were any ambiguities in the methods.

5.2.4.2 External Threats to Validity

Language. Participants may be disadvantaged if experiment materials are not written in their native language (Vegas et al. 2003). Although some international students were involved and all materials were

written in English, as the students were enrolled at an English-speaking university, it was expected that they would be able to understand the language used and if not, that they would ask questions.

Interaction of setting and treatment. This relates to the ability to generalise experiment findings across other environmental settings (Creswell 2002), such as determining whether the experiment results are applicable to industry professionals. This threat is not applicable as industry-based testers can be considered to be expert software testers, whereas these experiments were aimed at novices.

Interaction of history and treatment. This relates to the ability to generalise research outcomes to the past and future; e.g. if a classroom experiment runs during the main semester, the outcomes may be different if it were conducted over summer break, due to different types of students being enrolled (Creswell 2002). One way of resolving this is to replicate the experiment at a different time of year. These experiments were run during the same semester over two years and the experiments could not be repeated over summer as very few third and fourth year subjects run at that time at La Trobe University and there have never been any official student requests to do so in this software testing subject.

5.2.4.3 Construct Validity Threats

Measures. If participants can guess the measures that will be used during data analysis then this may affect how they answer questions (Creswell 2002). For example, measures need to be complex enough so that participants are less likely to provide answers that specifically make them appear more competent in the particular techniques being used (Creswell 2002). To mitigate this threat, participants were not told how the data was going to be analysed until after the experiment was complete.

5.3 Results

5.3.1 Demographic

In the Initial Questionnaire, students were asked about their prior software testing and industry experience. Twenty-six out of thirty-two students completed this questionnaire in 2004 (81.25%), while thirty-seven out of forty completed it in 2005 (92.5%). Many students reported having prior experience with testing in university lectures and assignments (Table 5-2). While 19.2% in 2004 and 13.5% in 2005 reported having no prior experience with software testing methods, 73.1% in 2004 and only 48.6% in 2005 reported gaining that experience through university lectures.

Table 5-2: Prior software testing experience.

Software Testing Experience	2004 % (n = 26)	2005 % (n = 37)
None	19.2	13.5
University Lectures	**73.1**	48.6
University Assignments	61.5	**62.2**
As a Tutor	0	0
Other	11.5	16.2

Also, 84.6% in 2004 reported having prior experience with black-box methods, compared with only 54.1% in 2005 (Table 5-3). These results suggests that there may have been a decrease in the amount of software testing training given to students in the 2005 group in earlier years of their degrees.

Table 5-3: Prior experience with black-box testing methods.

Ever used any black-box testing methods?	2004 % (n = 26)	2005 % (n = 37)
Yes	**84.6**	**54.1**
No	15.4	45.9

The students rated their experience with the following black-box testing methods: Boundary Value Analysis (BVA), Cause-Effect Graphing (CEG), Decision Tables (DT), Equivalence Partitioning (EP), Orthogonal Array Testing (OAT), Random Testing (RT), Specification-Based Mutation Testing (SBMT), State-Transition Diagram Testing (STT), Syntax Testing (ST) and Worst Case Testing (WCT). Although data was only required for EP and BVA, the questionnaire enquired about nine other methods to obtain an overall picture of the group's current black-box testing knowledge. Students rated their understanding using a Likert scale of: 1 = none, 2 = basic, 3 = intermediate, 4 = advanced and 5 = expert (Table 5-4 and Table 5-5). With the exception of BVA and RT in 2004, the majority of students reported having limited amounts of experience with black-box testing methods.

Table 5-4: Participants initial understanding of black-box testing methods in 2004 (n = 26).

	Black-Box Testing Methods										
	BVA	CEG	DT	EP	EG	OAT	RT	SBMT	STT	ST	WCT
Rating	Percentages (%)										
None	19	**96**	**54**	**65**	**73**	**100**	**46**	**96**	**73**	**54**	**65**
Basic	15	0	8	4	8	0	15	0	8	8	15
Intermediate	**31**	4	27	8	12	0	31	4	12	23	12
Advanced	23	0	8	23	4	0	4	0	8	15	8
Expert	12	0	4	0	4	0	4	0	0	0	0

Table 5-5: Participants initial understanding of black-box testing methods in 2005 (n = 37).

	Black-Box Testing Methods										
	BVA	CEG	DT	EP	EG	OAT	RT	SBMT	STT	ST	WCT
Rating	Percentages (%)										
None	**65**	**95**	**76**	**84**	**81**	**97**	**73**	**97**	**87**	**78**	**90**
Basic	16	0	5	5	8	0	19	0	8	14	5
Intermediate	16	5	16	11	11	3	8	3	5	8	5
Advanced	3	0	3	0	0	0	0	0	0	0	0
Expert	0	0	0	0	0	0	0	0	0	0	0

Interestingly, very few students reported having prior experience working in industry (Table 5-6).

Table 5-6: Prior industry experience.

Position in Industry	2004 % (n = 26)	2005 % (n = 37)
Project Manager	3.8	0
Technical Team Leader	3.8	0
Business Analyst	0	0
Programmer	3.8	2.7
Analyst	3.8	2.7
Test Team Leader	3.8	0
Test Team Member	0	2.7
Other	0	5.4

A comparison of the mean overall grade of each group in the subject showed that there was no significant difference between the groups in 2004 or 2005 (Table 5-7).

Table 5-7: Comparison of overall grades for each group.

Year	N	Approach	Mean %	Std Dev	t-test
2004	18	Myers	66.17	20.88	$t(38) = .428$, $p = .336$
	14	Atomic Rules	73.29	14.56	
2005	20	Myers	68.1	19.49	$t(30) = -1.08$, $p = .143$
	20	Atomic Rules	65.8	14.1	

5.3.2 Completeness (Effectiveness) (H_{01}/H_{11})

To assess completeness, a 'complete' set of test data values and test cases for EP and BVA were derived by the author of this thesis, to represent the 'ultimate' test sets that were derivable for the specifications under test. This was carried out by applying every test case design rule from both Myers' definition of EP and BVA and the corresponding Atomic Rules, to every possible input field.

Then, the percentage of EP equivalence classes, BVA boundary values and EP and BVA test cases derived correctly by each group of students was compared. In 2004, a t-test revealed a significant difference between the groups for EP equivalence class and EP test case derivation, where the mean was higher for the Atomic Rules group (Table 5-8, Table 5-9). According to Cohen's Effect Size (Mujis 2004), these relationships were 'strong.' The 2004 BVA results were inconclusive (Table 5-10, Table 5-11). Conversely, in 2005 the mean EP equivalence class and test case coverage (Table 5-8, Table 5-9) and BVA boundary value coverage (Table 5-10) was significantly higher for Myers' group and Cohen's Effect Size showed moderate to strong relationships. The results for BVA test cases in 2005 were inconclusive (Table 5-11).

Interestingly, the mean EP and BVA coverage by Myers' group in 2004 and 2005 was relatively similar over both years (Table 5-8, Table 5-10).

Table 5-8: Percentage of coverage of EP equivalence classes (completeness – H_{01}/H_{11}).

Year	N	Approach	Mean %	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	49.76	16.95	$t(30) = -3.82$, $p = .0003$	1.35 – strong
	14	Atomic Rules	78.86	26.10		
2005	20	Myers	48.61	16.13	$t(38) = 4.815$, $p < .001$	1.53 – strong
	20	Atomic Rules	26.54	12.65		

Table 5-9: Percentage of coverage of EP test cases (completeness – H_{01}/H_{11}).

Year	N	Approach	Mean %	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	36.23	23.29	$t(30) = -4.87$, $p = .0002$	1.73 – strong
	14	Atomic Rules	78.86	26.10		
2005	20	Myers	38.94	20.92	$t(38) = 2.649$, $p = .006$	0.85 – moderate
	20	Atomic Rules	23.65	15.11		

Table 5-10: Percentage of coverage of BVA boundary values (completeness – H_{01}/H_{11}).

Year	N	Approach	Mean %	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	18.88	19.98	$t(30) = .58$, $p = .28$	NA
	14	Atomic Rules	23.81	26.82		
2005	20	Myers	26.00	19.51	$t(38) = 2.776$, $p = .004$	.90 – moderate
	20	Atomic Rules	11.52	12.80		

Table 5-11: Percentage of coverage of BVA test cases (completeness – H_{01}/H_{11}).

Year	N	Approach	Mean %	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	14.88	18.83	$t(40) = -.39$, $p = .35$	NA
	14	Atomic Rules	18.81	26.22		
2005	20	Myers	10.22	16.92	$t(38) = .141$, $p = .445$	NA
	20	Atomic Rules	9.56	12.63		

The results obtained in 2005 appear to not support the view that the Atomic Rules approach improves the usability of EP and BVA, while then results from 2004 do support it. In 2005, a significantly larger specification was used. As a result, the Atomic Rules approach required the derivation of significantly larger numbers of equivalence classes and boundary values, since it contains more rigorous test case design rules than Myers' definition. For example, Myers' "has something else" rule is decomposed into seven different Atomic Rules (see Chapter 3, Section 3.2.3). As a result, testers using the Atomic Rules approach have to derive significantly more equivalence classes (Table 5-12) and boundary values (Table 5-13) to produce test sets that are as complete as those using Myers' approach, and in this experiment, the students may simply have not had enough time to do this properly. Myers' definition of EP was able to produce 23 and 61 equivalence classes in 2004/2005, compared to 50 and 133 for the Atomic Rules approach (Table 5-12). Myers' definition of BVA was able to produce 28 and 45 boundary values in 2004/2005, compared to 45 and 75 for the Atomic Rules approach. Thus, the efficiency of the representations must be considered when examining test method usability (see Section 5.3.3).

Table 5-12: Number of equivalence classes derivable when EP is applied 'completely' to the specifications under test.

Year	Approach	Equivalence Classes Coverable (count)	Ratio
2004	Myers	23	1:2.17
	Atomic Rules	50	
2005	Myers	61	1:2.18
	Atomic Rules	133	

Table 5-13: Number of boundary values derivable when BVA is applied 'completely' to the specifications under test.

Year	Approach	Boundary Values Coverable (count)	Ratio
2004	Myers	28	1:1
	Atomic Rules	28	
2005	Myers	45	1:1.76
	Atomic Rules	79	

5.3.2.1 Completeness in Class Assignments

In 2005, the students were also asked to derive black-box test cases in their assignment using one of the approaches used in the experiment. This task was carried out in the four weeks following the experiment. It was interesting to find that significantly more students chose to use the Atomic Rules approach in their assignment in that year (Table 5-14).

Table 5-14: Representation used in the assignment (n = 38).

Year	Approach	Used in Assignment (%)	Chi-Square
2005	Myers	27.5	$\chi^2(1, N = 38) = 6.737, p = .009$
	Atomic Rules	67.5	

The average assignment mark in 2005 for students who chose to use the Atomic Rules approach was significantly higher (Table 5-15). Students who used Myers' representation scored an average of 68% on their assignment, while those who used the Atomic Rules approach achieved an average of 86%, which is nearly 20% or two grades different (i.e. the difference between a 'C' and an 'A' grade).

Table 5-15: Average mark achieved in the assignment compared by approach (n = 38).

Year	Approach Used on Assignment	Mean Assignment Mark (%)	t-test
2005	Myers	67.91	$t(36) = -1.93, p = .03$
	Atomic Rules	85.52	

A question that was raised after the experiment was complete was whether the 'brighter' students chose to use the Atomic Rules approach in their assignment because they knew they could achieve a higher grade with that approach. This was assessed by comparing the two groups of students by the approach used

(Myers or Atomic Rules) with the 'overall mean mark' they achieved in the subject, which was a combination of their exam mark that was worth 70% of their overall grade, and an assignment mark that was worth 30% of their overall grade. (Table 5-16). Interestingly, no significant difference was found, indicating that student intelligence levels did not affect their choice of representation on the assignment.

Table 5-16: Average mark achieved in the subject compared by approach (n = 38).

Year	Approach Used on Assignment	Mean Overall Subject Mark (%)	t-test
2005	Myers	63.18	$t(36) = -1.03$, $p = .15$
	Atomic Rules	69.44	

5.3.3 Efficiency (H_{02}/H_{12})

Efficiency can be assessed by tester productivity (see Section 1.3.1), which in this experiment is the number of correct EP equivalence classes and BVA boundary values derived over the total time taken (60 minutes in 2004, 120 minutes in 2005). EP and BVA test cases were excluded as many students did not have enough time to derive test cases (see Section 5.3.3.2). Erroneous equivalence classes and boundary values were excluded since the definition of efficiency refers to 'correct' test cases.

5.3.3.1 Productivity

Productivity was assessed in two stages. First, a comparison of the mean number of *correct* EP equivalence classes (Table 5-17) and BVA boundary values (Table 5-18) was performed (i.e. despite time taken). For EP, a t-test indicated a significant difference in 2004 and a difference that was just outside the 95% confidence interval in 2005, where the mean was higher for students using the Atomic Rules approach and Cohen's effect size indicated moderate to strong relationships (Table 5-17). This indicates that novice testers are more productive when using the Atomic Rules approach. The 2004 and 2005 results for boundary value derivation were inconclusive (see Table 5-18).

Table 5-17: Number of correct EP equivalence classes derived (efficiency – H_{02}/H_{12}).

Year	N	Approach	Max Number of Equivalence Classes Derivable	Mean Number of Correct Equivalence Classes Derived	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	23	8.89	4.43	$t(30) = -8.01$, $p < .01$	3.33 – strong
	14	Atomic Rules	50	37.93	12.99		
2005	20	Myers	61	25	12.82	$t(38) = 1.62$, $p = .06$	0.52 – moderate
	20	Atomic Rules	133	32.9	17.6		

Table 5-18: Percentage of correct EP equivalence classes derived (efficiency – H_{02}/H_{12}).

Year	N	Approach	Max Number of Boundary Classes Derivable	Mean Number of Correct Boundary Values Derived	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	28	5.83	5.04	$t(30) = .603$, $p = .28$	NA
	14	Atomic Rules	28	4.57	6.80		
2005	20	Myers	45	11.30	8.68	$t(38) = 1.20$, $p = .11$	NA
	20	Atomic Rules	79	7.90	9.72		

The second stage of evaluating efficiency compared tester productivity as determined by the quantity of equivalence classes (Table 5-19) and boundary values (Table 5-20) derived per unit of time (60 minutes in 2004 and 120 minutes in 2005). For EP, a t-test indicated a significant difference in 2004 and a difference that was just outside the 95% confidence interval in 2005, where the mean was higher for students using the Atomic Rules approach and Cohen's effect size indicated moderate to strong relationships (Table 5-19). This suggests that novice testers are able to be more productive using the Atomic Rules approach.

The results for BVA in both years were inconclusive, as no significant differences were found. This was caused by most students not having enough time to complete their BVA derivations during the tutorials.

Table 5-19: Productivity of the testers in terms of the number of equivalence classes derived over total time taken (efficiency – H_{02}/H_{12}).

Year	N	Approach	Mean Number of Correct Equivalence Classes Derived Per Unit of Time	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	.15	.07	$t(30) = 8.01$, $p < .01$	3.23 – strong
	14	Atomic Rules	.63	.21		
2005	20	Myers	.20	.10	$t(38) = -1.57$, $p = .06$	0.58 – moderate
	20	Atomic Rules	.27	.14		

Table 5-20: Productivity of the testers in terms of the number of boundary values derived over total time taken (efficiency – H_{02}/H_{12}).

Year	N	Approach	Mean Number of Correct Boundary Values Derived Per Unit of Time	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	.10	.08	$t(30) = 60$, $p = .28$	NA
	14	Atomic Rules	.08	.11		
2005	20	Myers	.09	.07	$t(38) = 1.20$, $p = .11$	NA
	20	Atomic Rules	.07	.77		

5.3.3.2 Total Time Taken

As the previous section shows, the number of test cases derived by the students was affected by whether they were given enough time to complete test case design. This can be assessed by counting the number of students who ran out of time during tutorials before finishing test case derivation. Significantly more students in the Atomic Rules group ran out of time before completing their work in both years (Table 5-21), indicating that using the Atomic Rules approach to test case design takes more time than Myers' approach.

Thus, had students been given more time to derive test cases, they may have been able to produce more complete test sets.

Table 5-21: Number of participants who ran out of time (efficiency– H_{02}/H_{12}).

Year	Approach	N	Out of Time (Count)	Out of Time (%)	Test of Two Proportions
2004	Myers	18	5	27.77	$\delta = -.4366$, $z = -2.45$, $p = .007$
	Atomic Rules	14	10	71.43	
2005	Myers	20	7	35	$\delta = -.6$, $z = -3.98$, $p < .001$
	Atomic Rules	20	19	95	

5.3.4 Errors Made (Accuracy) (H_{03}/H_{13})

To assess accuracy, the number of errors made by students during test case derivation was counted. Significantly fewer errors were made in the Atomic Rules group during EP equivalence class derivation in 2004 (Table 5-22). A similar result was seen in BVA boundary value derivation in 2004, although the result was just outside the 95% confidence interval (Table 5-24). No significant difference was found between the groups during BVA or EP test case derivation in 2004 (Table 5-23, Table 5-25) or during EP and BVA derivation in 2005 (Table 5-22 to Table 5-25). This suggests that the prescriptive nature of the Atomic Rules approach does not make the methods any harder to learn or to use correctly than Myers' approach.

Table 5-22: Errors made during EP equivalence class derivation (correctness – H_{03}/H_{13}).

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	20.81	374.50	$U = 48.5$, $p = .001$
	14	Atomic Rules	10.96	153.50	
2005	20	Myers	21.65	433	$U = 177$, $p = .274$
	20	Atomic Rules	19.35	387	

Table 5-23: Errors made during EP test case derivation (correctness – H_{03}/H_{13}).

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	15.56	280	$U = 109$, $p = .245$
	14	Atomic Rules	17.71	248	
2005	20	Myers	21.55	431	$U = 179$, $p = .292$
	20	Atomic Rules	19.45	389	

Table 5-24: Errors made during BVA boundary value derivation (correctness – H_{03}/H_{13}).

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	18.5	333	$U = 90$, $p = .0675$
	14	Atomic Rules	13.93	195	
2005	20	Myers	22.15	443	$U = 167$, $p = .192$
	20	Atomic Rules	18.85	377	

Table 5-25: Errors made during BVA test case derivation (correctness – H_{03}/H_{13}).

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	17	306	U = 117, p = .349
	14	Atomic Rules	15.86	222	
2005	20	Myers	20.70	414	U = 196, p = .463
	20	Atomic Rules	20.30	406	

5.3.5 Questions Asked (Learnability) (H_{04}/H_{14})

Participants were asked to document the questions they asked during tutorials. Only three students in 2004 and no students in 2005 recorded questions. Possible reasons could be that students:

1. were reluctant to ask questions,
2. did not have enough time to record questions, or
3. had a sound understanding of the methods taught.

Although it is hoped that the third option was the actual reason, not enough data was collected to clarify this. In future experiments students could be asked on a questionnaire whether they recorded any questions, and if not, why.

5.3.6 Satisfaction (H_{05}/H_{15})

Satisfaction was assessed through the Reflect and Review Questionnaire. Thirty-two students completed this in 2004 (100% of the class) and twenty-eight in 2005 (70% of the class).

In 2004, students were asked which model they would prefer to use in future and this was compared to the model they learned first (Table 5-26) (Murnane, Hall & Reed 2005). A chi-square test indicated if they had the opportunity, significantly more students would prefer to use the Atomic Rules approach in future. While it is possible that the students answered this question in favour of the Atomic Rules approach simply to show that they had an interest in the new approach to the teaching staff, since they were not told which of the two approaches were new and as they were told the responses would remain anonymous, it is hopeful that this did not bias their response.

Table 5-26: Approach students leant first versus approach they indicated they would use in future (n = 32) (satisfaction – H_{05}/H_{15}).

Year	Approach	Leant First	Use in Future	Chi-Square
2004	Myers	61%	9%	$\chi^2(1, N = 32) = 21.16, p < .001$
	Atomic Rules	39%	91%	

In 2005, a slightly different question was posed. Students were asked to rate the likelihood that they would use the models in future (Table 5-27) using a Likert scale of: 1 = very unlikely, 2 = somewhat

unlikely, 3 = neither likely nor unlikely, 4 = somewhat likely, 5 = very likely. However, the mean response was almost the same for both groups and no significant difference was found.

Table 5-27: Likelihood of using approaches in future (n = 28) (satisfaction – H_{05}/H_{15}).

Year	Approach	Model Learnt First	Model use in future (mean)		t-test
			Myers	Atomic Rules	
2005	Myers	46.43%	3.46	3.47	$t(26) = -.01$, $p = .307$
	Atomic Rules	53.57%	3.23	3.73	$t(26) = -1.12$, $p = .445$

5.3.7 Understandability (H_{06}/H_{16})

Three approaches for assessing test method understandability that were defined in Chapter 1 are:

1. to determine whether a tester understands the conditions under which a bb test case design method should be applied;
2. to determine whether a tester is able to apply the test method correctly; and
3. by assessing their self-rated understanding of the test method.

Although option 1 has been used in other software testing experiments (e.g. (Vegas 2004) used it to determine whether testers could identify the conditions under which one test method should be applied over another), it could not be used in this experiment since the students were given specific instructions to apply either Myers' representation for EP and BVA or the corresponding Atomic Rules. Option 2 has already been assessed under the hypotheses for effectiveness (see Section 5.3.2). Option 3 was addressed as follows.

On the Reflect and Review Questionnaire, the students were asked to rate their 'initial' and 'final' understanding of EP and BVA, using a using a Likert scale of: 1 = very poor, 2 = poor, 3 = average, 4 = good, 5 = very good, 6 = excellent (see Table 5-28 cols 2-5 and Table 5-29 cols 2-5). In both years, students reported that their understanding of EP and BVA had improved by the end of the experiment, indicating that their participation in the experiment had improved their understanding of these test methods (Table 5-28 cols 2-5 and Table 5-29 cols 2-5).

The students were also asked to rate their initial and final understanding of Myers' representation and the Atomic Rules approach, using a Likert scale of: 1 = very poor, 2 = poor, 3 = average, 4 = good, 5 = very good, 6 = excellent (see Table 5-28 cols 6-7 and Table 5-29 cols 6-7). In 2004, 57% of students rated their understanding of Myers' representation as below-average, while 100% rated their understanding of Atomic Rules as good or above (Table 5-28, cols 2-5). In 2005, 82% of students rated their understanding of Atomic Rules as Good to Excellent, compared to only 54% for Myers's representation (Table 5-29, cols 2-5). In both years, a significant difference was found in these ratings, where the mean was higher for Atomic Rules approach (in 2004, $t(30) = -7.65$, $p < .01$, while in 2005, $t(26) = -3.22$, $p = .03$). Thus, the students in both years reported that they were able to gain a better understanding of the Atomic Rules representation of EP and BVA.

Table 5-28: Self-rated understanding of test methods and representations in 2004 (n = 32) (Understandability – H_{06}/H_{16}).

	Understanding of Black-Box Testing Methods				Understanding of Representations	
	Initial		Final			
	EP	BVA	EP	BVA	Myers	Atomic Rules
Rating	**Percentages (%)**					
1. Very Poor	3	9	0	0	6	0
2. Poor	15	18	0	0	15	0
3. Average	**36**	**45**	0	3	**36**	0
4. Good	15	18	12	21	15	12
5. Very Good	24	9	**58**	**55**	18	**70**
6. Excellent	6	0	30	21	3	18
Frequency	**Values**					
Mean	3.61	3.00	5.18	4.94	3.35	5.03
Std Dev	1.27	1.06	0.64	0.75	1.25	0.56
Missing	*0*	*0*	*0*	*0*	*1*	*0*

Table 5-29: Self-rated understanding of test methods and representations in 2005 (n = 28) (Understandability – H_{06}/H_{16}).

	Understanding of Black-Box Testing Methods				Understanding of Representations	
	Initial		Final			
	EP	BVA	EP	BVA	Myers	Atomic Rules
Rating	**Percentages (%)**					
1. Very Poor	**32**	21	4	4	4	4
2. Poor	21	18	0	0	21	0
3. Average	29	**29**	7	11	21	11
4. Good	7	21	36	25	**29**	**46**
5. Very Good	11	7	**50**	**46**	25	29
6. Excellent	0	4	4	14	0	7
Frequency	**Values**					
Mean	2.43	2.86	4.39	4.54	3.50	4.22
Std Dev	1.32	1.38	.96	1.11	1.20	1.01
Missing	*0*	*0*	*0*	*0*	*0*	*1*

At the end of the experiment, the student's final understanding of EP and BVA was also compared by the approach they learnt first (i.e. Myers or Atomic Rules) to determine whether there was any relationship between these. No significant difference was found, indicating that the order in which they learnt the approaches did not affect their self-rated understanding.

Table 5-30: Affect of representation learnt on understanding of Equivalence Partitioning (Understandability – H_{06}/H_{16}).

Year	Approach	Model Learnt First	Final Understanding of EP (Mean)	t-test
2004	Myers	61%	5.05	$t(30) = .537$, $p = .30$
	Atomic Rules	39%	4.92	
2005	Myers	46.43%	4.31	$t(26) = -.42$, $p = .33$
	Atomic Rules	53.57%	4.47	

Table 5-31: Affect of representation learnt on understanding of Boundary Value Analysis (Understandability – H_{06}/H_{16}).

Year	Approach	Model Learnt First	Final Understanding of BVA (Mean)	t-test
2004	Myers	61%	5.20	$t(30) = .16$, $p = .43$
	Atomic Rules	39%	5.17	
2005	Myers	46.43%	4.23	$t(26) = -1.38$, $p = .09$
	Atomic Rules	53.57%	4.80	

5.4 Related Research

A number of researchers have evaluated testing methods taught at universities and compared the effectiveness of black-box test methods to other testing methods. In this section, these studies are discussed in terms of their approaches for assessing effectiveness and the numbers of participants included in these studies. This section also contributes to the continuing debate in the literature as to whether students should be used in software engineering experiments.

Roper et al. (1993) suggest that one way to progress towards a better understanding of test method effectiveness is to develop tighter definitions of the methods themselves, so experimental derivation of test data becomes predictable and repeatable. This would also allow deviations from the method to be detected unambiguously. This was one of the main aims in developing the Atomic Rules approach and one of the primary motivations behind assessing black-box test method usability. In the university experiments, learnability was examined in terms of the ease with which novice testers gained an understanding of particular concepts, and usability in terms of the satisfaction the students felt when using the two black-box test method representations. This included assessment of the completeness and correctness of test cases derived. Chen and Poon (2004) used similar measures when reviewing forty-eight student projects to identify the types of classifications students missed and the numbers and types of mistakes they made when using the Classification Tree Method. In another study of CTM, run with 104 students and rerun with fifty-eight students, participants tested programs they developed themselves, using whatever test methods they felt were appropriate (Yuen et al. 2004) (see Chapter 2). Their programs were graded by an automated test suite in terms the number of tests that resulted in correct output (Hoffman, Strooper and Walsh (1996) also used automated testing tools to grade student's work). Then, the students were taught CTM and asked to retest their programs with that method, to critically evaluate CTM, compare it to the test methods they

previously used and to rate their future preference of test methods. With the exception of the critical evaluation, these measures are similar to those that were used in the university experiments and to those that are used in the industry experiment (Chapter 6).

Fault-detection effectiveness has also been used as a measure of test method effectiveness. Basili and Selby (1987) conducted a seminal experiment with forty-two students (twenty-nine novices, thirteen intermediates) and thirty-two industry professionals, in which they compared the fault detection effectiveness, fault detection rate and classes of faults detected by three testing techniques: black-box testing (EP and BVA), white-box testing (100% statement coverage) and static testing (code reading by stepwise abstraction). Contrary to what may have been believed at the time, they found that the industry professionals were able to detect the most faults with code reading and did so at a faster rate. They were also able to detect more faults with black-box testing than white-box testing, but there was no difference in the rate at which they detected the faults. In one university group, the same numbers of faults were detected with code reading and black-box testing and both detected more faults than white-box testing, although the rate at which students detected faults did not differ for any method. Kamsties and Lott (1995) repeated this experiment with fifty students and found that while the fault-detection effectiveness of the two dynamic approaches (white-box and black-box) were comparable to that of the static approach (code reading), participants detected more faults using black-box testing. This experiment was also repeated by Wood et al. (1997) with forty-seven students. They found that the students detected similar numbers of faults for all three techniques, but their effectiveness depended on the nature of the program under test and the nature of program faults.

Thus, fault-detection effectiveness is a common basis for evaluating test method effectiveness. Unfortunately, this could not be measured in the university experiments discussed in this chapter, since programs were not used (i.e. only ‘toy’ specifications were used in the experiment, not working programs). On the other hand, this measure was used in the industry-based experiment that is discussed in Chapter 6. Another factor that was not covered in the experiments by Basili and Selby, Kamsties and Lott or Wood et al., is whether tester domain knowledge has any impact on test case effectiveness. This was also investigated in the industry-based experiment (see Chapter 6, Section 6.5.3).

Reid (1997) conducted an experiment that compared the probability that test cases derived by EP, BVA and RT would be capable of detecting specific classes of program faults. He noted that participants using black-box testing methods during experiments often select test cases that are not representative of other testers, therefore experiment results could not be generalised unless large enough groups of testers and test cases were used. Thus, Reid sought to derive every test case that satisfied the three black-box methods under study. In this study, BVA was found to be the most effective method, followed by RT and EP. However, it was important to note that BVA required twice as many test cases as EP to achieve higher levels of effectiveness (13.6 BVA tests compared to 7.6 EP tests), while RT required a “prohibitive” number of test cases in order to be as effective as BVA (50,000 RT test cases compared to 13.6 BVA tests).

The two university-based experiments discussed in this chapter involved a total of seventy-two students. This is comparable to subject numbers participating in other software engineering experiments. For

example, a software testing experiment that was initially run with thirty-six student participants was rerun by a different researcher with fifty-nine students and ninety-nine industry professionals who were paid standard consultancy rates (Arisholm & Sjøberg 2004). This relatively high number of industry participants may have been due to the remuneration. In another testing experiment, twelve industry professionals participated (Hungerford, Henver & Collins 2004) and they did not appear to be remunerated. Thus, remuneration may be an effective approach of obtaining more industry participation in future experiments. On the other hand, securing funding to pay for remuneration can be very challenging; (Arisholm & Sjøberg 2004) is one of the only papers to report paying standard consultancy rates for participation.

Carver et al. (2003) are of the opinion that running pilot experiments with students is effective preparation for industry-based experiments. Tichy (2000) supports this, stating that student experiments can be used to predict future trends in experiments that are rerun with industry professionals. Tichy (2000) also argued that graduate computer science students are only marginally different from industry professionals. Supporting this, Runeson (2003) reported the results of a study of the feasibility of using students as subjects in software engineering experiments, which compared the use of Humphrey's Personal Software Process (PSP) by undergraduate students, graduate students and industry professionals, to determine which group produced better estimations of program size and development effort, lower defect densities (i.e. defects per program size), lower defect intensities (i.e. defects per unit of time) and higher productivity. Runeson (2003) found significant differences between graduate and undergraduate students, but only small differences between graduates and industry professionals, supporting the view that graduate students are similar to industry professionals in terms of their capabilities. Thus, one negative aspect that has been reported on the use of students as experiment subjects is that results may not be generalised to industry professionals (Carver et al. 2003, Runeson 2003).

On the other hand, since the aim of the university experiments reported in this chapter were to assess the usability of black-box testing methods by novice testers, the use of undergraduate students was not seen as a disadvantage. If one finds that relatively inexperienced practitioners (students) can learn a method quickly, and the method produces equal or better results to existing approaches, and has other benefits such as auditability, then this is useful knowledge. Conversely, if there is no improvement, it cannot be concluded that professionals given 'proper' industrial training would not improve their testing practice. Also, these experiments were excellent preparation for an industry-based experiment that assessed the usability of the Atomic Rules approach by industry professionals, as they allowed the characteristics of test method usability to be explored and refined. They also facilitated an initial assessment of the threats to validity that need to be considered and controlled and have provided some indication of the response professional testers may experience when learning and using the Atomic Rules approach.

5.5 Discussion

Six hypotheses were defined for this experiment, corresponding to the six attributes of test method usability defined in Chapter 1: completeness (effectiveness) (H_{01}/H_{11}), efficiency (H_{02}/H_{12}), errors made (effectiveness – accuracy) (H_{03}/H_{13}), questions asked (learnability) (H_{04}/H_{14}), satisfaction (H_{05}/H_{15}) and understandability (H_{06}/H_{16}). The results for these are as follows.

Students who used the Atomic Rules approach during tutorials in 2004 produced significantly more *complete* EP equivalence classes and test cases than did those using Myers' approach. Thus, the null hypothesis for completeness could be rejected in favour of the alternate hypothesis for this attribute (Table 5-32). BVA results for *completeness* in 2004 were inconclusive (Table 5-32). On the other hand, students in Myers' group in 2005 produced more *complete* EP equivalence classes and test cases and BVA boundary values; thus, the null hypothesis could not be rejected for *completeness* in 2005 (Table 5-32). This was likely caused by the students in 2005 being given longer and more complicated specifications during tutorials and (more generally) by the Atomic Rules approach requiring more test cases to be derived per specification field (see Section 5.3.2). Supporting this view is that significantly more students in the Atomic Rules group in both years ran out of time before completing tutorial tests (see Section 5.3.3.2). Despite this, in 2004 the Atomic Rules group still achieved significantly higher EP coverage than those using Myers' approach (Table 5-32).

Setting a potentially overly-complicated test in the 2005 tutorials may have been caused by the "second system effect" (Brooks 1975), in which system engineers, having developed small, elegant solutions the first time around, design overly complicated solutions the second time; it is only by the design of a third system that the engineer will develop an effective solution that is not under or over designed. As students did well with the Atomic Rules approach in the 2004 tutorials, it was considered reasonable to increase the length and complexity of the specifications used in 2005. A third experiment using a complex and a non-complex specification in the one experiment could clarify whether specification complexity caused students in the Atomic Rules group to produce less complete test sets than Myers' group in 2005.

Nonetheless, in 2004 students in the Atomic Rules group produced more *accurate* answers during EP equivalence class design than Myers' group, in that they made fewer mistakes (Table 5-34). Thus, the null hypothesis for *completeness* and *accuracy* for EP equivalence class design could be rejected in favour of the alternate hypothesis. The results for the *completeness* hypotheses of EP test cases and for BVA in 2004 and 2005 were inconclusive (Table 5-34).

For *efficiency*, students using the Atomic Rules approach were found to be more productive in both years (assuming a 94% confidence interval in 2004), in that they derived more equivalence classes per hour than those using Myers' representation. Thus, the null hypothesis for *efficiency* of EP equivalence class derivation could be rejected in favour of the alternate hypothesis (Table 5-33). The results for *accuracy* of BVA derivation were inconclusive in both years (Table 5-33).

Learnability in terms of quantity of questions asked could not be assessed, as students did not ask or record questions during tutorials. Thus, the results for this hypothesis were inconclusive.

In assessing *satisfaction*, it was found that significantly more students in 2004 reported that they would prefer to use the Atomic Rules approach in future; thus the null hypothesis for this attribute in 2004 could be rejected in favour of the alternate hypothesis (Table 5-36). Supporting this, more students in 2005 chose to use the Atomic Rules approach in their class assignment and those that did achieved significantly higher assignment marks than those who used Myers' approach. After the experiment, the data was checked to

determine whether 'brighter' students chose to use the Atomic Rules approach in their assignment, in case they knew they could achieve a higher grade with that approach. A comparison of approach used to overall mean mark in the subject found no significant difference, indicating that intelligence levels did not affect representation choice on the assignment.

For *understandability*, in both years a significant difference was found in the student's self-rated understanding of the two approaches, where the mean was higher for Atomic Rules. Thus, students in both years felt that they had gained a better understanding of the Atomic Rules representation by the end of the experiment. Therefore, the null hypothesis for usability in both years could be rejected in favour of the alternate hypothesis (Table 5-37).

Student preference for lecturer did not appear to affect experiment results, as it would be fair to assume that the results of the two groups would have swapped for all hypotheses in 2005 if this was the case.

An observation that was made during data analysis was that the structure of the Atomic Rules approach can stifle tester creativity, even for novice testers. Since the Atomic Rules approach is more systematic than Myers' original definitions, it did not allow the students to derive test cases based on their own unique knowledge and experience. During data analysis, it became apparent that some testers in Myers' group created test cases that were not derivable from Myers' representation. As noted by Kaner et al. (1999), prior testing experience can be used to identify effective test cases through Error Guessing in similar testing scenarios, even if a tester cannot remember where they gained the domain knowledge. Systematic Method Tailoring was developed as an approach for guiding testers in the definition of new and reusable black-box test case design rules during prescriptive and non-prescriptive testing. However, this approach was not available when the first university experiment was run. Therefore, the industry-based experiment reported in the following chapter explores whether test case design rules used by industry professionals can be described as Atomic Rules.

Table 5-32: Outcomes of hypothesis testing for Completeness (H_{01}/H_{11}).

Hypothesis: Completeness ***H_{0x} = Null*** ***H_{1x} = Alternate***	**Equivalence Partitioning**				**Boundary Value Analysis**			
	2004		**2005**		**2004**		**2005**	
	Equiv. Classes	**Test Cases**	**Equiv. Classes**	**Test Cases**	**Boundary Values**	**Test Cases**	**Boundary Values**	**Test Cases**
H_{01}: The completeness of the black-box test set derived by novice testers is independent of the representation used.	Reject	Reject	Fail to reject	Fail to reject	Inconclusive	Inconclusive	Fail to reject	Inconclusive
H_{11}: Novice testers using the Atomic Rules approach derive a more complete test set compared to those using Myers' representation.	Accept	Accept	Fail to accept	Fail to accept	Inconclusive	Inconclusive	Fail to accept	Inconclusive

Table 5-33: Outcomes of hypothesis testing for Efficiency (H_{02}/H_{12}).

Hypothesis: Efficiency ***H_{0x} = Null*** ***H_{1x} = Alternate***	**2004** **Equiv. Classes**	**2004** **Boundary Values**	**2005** **Equiv. Classes**	**2005** **Boundary Values**
H_{02}: The efficiency of black-box test case derivation by novice testers is independent of the representation used.	Reject	Inconclusive	Reject (at 94% confidence)	Inconclusive
H_{12}: Novice testers using the Atomic Rules approach derive test cases more efficiently compared to those using Myers' representation.	Accept	Inconclusive	Accept (at 94% confidence)	Inconclusive

Table 5-34: Outcomes of hypothesis testing for Accuracy (H_{03}/H_{13}).

Hypothesis: Accuracy ***H_{0x} = Null*** ***H_{1x} = Alternate***	**2004**		**2005**		**2004**		**2005**	
	Equiv. Classes	**EP Test Cases**	**Equiv. Classes**	**EP Test Cases**	**Boundary Values**	**BVA Test Cases**	**Boundary Values**	**BVA Test Cases**
H_{03}: The number of errors made by novice testers during black-box test case derivation is independent of the representation used.	Reject	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive
H_{13}: Novice testers using the Atomic Rules approach make fewer errors during test case derivation compared to those using Myers' representation.	Accept	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive	Inconclusive

Table 5-35: Outcomes of hypothesis testing for Learnability (H_{04}/H_{14}).

Hypothesis: Learnability ***H_{0x} = Null*** ***H_{1x} = Alternate***	**Outcomes in 2004 and 2005**
H_{04}: The number of questions asked by novice testers during black-box test case derivation is independent of the representation used.	Inconclusive (no questions were asked by students)
H_{14}: Novice testers using the Atomic Rules approach ask fewer questions compared to those using Myers' representation.	Inconclusive (no questions were asked by students)

Table 5-36: Outcomes of hypothesis testing for Satisfaction (H_{05}/H_{15}).

Hypothesis: Satisfaction ***H_{0x} = Null*** ***H_{1x} = Alternate***	**2004**	**2005**
H_{05}: The preference of novice testers towards the black-box testing methods is independent of the representation used.	Reject	Inconclusive
H_{15}: Novice testers prefer to use the Atomic Rules approach for black-box test case derivation compared to the use of Myers' representation.	Accept	Inconclusive

Table 5-37: Outcomes of hypothesis testing for Understandability (H_{06}/H_{16}).

Hypothesis: Understandability ***H_{0x} = Null*** ***H_{1x} = Alternate***	**2004**	**2005**
H_{06}: A tester's understanding of black-box testing methods is independent of the representation used.	Reject	Reject
H_{16}: Novice testers rate the Atomic Rules approach to black-box test case derivation as easier to understand than Myers' representation.	Accept	Accept

5.6 Summary

The aim of the two university experiments discussed in this chapter was to compare the usability of Myers' original definition of EP and BVA to that of the Atomic Rules definition of the methods. While the results suggest that the Atomic Rules approach can improve the usability of EP and BVA, indicating that it could be a useful representation for teaching black-box testing methods to novice software testers, a number of inconclusive results indicate that further experimentation is required. Future research will ideally include repetition of these experiments, to determine whether the results reported here were by chance or whether they are indicative of how the Atomic Rules approach would be received by the wider software testing community.

In the next chapter, the results of an industry-based experiment are presented, which compared the usability and failure-detection effectiveness of the Atomic Rules approach to that of the black-box testing approaches that are used by testers in the software testing industry.

Chapter 6

Industrial Evaluation of the Atomic Rules Approach

"The thirteenth deadly sin is to leave the users to find the errors in your compiler."
P. J. Brown, 1979

6.1 Overview

In this chapter, the results of a two-day experiment examining the usability of the Atomic Rules representation of EP, BVA and ST are presented. The aim was to compare the usability and failure-detection effectiveness of black-box testing methods normally used by testers in the software testing industry with the Atomic Rules representations of EP, BVA and ST. Eleven testers working for a Queensland government organisation participated in the experiment. Their experience ranged from novice to expert. Although the experiment was carried out with practitioners from industry, due to the experiment design, it can be considered to be a controlled, classroom-based experiment.

An overview of the experiment is as follows. On day 1 the participants were randomly assigned to two groups of five and six people each (Table 6-1). The participants were then given 3.5 hours to derive and execute black-box test cases using whichever approach to black-box testing they felt was appropriate, which in this chapter is referred to as the 'Practitioner Normal Testing Practice' (PNTP) approach to test case derivation or 'PNTP testing.' During this time each participant tested one of two programs: a Batch Processor (assigned to Group 1) and an Address Parser (assigned to Group 2). These programs were developed by undergraduate students from La Trobe University and contained a variety of faults, including some that were intentionally seeded by a professional tester not involved in the experiment.

Once PNTP testing was complete, the participants were given a 2.5 hour presentation on the Atomic Rules representation of EP, BVA and ST. The groups were then crossed over and were given 3.5 hours to use the Atomic Rules approach to derive test cases for the 'other' program (i.e. the Address Parser was assigned to Group 1 and the Batch Processor to Group 2). This ensured that each participant tested a different program during each phase of testing, reducing the chances that what they learnt about the programs during PNTP testing would influence the results obtained during Atomic Rules testing. During this phase the testers were given lecture notes on the Atomic Rules approach and a two-page 'Quick-Reference Guide' that listed all Atomic Rules from EP, BVA and ST (Appendix C).

Table 6-1: The experiment plan.

Time	Day 1		Day 2	
09.00 – 10.30	Introduction Initial Questionnaire		Presentation (lecture) on the Atomic Rules representation of EP, BVA and ST (continued)	
10.30 – 11.00	*Morning tea*		*Morning tea*	
11.00 – 13.00	**Phase 1 - Group 1** Black-box testing Approach: PNTP Program: Batch Processor Post-Testing Questionnaire	**Phase 1 - Group 2** Black-box testing Approach: PNTP Program: Address Parser Post-Testing Questionnaire	**Phase 2 - Group 1** Black-box testing Approach: Atomic Rules Program: Address Parser Post-Testing Questionnaire	**Phase 2 - Group 2** Black-box testing Approach: Atomic Rules Program: Batch Processor Post-Testing Questionnaire
13.00 – 14.00	*Lunch*		*Lunch*	
14.00 – 15.30	Phase 1 testing continues	Phase 1 testing continues	Phase 2 testing continues	Phase 2 testing continues
15.30 – 16.00	*Afternoon tea*		*Afternoon tea*	
16.00 – 17.00	Presentation (lecture) on the Atomic Rules representation of EP, BVA and ST		Wrap-up discussion Reflect & Review Questionnaire	

An *Initial Questionnaire* was used to determine the group demographic and initial understanding of black-box testing methods. Post-testing questionnaires were used after PNTP and Atomic Rules testing to examine opinions of the test methods used. A *Reflect and Review Questionnaire* was used at the end of the experiment to examine the tester's final understanding of EP, BVA and ST and opinions of the test methods used. Other data collected included all test cases derived and failures detected.

The relatively small sample size of eleven participants means that the results cannot be generalised across the entire software testing industry. In addition, something that was (unfortunately) not identified prior to day one of the experiment was that the majority of the testers in the group did not have recent experience in test design (see Section 6.3.1.4). Nevertheless, the results provide insight into the way in which the testers who participated in our experiment conduct black-box testing and how they used the Atomic Rules representations of EP, BVA and ST to design test cases.

The remainder of this chapter is structured as follows. The experiment design is presented (Section 6.2), including hypotheses (Section 6.2.1), programs and specifications (Section 6.2.2), analysis approach (Section 6.2.4) and threats to validity (Section 6.2.5). Results are then presented (Section 6.3) and discussed (Section 6.4), following by a chapter summary (Section 6.5).

6.2 Experiment Design

The primary independent variable was the black-box testing approach used (i.e. PNTP and Atomic Rules). The other variable that was varied was the program tested (i.e. Address Parser or Batch Processor).

6.2.1 Hypotheses

Hypotheses were based on the following definition of test method usability (introduced in Chapter 1).

Test Method Usability. The extent to which a test case design method can be understood, learnt and used by software testers to achieve specified test case design goals effectively, efficiently and with satisfaction, within the context of applying software testing methods.

From this, the following null (H_{0X}) and scientific hypotheses (H_{1X}) (Christensen 2004) were defined.

Completeness (effectiveness):

H_{01}: The completeness of black-box test sets derived by industry-based testers is independent of the approach used.

H_{11}: Industry-based testers using the Atomic Rules approach derive more complete test sets in terms of EP, BVA and ST coverage compared to those using their own method for black-box test case design.

Failure-Detection Effectiveness:

H_{02}: There is no difference between the failure-detection effectiveness of the Atomic Rules approach compared to black-box testing approaches used by industry-based testers.

H_{12}: Industry-based testers detect more failures using the Atomic Rules approach then when using their own approaches to black-box test case design.

Efficiency (Productivity):

H_{03}: The efficiency of black-box test case derivation by industry-based testers is independent of the approach used.

H_{13}: Industry-based testers using the Atomic Rules approach derive test cases more efficiently compared to those using their own approach to test case design.

Errors Made (effectiveness – accuracy):

H_{04}: The number of errors made by novice testers during black-box test case derivation is independent of the approach used.

H_{14}: Industry-based testers using the Atomic Rules approach make fewer errors during test case derivation compared to those using their own approaches to test case design.

Understandability:

H_{05}: Learning the Atomic Rules approach has not affect on a tester's understanding of black-box testing methods.

H_{15}: Testers improve their understanding of black-box testing methods by learning the Atomic Rules approach.

Operability:

H_{06}: Testers using the Atomic Rules approach do not find it easy or difficult to use.

H_{16}: Testers using the Atomic Rules approach find it an easy to use.

Satisfaction:

H_{07}: The preference of industry-based testers towards the use of black-box testing methods is independent of the representation used.

H_{17}: Industry-based testers prefer to use the Atomic Rules approach for black-box test case design compared to using their own approaches.

Two additional hypotheses were defined. The first examines whether tester motivation levels threaten experiment validity (see Section 6.2.5.1). The second examines whether test case design rules used during PNTP testing can be described as Atomic Rules.

Motivation:

H_{08}: Testers feel more motivated when using a new technique simply because it is new.

H_{18}: Testers using the Atomic Rules approach do not find it more motivating to use simply because it is a new technique.

Test Method Representation:

H_{09}: Test case design rules used by experienced testers in industry when they are not prescribed a particular test method cannot be described by any black-box test method representation.

H_{19}: Test case design rules used by experienced testers in industry when they are not prescribed a particular test method can be described as Atomic Rules.

The validity of these hypotheses is explored in Section 6.3, together with interpretations. The only attribute of test method usability that was not assessed during this experiment is learnability in terms of how long it takes a tester to become competent with a test method, since the experiment did not permit long-term investigation. Learnability was also not assessed from the perspective of the types of questions asked during the experiment, as the participants were encouraged to work autonomously.

6.2.2 Programs and Specifications

Locating programs for use during the experiment proved to be one of the most challenging aspects of the experiment design. Three sets of programs were considered:

1. a program written by the participating organisation;
2. six C programs used in testing experiments by Kamsties and Lott (1995); and
3. mutated versions of C programs by Kamsties and Lott (1995), which were used in other (PhD) software testing experiments by Grindal (2007).

A number of unanticipated problems relating to the use of these programs were encountered. Privacy regulations prevented programs that were developed by the participating organisation (under option 1 above) from being installed at the training location. Furthermore, the programs mentioned under options 2 and 3 above were developed in C under Unix, and the testers were only experienced in the use of Windows, not Unix. An extensive search failed to locate a Windows-based C compiler that would compile the programs successfully (at least a dozen compilers were tested). Two weeks before the scheduled start of the experiment, it was realised that none of the programs under consideration could be used. An alternative needed to be sought, but time was a major issue. The scheduling of the experiment meant that the participant's work-load had been rescheduled and the training facilities had been paid for in full. Attempting to reschedule the experiment at this late stage may have caused the participant's employer to

withdraw support, leading to its cancellation. It was therefore decided that the date of the experiment could not be moved. Instead, alternate programs had to be sought.

In the end, two programs were located that could be compiled and executed under Windows. These were an Address Parser written in C++ by the author, and a Batch Processor written in Java by a student from the second university experiment discussed in Chapter 5. They were submitted as assignments under third year subjects at La Trobe University (*CSE31STM Software Techniques and Metrics* and *CSE32STR Software Testing and Reliability*). The primary requirement of the programs was to parse an input file and report any syntax errors. A secondary requirement was that they do so without failure (e.g. no endless loops, no unexpected terminations). All known failures in the programs were detected by applying all Atomic Rules from EP, BVA and ST (see Appendices D and E). These programs had also undergone black-box testing by their developers, using Myers' definitions of EP and BVA. Specifications of the program's input are provided in the subsections below, in a combination of Backus-Naur Form (BNF) and PL/I syntax. Table 6-2 describes the BNF symbols used.

Table 6-2: Definition of specification symbols.

Symbol	Actual Meaning
::=	Is defined as
∧	Represents one space
∧…	Represents one or more spaces
<ddd>	Represents three digits
[a | b]	Select one of the options (either a or b) contained within the square brackets [ ]
{<anything>}	The content of the curly braces { } is optional (i.e. <anything> is optional)
[a | b] MIN – MAX	The number of optional entities should be between MIN and MAX inclusive

6.2.2.1 The Address Parser

The Address Parser reads street addresses in a specific format (Figure 6-1) from an input file. Any addresses not in the specified format are rejected and error messages are written to a log file. The program contains a variety of faults (see Appendix D) but will not 'crash' when given any input (e.g. no endless loops, no unexpected terminations). The most common type of fault evident is that it produces incorrect error messages for certain types of input errors. For example, for the invalid address ***100 St Eltham* 3095**, which is missing the street name, the program would correctly detect the missing street name, but would incorrectly report that the suburb was invalid (Appendix D, Fault 8). Two additional faults were manually seeded into the program by an experienced tester not involved in the experiment, to ensure there were 'interesting' faults for participants to find (see Appendix D, Faults 6 and 7).

For simplicity, the 'state' name was not included in the Address Parser specification, as the domain of the application was limited to street addresses from the state of Victoria, Australia.

Figure 6-1: Input data specification of the Address Parser.

Standard addresses

<standard address> ::= <ddd> ∧ <street> ∧… <suburb> ∧… <postcode>.

Example

111 Main St Greensborough 3088.

Flat, unit or RSD addresses

<flat / unit / rsd address> ::=
[UNIT | FLAT | RSD] ∧ { <ddd> ∧ [, | /] ∧ } <ddd> ∧ <street> ∧… <suburb> ∧… <postcode>.

Examples

RSD 987 Main Street Greensborough 3088.

UNIT 100 / 220 Main Street Greensborough 3088.

Country address and care of addresses

<country address / care of address> ::= [C/- | C/o] ∧… <street> ∧… <suburb> ∧… <postcode>.

Example

C/- Main St North Greensborough 3088.

Street Syntax

<street> ::= <street_name> ∧… <street_type> {∧…<street_direction>}

<street_name> ::= [A – Z | a – z | -]1 - 40

<street_type> ::= [Street | St | Road | Rd | Avenue | Ave | Court | Crt | Grove | Grv | Lane | Ln | Place | Plc]

<street_direction> ::= [North | South | East | West]

Table 6-3: Valid suburbs-postcode pairs.

Suburb	Postcode		Suburb	Postcode
Altona	3018		Nutfield	3099
Box Hill North	3129		Ormond	3162
Coburg	3058		Osborne	3934
Collingwood	3066		Panton Hill	3759
Deer Park	3023		Phillip Island	3922
Eltham	3095		Rosebud	3939
Eltham North	3095		Rosebud South	3939
Fawkner	3060		Rosebud West	3940
Geelong West	3218		Seaford	3198
Greensborough	3088		Sorrento	3943
Healesville	3777		Torquay	3228
Ivanhoe	3079		Upfield	3061
Ivanhoe East	3079		Upwey	3158
Jacana	3047		Vermont	3133
Kangaroo Ground	3097		Vermont South	3133
Kew	3101		Victoria Park	3067
Lilydale	3140		Warrandyte	3113
Lovely Banks	3221		Warrandyte South	3134
Macleod	3085		Wattle Glen	3096
Melbourne	3000		Yan Yean	3755

6.2.2.2 The Batch Processor

The aim of the Batch Processor is to parse abstract representations of batches of sales records and calculate the total and average of values contained within each batch (Figure 6-2). Each batch starts and ends with an 'sbatch' (start batch) and 'ebatch' (end batch) tag. The last record of the file is 'lbatch.' This program was chosen from a set of forty similar programs. The version that was chosen contained a wide variety of defects, some that cause incorrect behaviour and others that caused the program to 'crash' (see Appendix 3). The specification for this program had a number of intentional ambiguities, and it was left to the implementer to resolve them. For example, it did not specify the maximum number of parts and part identifiers that could appear on a 'recordline' or the maximum number of recordlines.

Figure 6-2: Input data specification of the Batch Processor.

Input Specification

<input file> ::= <batch>* <nl> lbatch
<batch> ::= sbatch ∧… <batchid> <nl> <recordline>* <nl> ebatch ∧…<batchid>
<recordline> ::= { <partid> ∧ <value> ∧ , }* <partid> ∧ <value>
<batchid> ::=<digit><digit><char><char><char>
<partid> ::=<char><digit><digit><char>
<value> ::= [-99 – 98]
<digit> ::= [0 – 9]
<char> ::= [A – Z]
<nl> ::= newline

Output Specification

The program must calculate and output the total and average of:

a) the values of each individual partid within each batch,
b) all values within each batch,
c) the values of each individual partid over all batches,
d) all values over all batches.

Example Input

```
sbatch 11AAA
ebatch 11AAA
sbatch 63ABC
B22B -10
C99E 20, A11A 40, A11A -30
ebatch 63ABC
lbatch
```

6.2.3 Group Allocation

The participants were divided into two comparison groups (Table 6-4). Each group was given a different program to test during the two phases of testing, ensuring that what they learnt about the programs during PNTP testing did not affect their results during Atomic Rules testing.

Table 6-4: Group allocation.

Day	Testing Approach	Group	Participant Count	Program Tested
1	PNTP	1	5	Address Parser
		2	6	Batch Processor
2	Atomic Rules	1	6	Address Parser
		2	5	Batch Processor

6.2.4 Data Collection and Analysis Approach

Data collected consisted of test cases designed by participants during PNTP and Atomic Rules testing and answers to Initial, Post-testing and Reflect and Review Questionnaires. Standard tests for statistical significance (e.g. *t-tests* for parametric data, *chi-square* tests for non-parametric data) were used wherever possible. One-tailed tests were used, as the null hypothesis was assumed (e.g. that was no difference between PNTP and the Atomic Rules approach). A confidence interval of 95% was used for significance testing (i.e. any result with a p-value greater than 5% was rejected). A statistician and psychological researcher were consulted in the selection of all tests for statistical significance that were used during hypothesis testing.

6.2.5 Threats to Validity

In this section threats to internal and external validity are explored (these are similar to threats explored in Chapter 5; refer to section 5.3.4 for detailed definitions).

6.2.5.1 Internal Threats to Validity

Maturation. It was hoped that holding the experiment in a training room (Skillgate in Brisbane) and providing food and drink would mitigate boredom, fatigue and hunger. As the experiment was carried out over two consecutive days, changes in knowledge levels were unlikely. Questionnaires were used to determine if participant motivation influenced experiment outcomes (see Section 6.3.10).

Enthusiasm. To attempt mitigation of this threat (as with the student experiments), participants were not told that the Atomic Rules approach was new in the hope that this would allow them to focus on the work at hand, rather than considering how the new technique related to work conducted during PNTP testing. On the other hand, since the testers were learning a new approach to test case design, it was difficult to prevent enthusiasm from affecting their behaviour during that phase of the experiment. Therefore, data was collected on the level of motivation each tester felt before and after using PNTP and the Atomic Rules approach, enabling this to be built into the experiment design (see Section 6.3.10).

Instrumentation. Standard measurement scales were used during data analysis and only one person carried out analysis to ensure that the same approaches were followed throughout the experiment.

Regression. Selecting participants for inclusion in an experiment that have ‘extreme’ scores can cause them to perform better or worse on pre-tests than on post-tests (Creswell 2002). This threat could not be

controlled as the participants were chosen by the participating organisation, based on tester availability. An assessment of whether more experienced testers produced 'better' test cases was built into the experiment (see Sections 6.3.3.5 and 6.3.4.5).

Selection. To mitigate this threat, participants were randomly allocated to groups and the affect of participant experience was built into the experiment (see Sections 6.3.3.5 and 6.3.4.5).

Testing. This threat exists when participants are given the same test more than once and become familiar with the types of responses required. This was mitigated by assigning different programs to each participant during PNTP and Atomic Rules testing.

Procedure. If participants do not follow the procedures of a prescribed techniques this may affect results (Vegas et al. 2003). This was incorporated into the experiment by examining whether participants followed the procedures of the Atomic Rules approach correctly (see Section 6.3.7.4).

Reliability. Since the participants were only available for two days, the experiment could not be repeated to determine the reliability of the results. There was no way around this, so this threat could not be mitigated. Assumedly, repeating the experiment over time would give the participants more experience with the programs under test and with the Atomic Rules approach and that this would improve their results.

Copying. This was mitigated by conducting the experiment under 'exam' conditions, by asking the participants not discuss the experiment or the faults they detected until after their work was complete.

Population and Sample. As the sample of testers who participated in this experiment are not representative of all professional software testers, the results cannot be generalised across the entire population of professional testers. Nevertheless, they are considered to be indicative.

A number of internal threats to validity were not applicable, as follows.

History. This was not a threat as there was no time lapse between pre and post test measurements.

Diffusion of Treatments, Compensatory Equalization, Compensatory Rivalry, Resentful Demoralization and Method and Object Learning. These relate to experiments involving control groups. As this experiment involved a small number of participants, control groups could not be used.

6.2.5.2 External Threats to Validity

Language. All participants spoke English fluently and all written experiment materials were presented in English. As the specifications were written in BNF it was possible that the participants would not be able to understand them. To mitigate this threat, examples input files were provided.

Interaction of Setting and Treatment. Due to the small sample size, the results cannot be generalised across the entire population of professional testers. Thus, the results are considered to be indicative.

Programs and Techniques. If an experiment involves a small sample of programs or software testing methods, the results cannot be generalised to programs or methods (Vegas et al. 2003). In this experiment, data was collected for two programs that were implemented by third year university students and black-box test methods, EP, BVA and ST were used. Thus, broad generalisations have not been made across all specifications, programs or black-box testing methods.

Sample Heterogeneity. Differences in the age, gender, education or occupation of participants can affect individual results (Anastasi & Urbina 2007). This was built into the experiment by examining whether prior testing experience had any affect on test case effectiveness (see 6.3.3.5 and 6.3.4.5).

6.2.5.3 Construct Validity Threats

Measures. To mitigate this threat, participants were not told how data was going to be analysed.

6.3 Results

The experiment results are presented below, followed by a detailed discussion in Section 6.4.

6.3.1 Demographic

On the Initial Questionnaire, the participants were asked about their prior testing education and experience, programming and specification experience, understanding of black-box testing methods and level of motivation for participating in the experiment.

6.3.1.1 Industry Experience, Education and Training

All participants worked for the same government organisation in Queensland, Australia and 91% had worked as software testers for government organisations in the past (Table 6-5).

Table 6-5: Previous industries participants have tested software in (choose all that apply).

Prior Position	Count	Percent
Other – Government	10	91
Banking, finance & insurance	1	9
Education & training	0	0
Hotel, tourism, retail & trading	0	0
Manufacturing & engineering	0	0
Research & development	0	0
Software house & IT consultancy	0	0
Telecommunications	0	0

For their current role in testing, 46% were working as Testers and 46% as Test Leaders (Table 6-6).

Table 6-6: Participant's current role in testing (choose one).

Current Position	Count	Percent
Test Lead	5	46
Tester	5	46
Other[1]	1	9

In terms of the roles that participants held in the past, 91% had worked as Testers and 55% as Test Leads (Table 6-7). One participant had previous worked as a software developer.

Table 6-7: Prior roles held by the participants in testing (choose all that apply).

Prior Position	Count	Percent
Tester	10	91
Test Lead	6	55
Other	3	27
Test Manager	0	0
Test Specialist	0	0
Senior Test Consultant	0	0
Test Consultant	0	0
Test Analyst	0	0
Test Engineer	0	0

While 91% of the group stated that the received on the job training in software testing, 27% had attended an industry-based training course called Certified Software Test Professional (CSTP)[2], while one had received training in testing at TAFE (9%). No-one received training in software testing at university (Table 6-8), though this was not surprising as over half the group did not have postgraduate degrees and nobody had completed a 'pure' computing degree (Table 6-9).

Table 6-8: Prior software testing training (choose all that apply).

Training	Count	Percent
On the job training	10	91
In an industry training course	3	27
Other – Certified Software Test Professional (CSTP)	2	18
As part of a degree at TAFE	1	9
As part of a degree at university	0	0

[1] To ensure individuals cannot be identified (as required by the La Trobe University human ethics committee), the title of this position cannot be named.
[2] CSTP is a nationally recognised training course taught by K. J. Ross & Associates. The author is a presenter of CSTP training courses, but did not teach the course to any participants in this experiment.

Table 6-9: University and TAFE degrees completed (choose all that apply).

Degree	Count	Percent
None	6	55
Certificate IV Workplace Assessment & Training	2	18
Bachelor of Business (Accounting)	1	9
Diploma of Business	1	9
Diploma Frontline Management	1	9

The participants were asked how many years of software testing experience they had. Six (55%) reported less than 2 years of testing experience including one who was new to testing, while five (45%) had 2 to 10 years of experience. Thus, there was a variety of levels of experience, ranging from novice to expert.

6.3.1.2 Programs, Specifications and Test Methods

The participants were asked about the types of applications they tested in the past (Table 6-10). Most had experience with internet-based, real time and windows-style applications (63% each), while much fewer had tested text-based applications (27%).

Table 6-10: Types of applications tested in the past (choose all that apply).

Application Type	Count	Percent
Internet-based	7	64
Real time	7	64
Windows-based	7	64
Other – Mainframe	5	46
Text-based	3	27
Unix-based	1	9
Fault tolerant	1	9
Other – CSV, XML	1	9
Other – hardware	1	9

Participants were asked which specification languages they used in the past (Table 6-11). Most reported testing from informal specifications (82%). Since the specifications used in the experiment were written in BNF (see Section 6.2.2) and the testers indicated that they did not have experience with this language, there was expected to be a ‘learning curve’ during PNTP testing. As expected, the participants did take around 30 minutes to adjust. However, they were also given sample input files to reduce the learning curve.

Table 6-11: Familiarity with specifications languages (choose all that apply).

Specification Type	Familiar (Count)	Percent
Informal	9	82
Semi formal	3	27
Formal	0	0
Formal – BNF	0	0
Formal – Z	0	0
Formal – Object Z	0	0

6.3.1.3 Understanding of Black-Box Testing Methods

On the Initial Questionnaire the participants were rated their understanding of the following black-box testing methods (Table 6-12): Boundary Value Analysis (BVA), Cause-Effect Graphing (CEG), Decision Tables (DT), Equivalence Partitioning (EP), Error Guessing (EG), Exploratory Testing (ET), Model-Based Testing (MBT), Orthogonal Array Testing (OAT), Random Testing (RT), Specification-Based Mutation Testing (SBMT), Syntax Testing (ST) and Worst Case Testing (WCT). This was achieved using a Likert scale of: 1 = never heard of the method, 2 = none, 3 = novice, 4 = intermediate, 5 = advanced and 6 = expert. Although data was only required for EP, BVA and ST, they were asked about nine other methods to obtain an overall picture of their knowledge of black-box test methods. In a separate question, participants were asked how often they used the methods (Table 6-13), using a Likert scale of: 1 = never, 2 = rarely, 3 = occasionally, 4 = often, 5 = very often and 6 = always.

Few participants felt they had advanced knowledge of any black-box testing method and none felt they had expert knowledge. While 46% felt they had intermediate knowledge of EP and BVA, 54% had never heard of ST. This suggests that industry-based testers may not be experienced with prescriptive black-box testing methods. This is supported by a survey of software testing practices in Australian software development companies, which revealed that of the 65 organisations interviewed, 55% never used prescriptive black-box testing methods (Ng et al. 2004). Thus, while some testers in this experiment may have been inclined to use EP, BVA or ST during PNTP testing, they were not expected to do so in an 'expert' manner.

Interestingly, some participants reported having no experience with EG, ET, SBMT, OAT and ST (Table 6-12) but reported using them on occasion (Table 6-13). Since such inconsistencies were limited to 1 to 2 participants, they were not considered to impact the validity of the data detrimentally.

Table 6-12: Experience with black-box testing methods (choose one rating for each method).

	Black-Box Testing Methods											
	BVA	CEG	DT	EP	EG	ET	MBT	OAT	RT	SBMT	ST	WCT
Rating	Percentages (%)											
Never heard of method	18	27	27	18	18	9	**36**	**90**	**46**	**82**	**54**	**46**
None	0	**36**	9	0	18	9	27	0	27	9	18	9
Novice	9	0	18	9	27	27	9	0	9	0	9	18
Intermediate	**46**	27	27	**46**	**36**	**36**	27	9	18	9	9	18
Advanced	27	9	9	27	0	18	0	0	0	0	9	9
Expert	0	0	0	0	0	0	0	0	0	0	0	0
Missing	*0*	*0*	*9*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Table 6-13: Frequency of using black-box testing methods (choose one rating for each method).

	Black-Box Testing Methods											
	BVA	CEG	DT	EP	EG	ET	MBT	OAT	RT	SBMT	ST	WCT
Rating	Percentages (%)											
Never	9	**36**	**27**	27	9	9	**36**	**81**	**54**	**72**	9	**46**
Rarely	0	**36**	18	0	9	0	9	0	9	9	**54**	0
Occasionally	9	0	18	27	**27**	27	9	9	0	0	18	27
Often	**54**	27	**27**	**36**	**27**	**46**	27	9	27	18	18	27
Very often	18	0	0	0	9	9	9	0	0	0	0	0
Always	0	0	0	0	0	0	0	0	0	0	0	0
Missing	*9*	*0*	*9*	*9*	*18*	*9*	*9*	*0*	*9*	*0*	*0*	*0*

6.3.1.4 Frequency of Specification-Based Test Case Design

Participants were asked how often they derive test cases from specifications (Table 6-14). Unfortunately, it was realised that the majority of the participants had not had any recent experience in test case design. This became apparent during the completion of the Initial Questionnaire, when the participants reached the question "On average, what percentage of test cases do you derive from program specifications?" When a number of testers verbally reported to the author that they had not had any recent experience with test case design, and one explained that the majority of the test cases they executed were designed by Business Analysts, this author asked the testers to report on their questionnaire whether they designed their own test cases. Although 81% reported deriving 80 to 100% of tests from specifications, to the surprise of the research team, 72% reported that the majority of the test cases they execute are designed by Business Analysts. While the participants had recent experience with critiquing and executing test cases, they did not have recent experience with test case design. Unfortunately this was not identified prior to the experiment. This could explain why the participants conducted Exploratory Testing during PNTP testing, as this approach does not require the application of prescriptive test design methods (see Section 6.3.2).

Table 6-14: Percentage of tests derived from specifications (choose one).

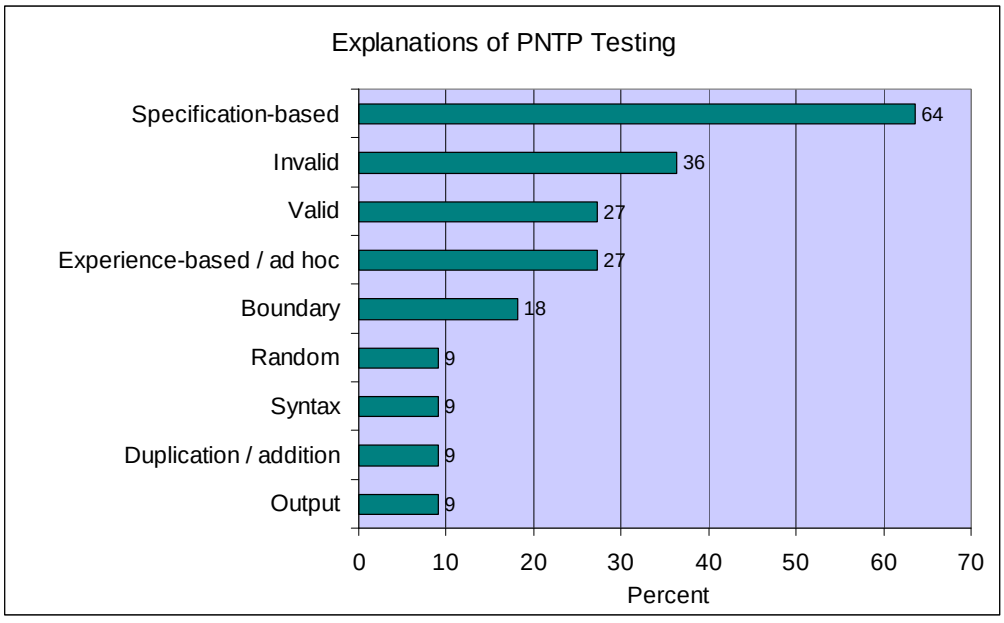
Range	Count	Percent
80 to 100%	9	81
20 to 39%	1	9
None	1	9
Less than 20%	0	0
40 to 59%	0	0
60 to 79%	0	0

6.3.2 Analysing the Practitioner Normal Testing Practice

The test cases derived during PNTP testing were analysed to identify the test case design approach used. This was carried out in two phases: first, by assessing the approach to test data and test case design the participants *reported* used (i.e. on the PNTP Post-Testing Questionnaire) and second, by analysing the approach they *actually* used.

When asked to describe the approach to testing they used during the PNTP phase (Figure 6-3), 27% of the group described their approach as 'ad hoc' or 'experience-based,' while 64% described it as 'specification-based.' Other popular responses were the use of valid (27%) and invalid (36%) test data values. Only 18% reported the use of boundary values and few reported testing program syntax (9%).

Figure 6-3: Explanations of the PNTP testing approaches used on day 1.



None of the participants reported doing any test planning or test case design prior to test execution, apart from one participant who (quite surprisingly) used the '*randbetween*' function in Microsoft Excel to implement a test data generation tool for testing the Batch Processor. Most participants verbally described starting the testing process by executing valid test cases and, as they began to understand how the programs worked, introducing small input errors to target specific types of faults, such as "boundary" faults or

"missing" (i.e. null) input errors. An analysis of the test cases they *actually* derived exemplified this, indicating that all participants conducted ad hoc Exploratory Testing during PNTP testing (even the participant who developed the automated testing tool followed it with some manual Exploratory Testing).

This suggests that testers may choose to use Exploratory Testing when they are unfamiliar with a program or when they have limited time for testing. As discussed in Chapter 2, Exploratory Testing allows testers to become familiar with a program while simultaneously executing test cases against it (see Section 2.2.5.2). It can be a very effective approach for defect detection that allows testers to become effective early in the testing process. Supporting this theory, on the Reflect and Review Questionnaire a number of participants reported that if they were to test the programs again, they would use more structured testing approaches, such as the use of Test Matrices, test planning and test design in advance of test execution, to ensure that "all areas of functionality were covered." This suggests that after using Exploratory Testing to gain an initial understanding of the program and the nature of faults present, testers may opt to use more prescriptive test case design methods.

6.3.3 Completeness (Effectiveness) (H_{01}/H_{11})

Since the Atomic Rules representation of EP, BVA and ST covers every published test case design rule from each method, it was considered to be a useful benchmark for measuring the completeness of test cases derived during PNTP and Atomic Rules testing. Completeness was assessed via test method coverage (see Section 6.3.3.1), individual Atomic Rule coverage (see Section 6.3.3.2), Atomic Rule class coverage (see Section 6.3.3.3) and specification coverage (see Section 6.3.3.4). The same data was also analysed to determine whether tester experience had any affect on test method coverage (see Section 6.3.3.5).

6.3.3.1 Test Method Coverage

The mean percentage of EP (Table 6-15), BVA (Table 6-16) and ST (Table 6-17) coverage achieved during PNTP and Atomic Rules testing was compared. A paired sample t-test was used to test for a significance difference.

A significant difference was found in the BVA and ST test cases, where the mean was higher for Atomic Rules and Cohen's effect size was strong. A significant difference was not found for EP, as the means were relatively even. This suggests that while the testers may have been skilled with EP prior to the experiment, learning Atomic Rules enabled them to increase their skill with BVA and ST test design.

Table 6-15: Mean percentage of coverage of Atomic Rules from EP.

N	Approach	Mean (%)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	13.31	3.62	$t(10) = -.226$, $p = .413$	N/A
11	Atomic Rules	13.89	8.62		

Table 6-16: Mean percentage of coverage of Atomic Rules from BVA.

N	Approach	Mean (%)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	3.02	1.73	$t(10) = -2.59$, $p = .01$	Strong
11	Atomic Rules	8.99	6.83		

Table 6-17: Mean percentage of coverage of Atomic Rules from ST.

N	Approach	Mean (%)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	1.79	0.81	$t(10) = -2.50$, $p = .01$	Strong
11	Atomic Rules	9.29	10		

The relatively low percentages of coverage achieved by the testers (see column 3 of Table 6-15, Table 6-16 and Table 6-17) was due to the large quantities of test data and test cases that were required to cover all specification fields with all applicable Atomic Rules from EP, BVA and ST (Table 6-18). Also, the ratio of tests derivable for the two programs was almost 1:2, with fewer tests being required to cover all fields of the Batch Processor (see Table 6-18, column 4).

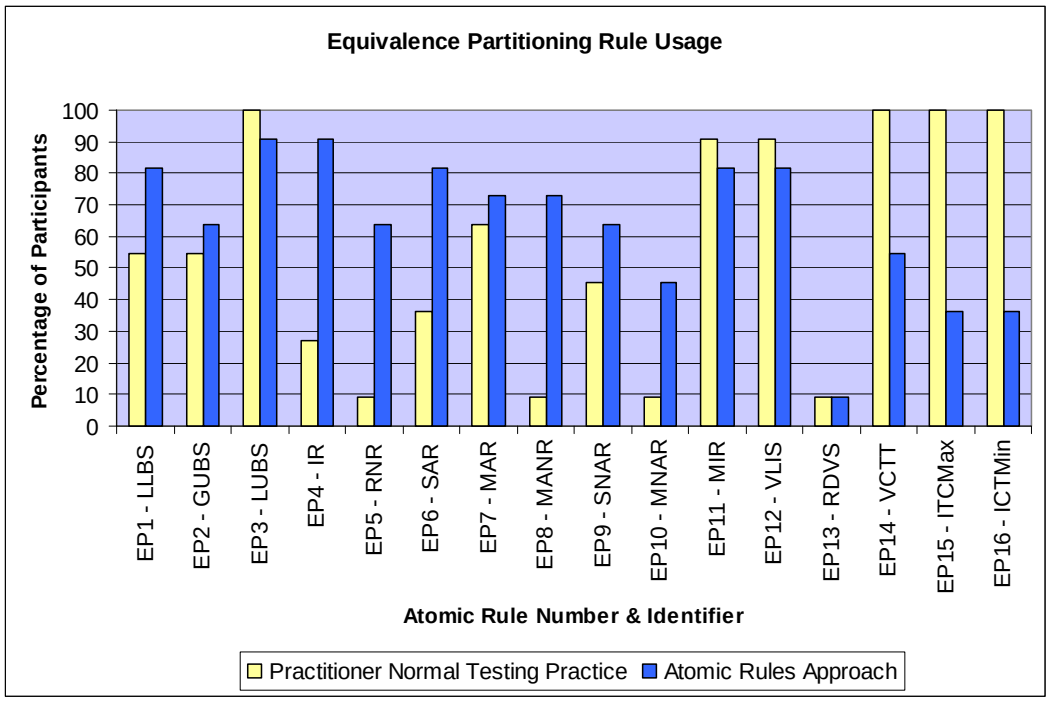
Table 6-18: Number of test data values and test cases derivable by applying EP, BVA and ST to all specification fields.

Test Method	Number of Test Data Values and Test Cases Derivable		Ratio
	Batch Processor	Address Parser	
Equivalence Partitioning	161	420	1:2.61
Boundary Value Analysis	109	231	1:2.11
Syntax Testing	295	700	1:2.37
Total	**565**	**1,351**	**1:2.39**

6.3.3.2 Atomic Rule Coverage

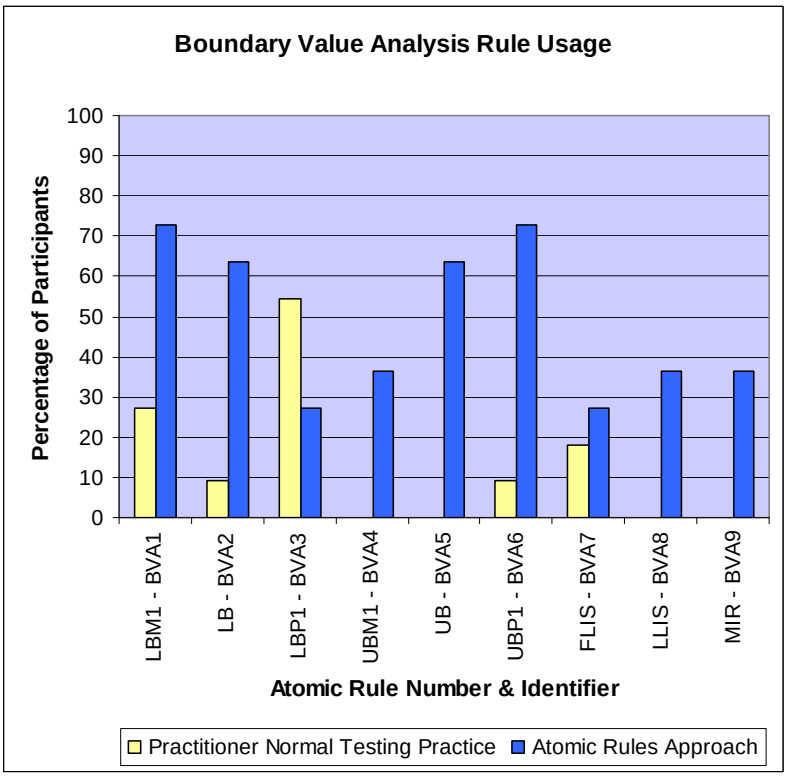
The percentage of Atomic Rules from EP (Figure 6-4), BVA (Figure 6-5) and ST (Figure 6-6) that were used at least once by any participant during PNTP and Atomic Rules testing were compared. As Figure 6-4 illustrates, 56% of EP Atomic Rules were used by more participants during Atomic Rules testing than PNTP testing. Although Test Case Construction Rules *EP14* to *EP16* were used by more participants during PNTP testing, analysis of the tests derived during Atomic Rules testing found that 55% of the group did not have enough time to derive complete test cases (i.e. they used many DISRs, DSSRs and DIRMs but few TCCRs). Given more time, it is possible that the testers would have achieved higher levels of EP coverage during Atomic Rules testing.

Figure 6-4: Comparison of mean percentage of EP coverage (graphical view).



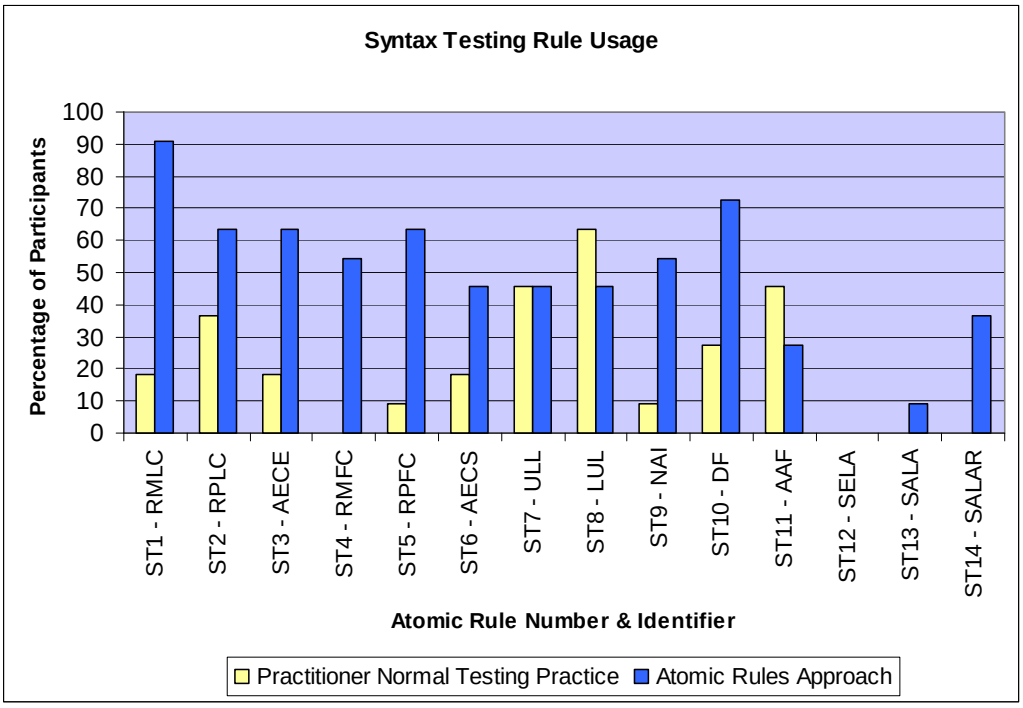
For BVA (Figure 6-5), all Atomic Rules other than *BVA3* (Lower Boundary + Selection) were used at least once by more participants during Atomic Rules testing. This indicates that learning the Atomic Rules approach increased the group's skill with achieving adequate boundary value coverage.

Figure 6-5: Comparison of mean percentage of BVA coverage.



A similar result was found for ST (Figure 6-6), where all but two rules (*ST8* and *ST11*) were used at least once by the same or greater numbers of participants during Atomic Rules testing. This indicates that learning the Atomic Rules approach also increased the group's skill with ST.

Figure 6-6: Comparison of mean percentage of ST coverage.



6.3.3.3 Classes of Atomic Rules Covered

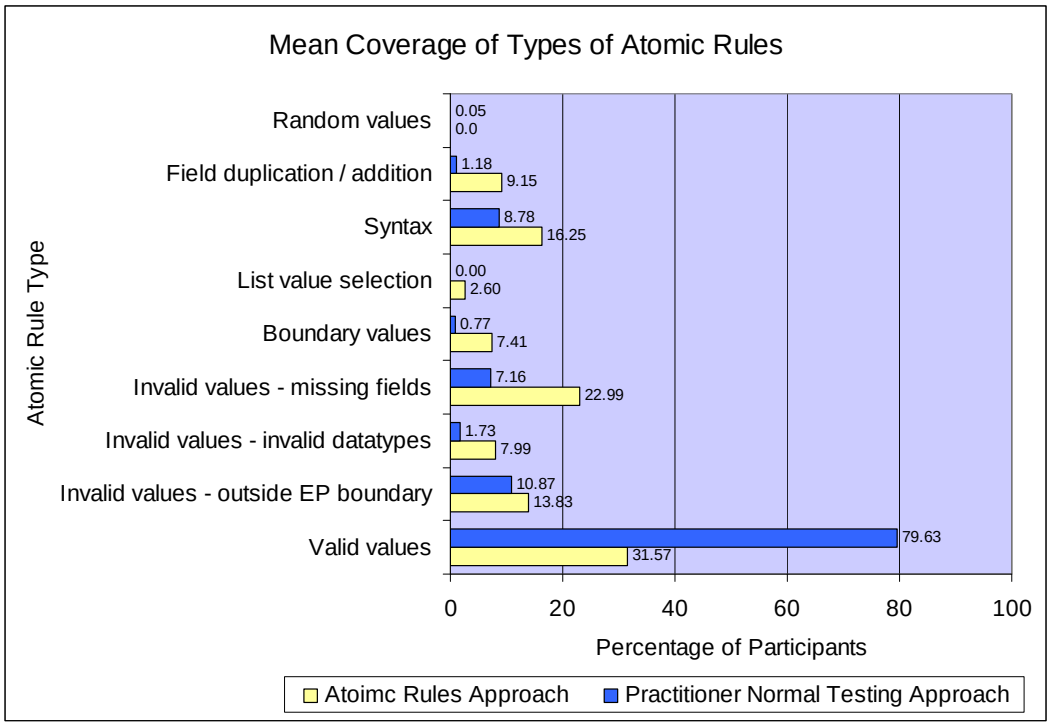
Atomic Rule usage was analysed from the perspective of the classes of test data that can be selected by certain types of rules (Figure 6-7). The following test data classes were analysed:

1. valid values (covers EP3 and EP12);
2. invalid values – outside EP boundaries (i.e. partitions above or below valid partitions) (covers EP1 and EP2);
3. invalid values – invalid datatypes (e.g. alphas, integers, non-alphanumeric) (covers EP4 to EP10);
4. invalid values – missing fields (i.e. testing with null) (covers EP11 and BVA9);
5. boundary values (on, just above and just below field boundaries) (covers BVA1 to BVA8);
6. list value selection (i.e. selecting a valid value from a list) (covers EP12);
7. syntax (i.e. testing input fields with valid and invalid syntax) (covers ST1 to ST8); and
8. field addition and duplication (i.e. adding or repeating fields) (covers ST10 and ST11).

These classes were analysed by examining the mean coverage achieved by the group, where rules within each class could be applied to many different input fields of each program. As Figure 6-7 illustrates, the

testers achieved higher mean coverage of all rule classes during Atomic Rules testing, with the exception of rules that select valid test data. Thus, their ability to derive more diverse classes of test data improved after learning Atomic Rules, suggesting that this approach is useful for broadening a tester's knowledge of the classes of black-box test data that can be derived.

Figure 6-7: Comparison of mean Atomic Rule coverage achieved per test data class.



6.3.3.4 Specification Coverage

Completeness was also analysed by the mean coverage of input fields in the Address Parser (Table 6-19, Figure 6-8) and Batch Processor (Table 6-20, Figure 6-9). This analysis revealed that there were classes of input fields that were not well tested. Spaces and non-alphanumeric (i.e. special) characters were poorly covered during both PNTP and Atomic Rules testing. Also, participants did not detect the seeded fault in the Address Parser that allowed a forward slash or hyphen to be accepted in place of the period at the end of the address (see Section 6.3.4.7 for a further discussion). These results could suggest that the participants might not have been experienced with testing special character fields. On the other hand, since they were taught how to test non-alphanumeric fields with the Atomic Rules approach (e.g. many Atomic Rules from EP and ST apply to non-alphanumeric fields) and as they stated that they were experienced in recognising the need to test punctuation (see Section 6.3.4.3), it is possible that they either did not recognise the need to test these fields, they did not know how to test them or they did not have enough time to test them.

Other fields in the Address Parser that were not well covered include street name length, street type and street direction during PNTP Testing and street contents during Atomic Rules testing.

With the exclusion of the space field in the Address Parser, the group achieved equal or higher levels of coverage of all input field types in both programs during Atomic Rules testing (Table 6-19 and Figure 6-8 for Address Parser, Table 6-20 and Figure 6-9 for Batch Processor). The mean coverage during PNTP testing when it was applied to the Address Parser was 9.89%, compared to 47.54% by the application of the Atomic Rules approach (Table 6-19). Similarly, for the Batch Processor the mean coverage achieved during the PNTP testing phase was 10.84%, compared to 36.83% during Atomic Rules testing (Table 6-20). This suggests that testing teams as a whole will produce more complete test sets with the Atomic Rules approach than during Exploratory Testing (since most participants used this approach during PNTP testing, as discussed in Section 6.3.2).

Table 6-19: Percentage of Atomic Rules applied by at least one participant during testing of the Address Parser (tabular view).

Field	PNTP (%)	Atomic Rules (%)	Difference (%)
Address Type (UNIT FLAT RSD C/o C/-)	15.71	55.48	+39.76
House / Unit Number (ddd)	18.14	52.23	+34.09
Spaces (single or multiple)	5.25	5.24	-0.02
Street Name (A-Z, A-z, -)	4.23	16.06	+11.82
Street Name Length (1-40)	7.22	49.38	+42.17
Street Type	9.21	69.74	+60.53
Street Direction	7.94	62.54	+54.60
Suburb	13.07	63.39	+50.32
Postcode	12.35	85.59	+73.25
Special Characters (. , /)	5.80	15.77	+9.97
Mean	***9.89***	***47.54***	***+37.65***

Figure 6-8: Percentage of Atomic Rules applied by at least one participant during testing of the Address Parser (graphical view).

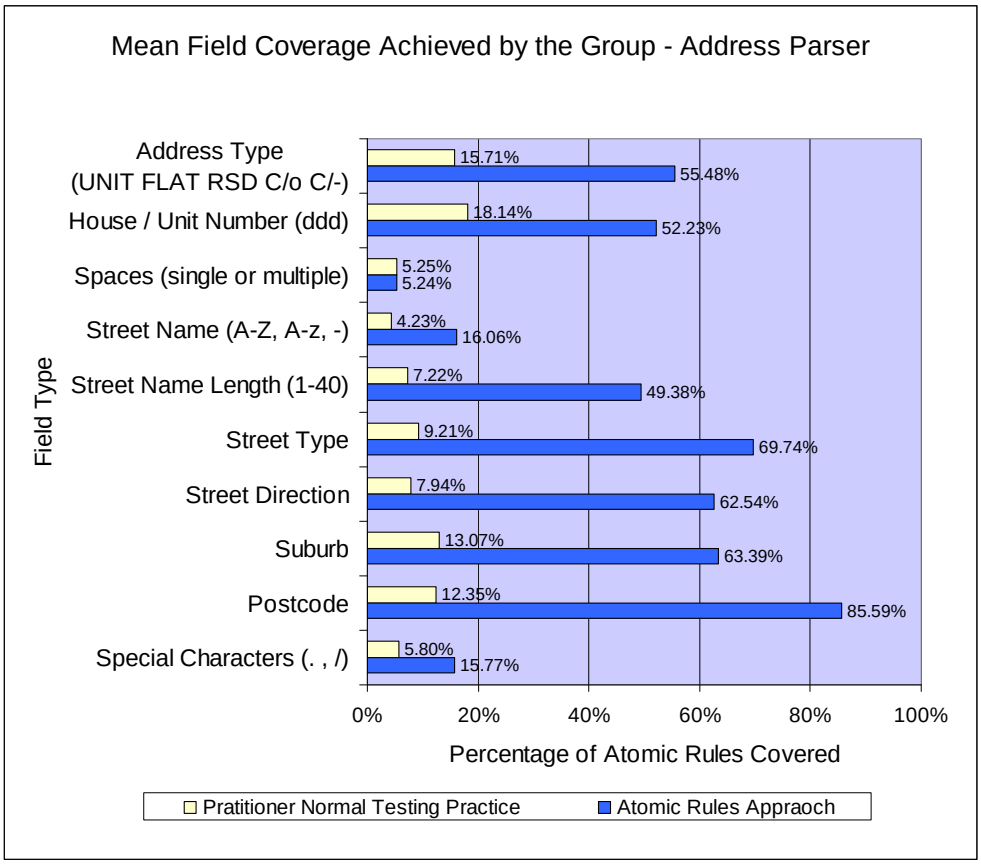
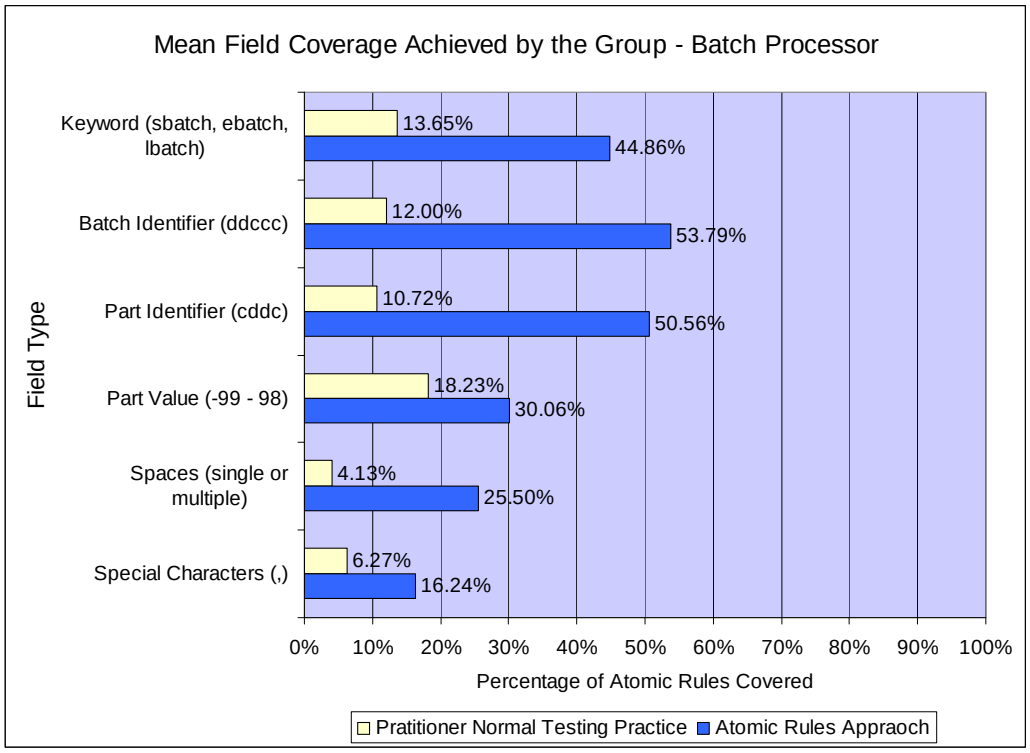


Table 6-20: Percentage of Atomic Rules applied by at least one participant during testing of the Batch Processor (tabular view).

Field	PNTP (%)	Atomic Rules (%)	Difference (%)
Keyword (sbatch, ebatch, lbatch)	13.65	44.86	+31.20
Batch Identifier	12.00	53.79	+41.79
Part Identifier	10.72	50.56	+39.83
Part Value (-99 - 98)	18.23	30.06	+11.82
Spaces (single or multiple)	4.13	25.50	+21.37
Special Characters (,)	6.27	16.24	+9.97
Mean	***10.84***	***36.83***	***+26.00***

Figure 6-9: Percentage of Atomic Rules applied by at least one participant during testing of the Batch Processor (graphical view).



6.3.3.5 The Effect of Experience on Test Method Coverage

To determine whether prior testing experience had any affect on EP, BVA or ST coverage, the mean test method coverage achieved during each phase of the experiment was compared by two factors: years of testing experience and current role in testing.

To assess the impact of *years of testing experience*, coverage was compared for participants who had less than 2 years of experience (6 participants) to those who had 2 to 10 years experience (5 participants). No significant difference was found in the mean EP or ST coverage achieved during PNTP testing (Table 6-21). While a significant difference was found in the mean BVA coverage, it was under 2%, which was not considered conclusive (Table 6-21). This suggests that the number of years of testing experience does not affect a tester's ability to cover the test case design rules of EP, BVA or ST during Exploratory Testing.

Table 6-21: Coverage achieved during PNTP testing, by experience in years.

Method	N	# Years Experience	Mean (%)	Std Dev	t-test	Cohen's Effect Size
EP	6	Less than 2 years	13.78	2.73	$t(9) = .43$, $p = .34$	N/A
	5	2 to 10 years	12.74	4.76		
BVA	6	Less than 2 years	3.85	2.03	$t(9) = 1.96$, $p = .04$	strong
	5	2 to 10 years	2.03	0.36		
ST	6	Less than 2 years	1.76	0.93	$t(9) = -.11$, $p = .46$	N/A
	5	2 to 10 years	1.82	0.76		

In addition, there was no significant difference found in the mean EP, BVA and ST coverage during Atomic Rules testing (Table 6-22), suggesting that number of years of testing experience also does not affect a tester's ability to use the Atomic Rules approach. On the other hand, since the mean EP coverage was just outside the 95% confidence interval, Cohen's effect size was strong (Table 6-22) and the mean was higher for the Atomic Rules approach, it is possible that testers with more experience could find the Atomic Rules representation of EP easier to use than less experienced testers. However, as these EP results were not within the significance range, they cannot be considered to be conclusive.

Table 6-22: Coverage achieved during Atomic Rules testing, by experience in years.

Method	N	# Years Experience	Mean (%)	Std Dev	t-test	Cohen's Effect Size
EP	6	Less than 2 years	10.08	6.72	$t(9) = -1.77$, $p = .06$	strong
	5	2 to 10 years	18.47	9.02		
BVA	6	Less than 2 years	7.51	4.54	$t(9) = -.77$, $p = .23$	N/A
	5	2 to 10 years	10.77	9.14		
ST	6	Less than 2 years	7.98	11.67	$t(9) = -.46$, $p = .32$	N/A
	5	2 to 10 years	10.86	8.59		

The mean EP, BVA and ST coverage was also compared by *current role in testing*. There were five Test Leads and five Testers. No significant difference was found during PNTP testing (Table 6-23) or Atomic Rules (Table 6-24) testing, suggesting that a tester's role does not affect their ability to derive test cases with either Exploratory Testing or the Atomic Rules approach. One participant, who was not working as a test lead or a tester, has been excluded from this comparison as this would have uniquely identified them, which was against rules agreed with the La Trobe University human ethics committee.

Table 6-23: Comparison of coverage achieved during PNTP testing by current testing role.

Method	N	Job Position	Mean (%)	Std Dev	t-test	Cohen's Effect Size
EP	5	Test Lead	12.09	4.36	$t(8) = -.894$, $p = .19$	N/A
	5	Tester	14.25	3.20		
BVA	5	Test Lead	2.36	1.24	$t(8) = -1.67$, $p = .07$	N/A
	5	Tester	4.03	1.85		
ST	5	Test Lead	1.68	.80	$t(8) = -.83$, $p = .21$	N/A
	5	Tester	2.10	.80		

Table 6-24: Comparison of coverage achieved during Atomic Rules testing by current testing role.

Method	N	Job Position	Mean (%)	Std Dev	t-test	Cohen's Effect Size
EP	5	Test Lead	12.09	4.36	$t(8) = .269$, $p = .39$	N/A
	5	Tester	14.25	3.20		
BVA	5	Test Lead	10.10	9.21	$t(8) = .80$, $p = .22$	N/A
	5	Tester	6.57	3.46		
ST	5	Test Lead	5.67	5.02	$t(8) = -1.33$, $p = .10$	N/A
	5	Tester	14.06	13.09		

6.3.4 Failure-Detection Effectiveness (H_{02}/H_{12})

An alternate approach to assessing the effectiveness of a test case design method is by the proportion of faults detectable by the method (Reid et al. 1999), as follows.

$$\text{Fault detection effectiveness} = \frac{\text{number faults detected}}{\text{total number of faults}} \text{ X } 100 \quad (6.1)$$

Since one fault can cause more than one failure and as this formula requires knowledge of the total number of faults within a program, which might only be known once the program has been operating in production for some time (Reid et al. 1999), this measure can be inaccurate when assessing black-box test method effectiveness. Since the Address Parser and Batch Processor were not developed for use in industry, the total number of faults cannot be determined. Thus, an alternate approach is to calculate failure-detection effectiveness for known program failures, as follows.

$$\text{Failure-detection effectiveness} = \frac{\text{number failures detected}}{\text{total number of failures}} \text{ X } 100 \quad (6.2)$$

To identify the *total number of known failures* in the Address Parser and Batch Processor, every Atomic Rule from EP, BVA and ST was applied to all applicable input fields by the author of this thesis. 'One-to-one' test cases were derived by applying Atomic Rules *EP14* and *EP15*, resulting in 1,351 test cases for the Address Parser and 565 for the Batch Processor; a grand total of 1,916 test cases (see Section 6.3.3.1) (in a one-to-one test case, each test data value is covered by exactly one test case (BS 7925-2)). The Address Parser required more than twice the number of tests as the Batch Processor because it had more input fields and each field contained more alternate values. Twenty-seven unique failures were identified in the Address Parser and twenty-four in the Batch Processor (see Appendices D and E).

As observed by Dijkstra (and stated in the introduction), "Program testing can be used to show the presence of bugs, but never to show their absence!" (Dijkstra 1969). Accordingly, it may be possible that latent faults exist in the two programs that were not detected as part of this testing, and as such, are not considered in the total number of known failures. On the other hand, given the rigour of the Atomic Rules approach, the author is confident that these figures are reflective of the total number of failures in the two programs. Nonetheless, an interesting question for future research is whether the Atomic Rules approach is 'complete' in the sense that it can detect the majority of input/output validation errors in any program.

6.3.4.1 'Ultimate' Failure-Detection Effectiveness

Once the total number of known failures was identified, the 'ultimate' failure-detection effectiveness of EP, BVA and ST could be calculated (using equation 6.2), by analysing the percent of program failures that were detectable when every single test selection rule from all three methods is applied to every possible input field (Table 6-25) (this test design and execution was carried out by the author). Surprisingly, the Atomic Rules representation of EP was capable of detecting 93% of the known program failures in the Address Parser, but only 54% of failures in the Batch Processor. BVA was detecting 67% of failures in the

Address Parser and 58% in the Batch Processor. ST was equally effective against both programs, detecting 67% of known failures in each program. In future work, it would be interesting to determine whether programs that utilise certain function libraries or have specific design styles (e.g. object oriented or functional) benefit from the application of a particular black-box testing method.

Table 6-25: 'Ultimate' failure-detection effectiveness of EP, BVA and ST when the methods are applied completely to the Address Parser and Batch Processor.

Program	# Input Fields	# Known Failures	Test Method	# Tests Derived	Failures Detected		Failure-detection effectiveness (Percent %)
					#	%	
Address Parser	42	27	EP	420	25	93%	93%
			BVA	231	18	67%	67%
			ST	700	18	67%	67%
Batch Processor	20	24	EP	161	13	54%	54%
			BVA	109	14	58%	58%
			ST	295	16	67%	67%

All known failures were detected by applying all Atomic Rules from EP, BVA or ST, with the exception of one failure in the Batch Processor (Appendix E, Failure 24), which was detected by a participant who applied *ST14* in a technically incorrect but effective way. *ST14* selects a test data value containing all values from a list in the opposite order to which they are specified. For example, if it is applied to the *UNIT/FLAT/RSD* field of the Address Parser, it would select the invalid test data value *RSDUNITFLAT*. One participant applied this rule to the *recordline* field of the Batch Processor to reverse the order of the *partid* and *value* fields, resulting in invalid test data *50 A11A*. The definition of ST in the British Computer Society's Component Testing Standard includes a ST rule that substitutes one field for another (BS 7925:2). This rule that was accidentally left out of the set of Atomic Rules presented to the participants. A new Atomic Rule for this has been defined (see ST19 in Appendix B).

6.3.4.2 Mean Failure-detection effectiveness

Once all known failures were identified, the failure-detection effectiveness of tests derived by the participants could be compared for PNTP and Atomic Rules testing (Table 6-26). A significant difference was found, where the mean was higher for Atomic Rules testing (10% more failures were detected on average). This suggests that the Atomic Rules approach enables testers to detect more failures than Exploratory Testing (since that approach was used by the participants during the PNTP testing phase).

Table 6-26: Comparison of the failure-detection effectiveness achieved by the participants.

N	Program	Mean Failure-detection effectiveness (Percent %)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	25	10	$t(10) = -1.85$, $p < .05$	moderate
11	Atomic Rules	35	12		

6.3.4.3 Affects of Application Domain Knowledge on Failure-detection effectiveness

After the experiment was complete, the participants explained that they were more familiar with the logical concept of the Address Parser, as it was based on concepts they use in real life, and because they tested address-based inputs on a regular basis in their current jobs. They explained that address correctness was critically important to their organisation, since their customers not only relied on postal communication, they also often demanded high levels of accuracy in their addresses (e.g. in one case, a disgruntled customer wrote a letter to a state government minister to complain about the misplacement of a comma in their postal address). Conversely, the participants stated that the Batch Processor was much more 'abstract,' as it was not based on concepts they were familiar with.

Thus, to determine whether differences in the "application solution domain knowledge" (Reed 1990) in the programs under test affected the participant's ability to detect failures in them, the mean percentage of failures detected with each testing approach was compared by the program tested (Table 6-27). A significant difference was found during the PNTP testing phase, where the mean was higher for the Address Parser. Conversely, a significant difference was not found during Atomic Rules testing.

Table 6-27: Comparison of failure-detection effectiveness achieved against each program.

Testing Approach	N	Program Tested	Mean failure-detection effectiveness (Percent %)	Std Dev	t-test	Cohen's Effect Size
PNTP	5	Address Parser	34	4	$t(9) = 5.16$, $p \leq .001$	Strong
	6	Batch Processor	18	6		
Atomic Rules	6	Address Parser	30	8	$t(9) = -1.28$, $p = .11$	N/A
	5	Batch Processor	40	15		

Likewise, when the mean percentage of failures detected in each program were compared by the testing approach used to detect the failures (Table 6-28), no significant difference was found for the Address Parser, whereas a significant difference was found for the Batch Processor, where the mean was higher during Atomic Rules testing and where Cohen's effect size was strong.

Table 6-28: Alternate comparison of failure-detection effectiveness achieved against each program.

Program Tested	N	Testing Approach	Mean failure-detection effectiveness (Percent %)	Std Dev	t-test	Cohen's Effect Size
Address Parser	5	PNTP	34	4	$t(9) = .41$, $p = 0.2$	N/A
	6	Atomic Rules	30	6		
Batch Processor	6	PNTP	18	8	$t(9) = 3.02$, $p < .05$	Strong
	5	Atomic Rules	40	15		

These results suggest that in order to test a program effectively (i.e. to detect failures), testers either need application solution domain knowledge in the program (if they are conducting Exploratory Testing) or they need to use a prescriptive black-box testing method to ensure that their test case design is rigorous.

6.3.4.4 Individual Failure-detection effectiveness

The failure-detection effectiveness achieved by each participant was also analysed (Table 6-29, Figure 6-10, Figure 6-11 and Figure 6-12). Participants in Group 1, who tested the Address Parser using their PNTP testing approach and the Batch Processor using the Atomic Rules approach, detected roughly the same number of failures during both phases of testing (Figure 6-10). Conversely, testers in Group 2 who tested the Batch Processor during PNTP testing and the Address Parser during Atomic Rules testing dramatically increased their failure-detection effectiveness during Atomic Rules testing (Figure 6-11). This strengthens the theory that testers either need application solution domain knowledge in the program under test or a prescriptive testing method to be effective. Furthermore, testers with both can be even more effective.

Table 6-29: Failure-detection effectiveness of each participant.

Group #	Participant #	Approach	Program Tested	Failure-detection effectiveness	
				Count	Percent
1	1	PNTP	Address Parser	10	37
		Atomic Rules	Batch Processor	6	25
	2	PNTP	Address Parser	9	33
		Atomic Rules	Batch Processor	8	33
	3	PNTP	Address Parser	10	37
		Atomic Rules	Batch Processor	5	21
	4	PNTP	Address Parser	10	37
		Atomic Rules	Batch Processor	10	42
	5	PNTP	Address Parser	7	26
		Atomic Rules	Batch Processor	7	29
2	6	PNTP	Batch Processor	2	8
		Atomic Rules	Address Parser	8	30
	7	PNTP	Batch Processor	4	17
		Atomic Rules	Address Parser	8	30
	8	PNTP	Batch Processor	4	17
		Atomic Rules	Address Parser	18	67
	9	PNTP	Batch Processor	6	25
		Atomic Rules	Address Parser	9	33
	10	PNTP	Batch Processor	5	21
		Atomic Rules	Address Parser	13	48
	11	PNTP	Batch Processor	4	17
		Atomic Rules	Address Parser	8	30

Figure 6-10: Failure-detection effectiveness achieved by participants in Group 1.

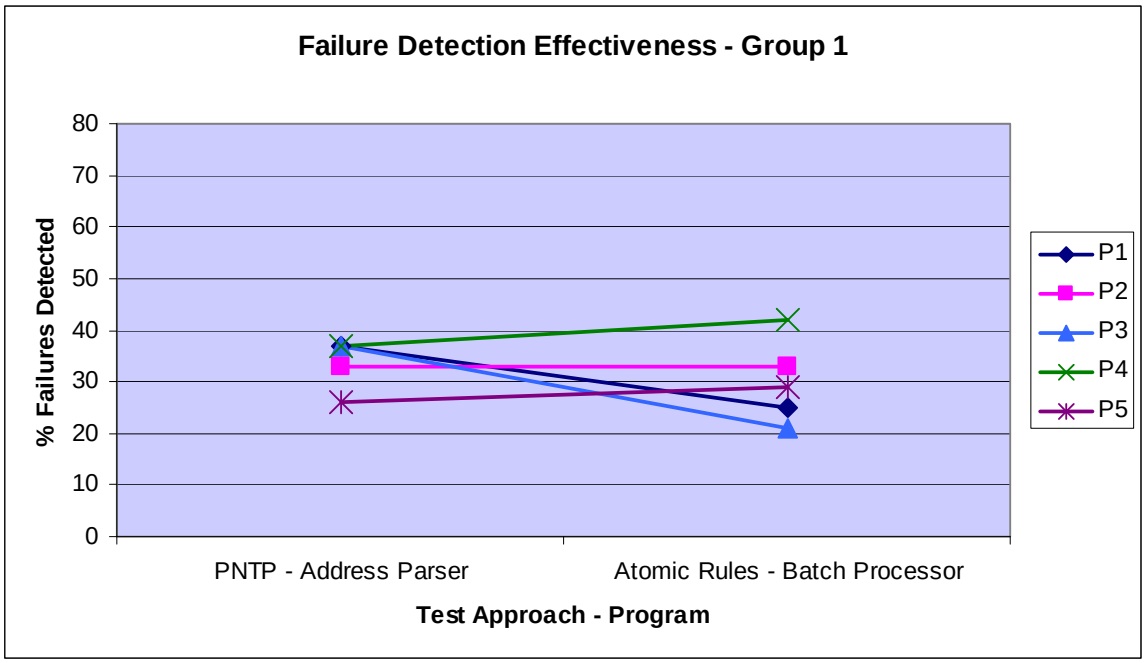


Figure 6-11: Failure-detection effectiveness achieved by participants in Group 2.

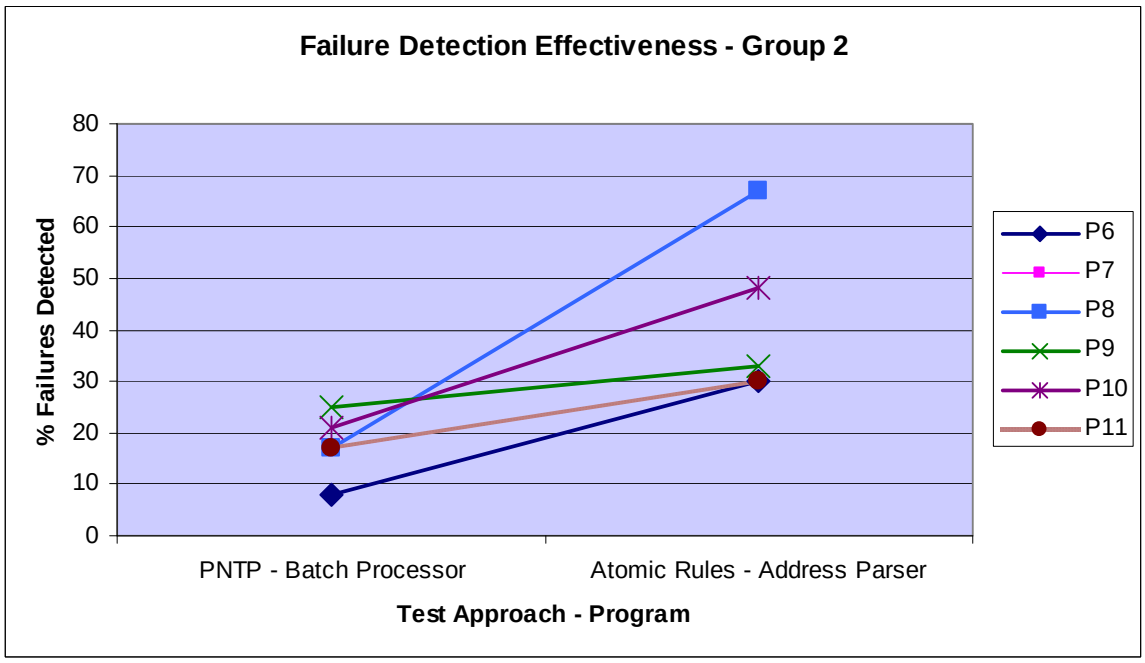
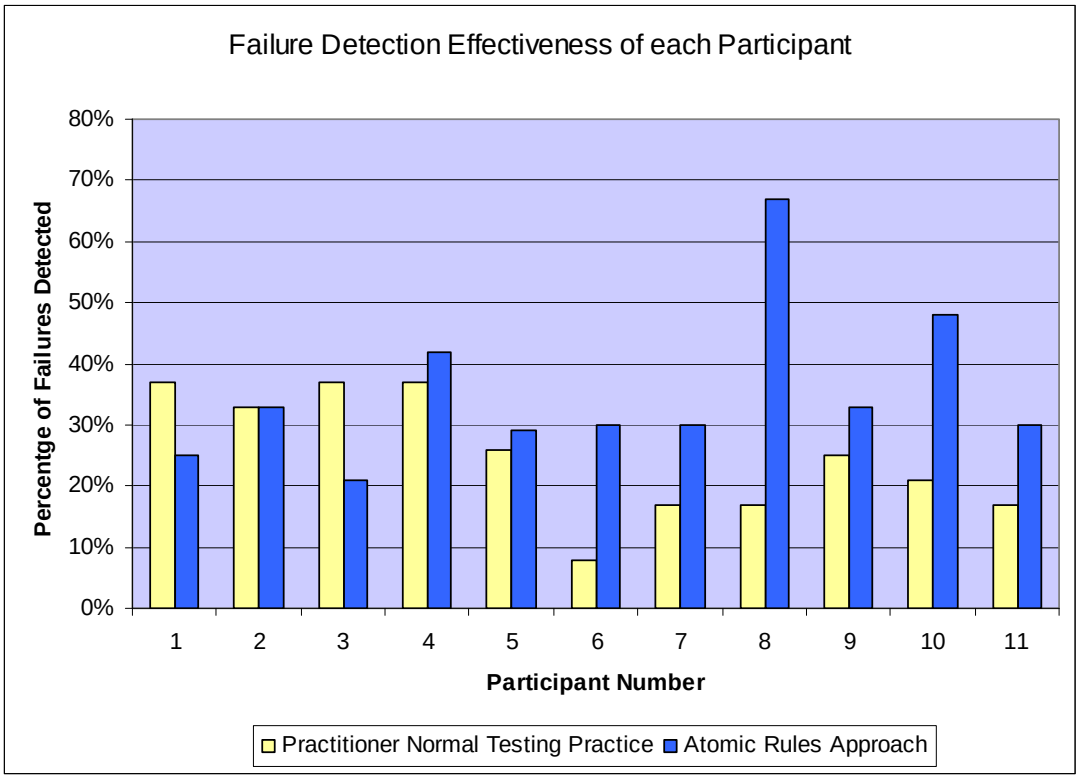


Figure 6-12: The failure-detection effectiveness achieved by each participant.



6.3.4.5 Affects of Experience on Failure-detection effectiveness

To determine whether prior testing experience affected failure-detection effectiveness, these results where compared using two factors: number of years of testing experience and current role in testing.

No significant difference was found when failure-detection effectiveness was compared by number of years of testing experience (Table 6-30). This suggests that the duration of time spent working in the software testing industry alone does not affect a tester's ability to detect failures, whether they use Exploratory Testing or a prescriptive testing method.

Table 6-30: Comparison of failure-detection effectiveness by tester experience in years.

Approach	N	# Years Experience	Mean (%)	Std Dev	t-test	Cohen's Effect Size
PNTP	6	Less than 2 years	22	10	$t(9) = -.97$, $p = .18$	N/A
	5	2 to 10 years	28	10		
Atomic Rules	6	Less than 2 years	35	8	$t(9) = .01$, $p = .49$	N/A
	5	2 to 10 years	35	18		

On the other hand, a significant difference was found in the failure-detection effectiveness achieved during the PNTP testing phase, where the mean was higher for those who were currently working as Testers (Table 6-31). This suggests that a person's role in testing has more impact on their failure-defection effectiveness than the length of time they have spent working in the software testing industry. This could be

because Testers would likely have more recent experience in test writing, whereas Test Leads would have more experience in the strategising, planning and managing of testing. Furthermore, no significant difference was found during Atomic Rules testing, suggesting that learning the Atomic Rules approach could fill the knowledge gap for Test Leads who are not currently involved in test design, allowing them to become as effective at detecting program failures as the Testers in their team.

Table 6-31: Comparison of failure-detection effectiveness by current role in testing.

Approach	N	Job Position	Mean (%)	Std Dev	t-test	Cohen's Effect Size
PNTP	5	Test Lead	0.19	.09	$t(8) = -1.97$, $p < .05$	strong
	5	Tester	0.30	.09		
Atomic Rules	5	Test Lead	0.41	.16	$t(8) = 1.42$, $p = .10$	N/A
	5	Tester	0.30	.08		

6.3.4.6 Failure-Detection Effectiveness of Individual Atomic Rules

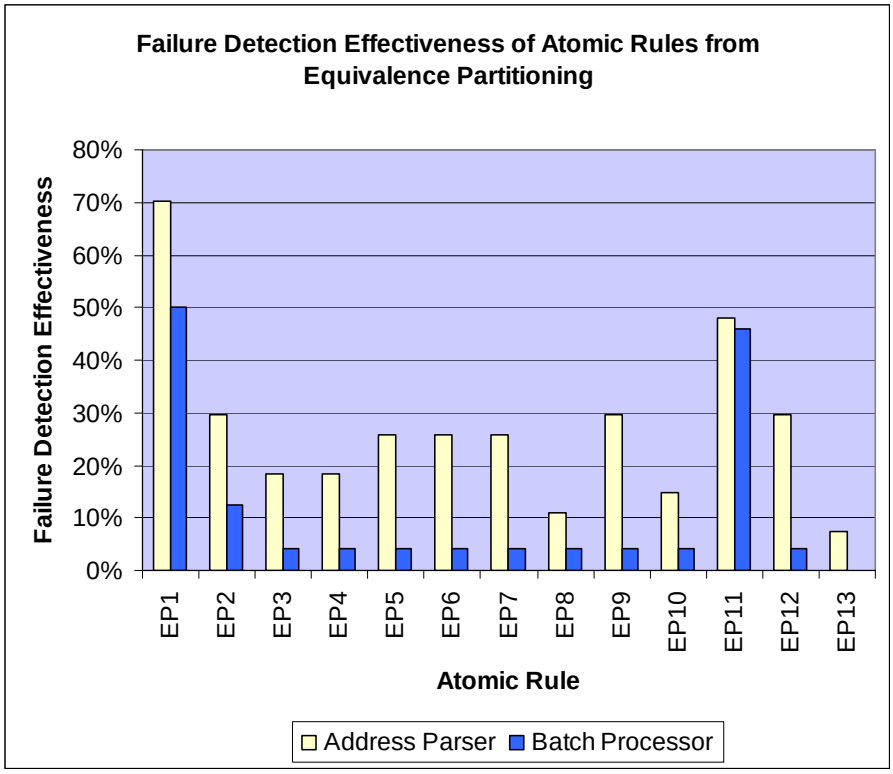
To identify whether particular Atomic Rules had a better chance of detecting failures than others, the 'ultimate' failures-detection effectiveness of Atomic Rules from EP (Table 6-32, Figure 6-13), BVA (Table 6-33, Figure 6-14) and ST (Table 6-34, Figure 6-15) was examined. Only DSSRs, DISRs and DIMRs were included in this analysis, since TCCRs cannot be analysed in isolation, because they are reliant on the other three rule types.

Atomic Rules from EP with the highest failure-detection effectiveness were *EP1* (select partition below lower boundary), detecting 70% of Address Parser failures and *EP11* (replace field with null), detecting 48% of Address Parser failures (Table 6-32). Invalid datatypes selection rules *EP4* to *EP10* were moderately effective against the Address Parser, detecting 11% to 30% of failures. Conversely, invalid datatype selection rules *EP4* to *EP10* were less effective against the Batch Processor, detecting only 4% of failures. This indicates that the Address Parser suffered from more faults that were related to invalid datatypes not being rejected correctly by the program than the Batch Processor.

Table 6-32: Failure-detection effectiveness of Atomic Rules from Equivalence Partitioning (tabular view).

Atomic Rule	Failure-Detection Effectiveness (Percent %)	
	Address Parser	Batch Processor
EP1	70	50
EP2	30	13
EP3	19	4
EP4	19	4
EP5	26	4
EP6	26	4
EP7	26	4
EP8	11	4
EP9	30	4
EP10	15	4
EP11	48	46
EP12	30	4
EP13	7	0

Figure 6-13: Failure-detection effectiveness of Atomic Rules from Equivalence Partitioning (graphical view).



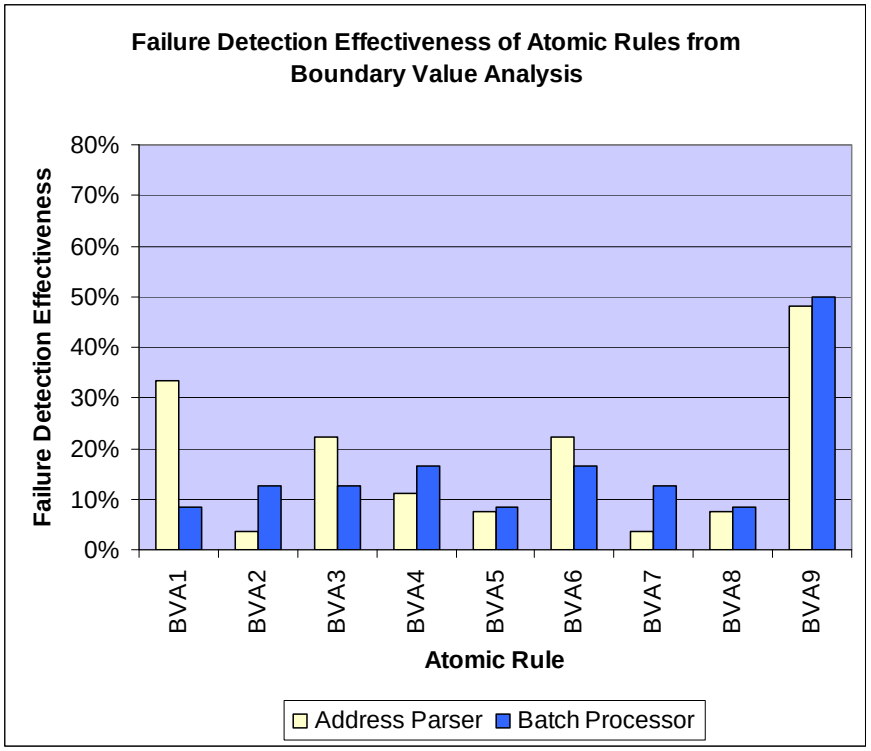
The most effective BVA test design rule was *BVA9* (replace field with null), detecting 48% of Address Parser failures and 50% of Batch Processor failures (Table 6-33, Figure 6-14). This is followed by *BVA1* (select value below lower boundary), which detected 33% of Address Parser failures but only 8% of Batch

Processor failures. Other BVA rules were only mildly effective, detecting 4% and 22% of failures in the two programs, indicating that they did not suffer from many boundary-related faults.

Table 6-33: Failure-detection effectiveness of Atomic Rules from Boundary Value Analysis (tabular view).

Atomic Rule	Failure-detection effectiveness (Percent %)	
	Address Parser	Batch Processor
BVA1	33%	8%
BVA2	4%	13%
BVA3	22%	13%
BVA4	11%	17%
BVA5	7%	8%
BVA6	22%	17%
BVA7	4%	13%
BVA8	7%	8%
BVA9	**48%**	**50%**

Figure 6-14: Failure-detection effectiveness of Atomic Rules from Boundary Value Analysis (graphical view).



The most effective ST rule was *ST1* (remove last character of keyword), detecting 63% of Address Parser failures (Table 6-34, Figure 6-15). The next most effective rules against this program were *ST3* (add extra character to keyword), detecting 37% of failures, *ST6* (remove first character of keyword) detecting 33% of failures and *ST5* and *ST2* (replace first and last characters of keywords) detecting 30% each. Given the effectiveness of these rules against the Address Parser, which consisted largely of keyword-based fields,

this suggests that Atomic Rules that add or remove characters from keywords are more effective for testing programs whose input consists of keywords of a very specific format.

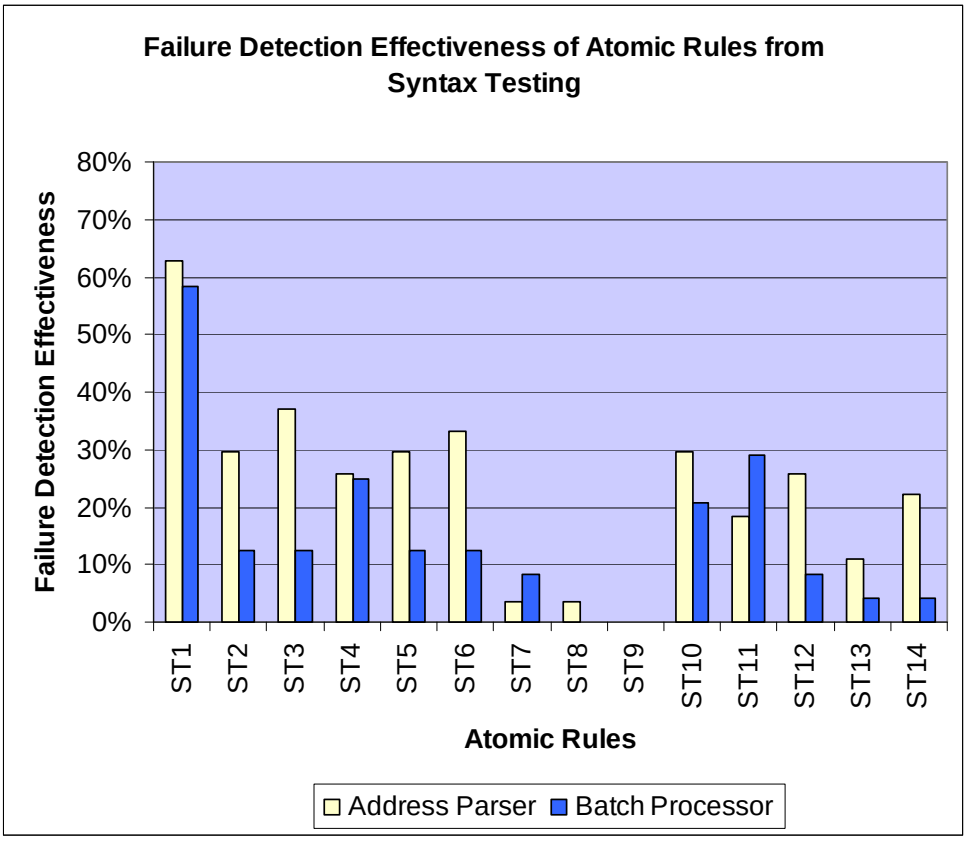
The most effective ST rule against the Batch Processor was also *ST1*, detecting 58% of failures, followed by *ST11* (add a field) detecting 29% of failures, *ST4* (remove first character of keyword) detecting 25% of failures and *ST10* (duplicate field) detecting 21% of failures. This was not surprising, since this program predominantly suffered from faults related to additional invalid input fields not being detected.

The most ineffective ST rule was *ST9* (null all input), detecting no failures. This is not surprising, since it derives only one test case and, consequently, one failure at most can be detected by that rule. *ST7* and *ST8* (change case of alphabetical letters) were also ineffective, indicating that the programs handled these types of inputs well. Other ST rules that were ineffective against the Batch Processor were *ST12*, *ST13* and *ST14*, which test list-based fields. This is also not surprising since this program consists of few list-based input fields, whereas the Address Parser processed many different list-based inputs. Accordingly, programs with list-based input fields would benefit from being tested with these Atomic Rules.

Table 6-34: Failure-detection effectiveness of Atomic Rules from Syntax Testing (tabular view).

Atomic Rule	Failure-detection effectiveness (Percent %)	
	Address Parser	Batch Processor
ST1	63	58
ST2	30	13
ST3	37	13
ST4	26	25
ST5	30	13
ST6	33	13
ST7	4	8
ST8	4	0
ST9	0	0
ST10	30	21
ST11	19	29
ST12	26	8
ST13	11	4
ST14	22	4

Figure 6-15: Failure-detection effectiveness of Atomic Rules from Syntax Testing (graphical view).



6.3.4.7 Detection of Seeded Faults

Two faults were intentionally seeded into the Address Parser by an independent professional tester who was not directly involved with the experiment, to ensure the program contained from some 'interesting' faults. The experiment's subjects and the author were not aware of which faults had been seeded until after the experiment was complete. The seeded faults were as follows:

- 'Fault A' caused the program to reject the valid suburb/postcode pair *'Ivanhoe East 3079'* (see Appendix D, Fault 7); and
- 'Fault B' allowed the program to accept a forward slash or hyphen in place of the period '.' at the end of the address (see Appendix D, Failure 6) (these characters sit either side of the period in the ASCII table).

Both faults were detectable with Atomic Rules from EP, BVA and ST, if the methods were applied 'completely' (i.e. applying all Atomic Rules from each method to all applicable input fields).

Interestingly, Fault A was detected by four participants during the PNTP testing phase and by two during Atomic Rules testing. Fault B was not detected by any participants during either phase of testing. If the participants had applied EP, BVA and ST 'completely' during Atomic Rules testing, they would have

detected both seeded faults. There are at least three reasons for why they did not apply the Atomic Rules approach in this way:

1. they did not have enough time;
2. they did not understand how to apply the rules in this way; or
3. they purposely omitted applying the rules that would have detected the faults as they did not think they would have detected any faults.

Although data collected during the experiment does not allow the experimenter to determine which option is true, it does show that these faults were detectable by the Atomic Rules approach.

Seeded faults were not added to the Batch Processor, as both the author and the independent professional tester who carried out the fault seeding felt that that program already contained enough input/output validation errors that were detectable by EP, BVA and ST.

6.3.5 Efficiency (H_{03}/H_{13})

Efficiency can be evaluated as tester 'productivity' in terms of the number of equivalence classes, boundary values and syntax testing test data values that are derived correctly by a tester over the total time taken (Section 6.3.5.1). The testers were also questioned as to whether they felt they had enough time to complete the testing that was assigned to them (Section 6.3.5.2).

6.3.5.1 Productivity

Productivity was assessed by comparing the total number of correct test data values derived by the testers during the 3.5 hours of each testing phase (Table 6-35). Although a t-test indicated a non-significant result, it was just outside the 95% confidence interval, Cohen's effect size was moderate and the mean was higher during Atomic Rules testing. Therefore, it is plausible that testers could be more productive when using the Atomic Rules approach to derive black-box test data.

Table 6-35: Mean productivity.

N	Approach	Mean Number Test Data Values Derived per Hour (EP, BVA & ST)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	13.01	7.33	$t(10) = -1.66$, $p = .06$	Moderate
11	Atomic Rules	19.74	10.9		

6.3.5.2 Enough Time for Testing

On the Post-Testing Questionnaire the participants were asked whether they had enough time to complete testing (Table 6-36), to which 45% replied 'no' for PNTP and Atomic Rules Testing. This is likely to have contributed towards the relatively low percentage of black-box test method coverage that was achieved during both phases of testing (Section 6.3.3.1).

Table 6-36: Opinions from participants as to whether they felt they had enough time for testing.

Enough time for testing?	PNTP		Atomic Rules	
	Count	Percent	Count	Percent
No	**5**	**45**	**5**	**45**
Yes	4	36	4	36
Undecided	2	18	1	9
Missing	*0*	*0*	*1*	*9*

Participants commented on their response to this question after PNTP (Table 6-37) and Atomic Rules testing (Table 6-38). Five participants (46%) reported that in future, they would like to use a matrix to track test cases executed against each field, such as Test Matrices (see Chapter 2, Section 2.2.6). A Test Matrix was used by the author during data analysis to determine which participants had applied each Atomic Rule to each individual input field. When teaching the Atomic Rules approach in future, a simple improvement could be to demonstrate how Test Matrices can be used to plan and track test coverage.

Table 6-37: Participant feedback on the time allocated for the PNTP testing phase.

Feedback: Do you feel you had enough time for testing?
Base on limited knowledge - yes.
But considering what I have stated above I would not have signed off on this product with the time constraints and ambiguity of the spec.
As above - gave the testing scope a good cover.
Needed time to plan. To set goals. Identify what needed to be tested, what could be tested together and what needed to be separated.
Confident that testing was reasonably comprehensive. Discovered many defects.
I was only able to write three negative tests.
I would have preferred a bit more time to check the Data Analyst output to ensure all entries added up correctly. I probably should have done this earlier but I was focussing more batchid + recordline data + determining what accepted + rejected.
I would say yes and no as with more tests comes more confidence.
Having not previously been exposed to test writing I would have preferred more time to plan a more structured approach.
I answered both yes and no. Reason, with the small scope of the application it was easy to thoroughly investigate some areas. However if in a real testing environment I would have preferred to establish a test matrix to ensure all areas of functionality were covered.

Table 6-38: Participant feedback on the time allocated for the Atomic Rules testing phase.

Feedback: Do you feel you had enough time for testing?
Day 2 [AR] approach somewhat needs investigation and analytical skills to produce a good test case and test writing.
This program would take quite a while to get to the stage where I would be happy to sign off on it.
I think I got a grasp on the above - you will be able to tell me if I did but it was very tiring and I was ready to stop.
It took longer than yesterday. I spent a lot of time on the minute detail and feel I still could have missed issues. I felt some concepts were too abstract for me to pick up immediately. It took a while.
Time seemed adequate - someone with more/better understanding may or may not agree.
I think I nearly achieved full coverage, a few more hours would have helped.
Sufficient time allowed.
Again yes & no. Able to structure tests but did not able to write them.
Need to become more familiar with it.
Due to limited time I was unable to write test cases for flat, unit RSD, country & C/o addresses. However, I do feel this was more structured approach to yesterday's "feeble" attempts!
Much better than yesterday.

6.3.6 Errors Made (Effectiveness – Accuracy) (H_{04}/H_{14})

The number of mistakes made by the testers during PNTP and Atomic Rules testing was analysed, to assess the 'accuracy' of their testing. There was no significant difference between the errors made during PNTP or Atomic Rules testing (Table 6-39), suggesting that there is no difference in the accuracy of test cases produced during Exploratory Testing or prescriptive testing.

Table 6-39: Errors made during testing (effectiveness – accuracy) (H_{04}/H_{14}).

N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
11	PNTP	1.91	103.5	U = 321.92, p = .07
11	Atomic Rules	2.91	149.5	

The most common mistakes made during PNTP testing were missing white spaces in test cases (e.g. *100 Main Rd Eltham 3095*) (27%) and incorrect alphabetical case (e.g. *'Flat'* instead of *'FLAT'*) (27%) (Table 6-40). While these mistakes resulted in 'interesting' test cases, they also caused the testers to write incorrect expected results and to ultimately produce 'false negatives' during testing (i.e. testers raising defect reports for functions that contained no defects).

Table 6-40: Mistakes made during PNTP testing.

Mistake	Count	Percent
Missing space	**3**	**27**
Incorrect case for alpha character	**3**	**27**
Extra spaces	2	18
Extra field added	2	18
Spelling mistake	1	9
Missing symbol /	1	9
Missing keyword	1	9
Missing full stop	1	9
Incorrect boundary value selected	1	9

The majority of mistakes made during Atomic Rules testing were due to misunderstandings Atomic Rule names (Table 6-41). For example, 73% misunderstood how to use EP 'replacement' rules *EP4* to *EP10*. The aim of these rules is to replace valid input fields with invalid datatypes, such as applying *EP4: Integer Replacement* to the <*street_type*> field of the Address Parser, to select the invalid test case *100 Main **1000** Greensborough 3088*. A number of participants used *EP4* to replace integer fields with other datatypes, such as replacing <*postcode*> with alpha characters, selecting the invalid test case *100 Main Road Greensborough **ABCD***. A similar mistake was made by 45% of the group who used *ST9* to select null in one test case field, while the purpose of *ST9* is to select null in all fields. Another mistake made by 36% of the group was using *ST7* to convert alpha characters from lowercase to uppercase, when *ST8* should be used for this purpose and visa versa.

Nevertheless, each mistake resulted in a useful test case that was in the 'spirit' of black-box testing. Future research will include reducing the ambiguity of these Atomic Rules (Chapter 7). In addition, the misuse of one rule, *ST14*, resulted in the definition of a new Atomic Rule (see Section 6.3.10).

Table 6-41: Mistakes made during Atomic Rules testing.

Mistake	Count	Percent
Misuse of EP replacement rule	**8**	**73**
Used ST9 instead of EP11	5	45
Applied TCCR at field level	4	36
Used ST8 instead of ST7 & visa versa	4	36
Used EP rule instead of ST rule	3	27
Used wrong EP rule number	2	18
Incorrect boundary value selected	2	18
Used ST14 to reverse fields of test case	2	18
Used BVA rule instead of EP rule	1	9
Used BVA rule instead of ST rule	1	9
Used EP12 to select invalid data	1	9
Used EP7 and EP8 together	1	9

6.3.7 Understandability (H_{05}/H_{15})

As discussed in Chapter 1 (Section 1.2.1), three approaches to assessing the understandability of a test case design method are by evaluating whether a tester understands the conditions under which the method should be applied (Section 6.3.7.1), by examining their ability to apply the method correctly (measured by effectiveness, see Section 6.3.3) and by assessing their self-rated understanding of EP, BVA and ST (Section 6.3.7.2) and of the Atomic Rules approach (Section 6.3.7.3) and their ability to apply the four-step test case design process correctly (Section 6.3.7.4).

6.3.7.1 Conditions for Test Method Application

Other researchers have examined whether testers understand the conditions under which one test method should be used over another (e.g. see (Vegas et al. 2003)), as this can be indicative of their level of skill in testing. During Atomic Rules testing, the participants were specifically instructed to use EP, BVA and ST; therefore, they were not given the option of deciding which test methods were applicable.

On the other hand, during the PNTP testing phase the participants could have chosen to use any black-box testing method they felt was appropriate. Interestingly, they did not use any prescriptive testing methods for PNTP test case design. The reasons for this could include the following:

a) they did not believe that prescriptive testing methods should be applied during PNTP testing;

b) they did not recognise the need for applying any prescriptive testing methods; or

c) they felt there was not enough time to use prescriptive test methods during that phase of testing.

Although the participants were asked (on the Post-PNTP Testing Questionnaire) to describe the approach they took to PNTP testing (see Section 6.3.2), they were not asked why they did (or did not) use any prescriptive testing methods during that phase of testing. Therefore, a conclusion cannot be drawn as to whether they understood the conditions under which each black-box testing method should be applied.

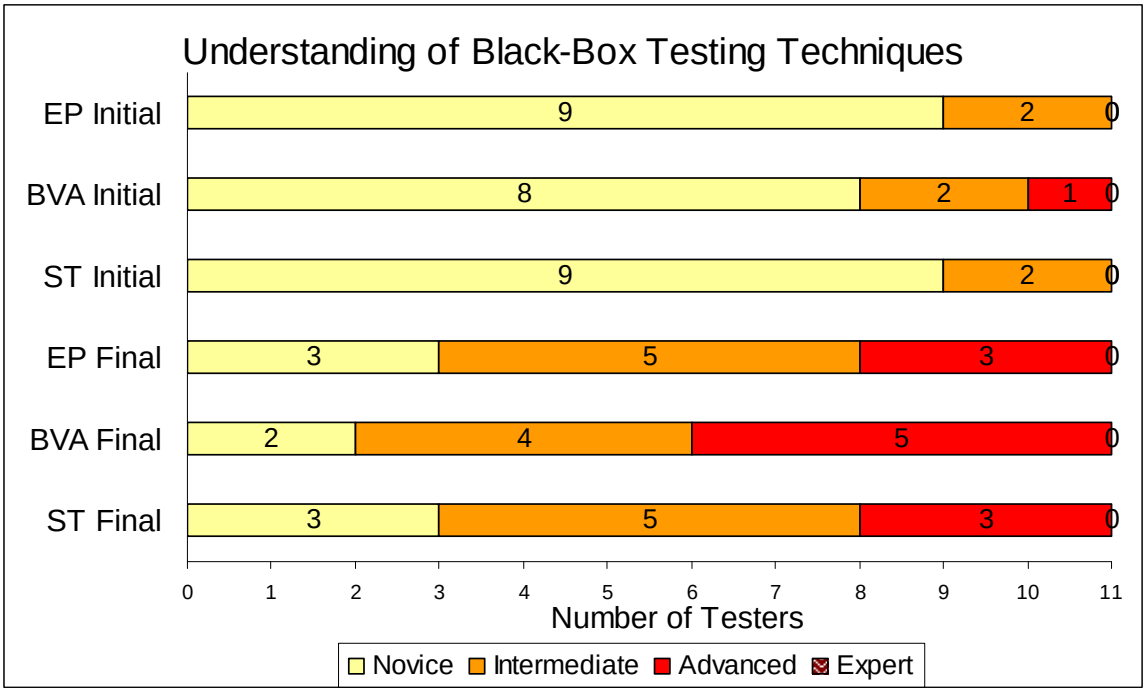
6.3.7.2 Understanding of Black-Box Testing Methods

On the Reflect and Review Questionnaire, the participants were asked to rate their initial and final understanding of EP, BVA and ST using a Likert scale of: 1 = novice, 2 = intermediate, 3 = advanced, 4 = expert (Table 6-42 and Figure 6-16). Most felt they had increased their understanding of these methods by the end of the experiment. They also achieved higher mean coverage of BVA and ST during Atomic Rules testing (see Section 6.3.3). This suggests that the Atomic Rules approach is an effective representation for teaching black-box testing methods to testers in industry.

Table 6-42: Self-rated understanding black-box testing methods before and after the experiment (choose one rating for each method).

Rating	Understanding of Black-Box Testing Methods					
	Initial			Final		
	EP	BVA	ST	EP	BVA	ST
	Percentages (%)					
Novice	**81**	**72**	**81**	27	18	27
Intermediate	18	18	18	**46**	36	**46**
Advanced	0	9	0	27	**46**	27
Expert	0	0	0	0	0	0

Figure 6-16: Self-rated understanding of black-box testing methods before and after the experiment.



Marginal Homogeneity (Table 6-43) and Crosstabulation tests for EP (Table 6-44), BVA (Table 6-45) and ST (Table 6-46) indicated significant differences, where understanding ratings were higher for all test methods after learning the Atomic Rules approach. These results indicate that learning the Atomic Rules approach improves a tester's understanding of black-box testing methods.

Table 6-43: Comparison of initial and final understanding of EP, BA and ST.

	Initial and Final Understanding		
	BVA	EP	ST
Distinct Values	3	3	3
Off-Diagonal Cases	8	8	8
Observed MH Statistic	10	10	10
Significance	*p* < .01	*p* < .01	*p* < .01

Table 6-44: Crosstabulation of initial and final understanding for Equivalence Partitioning.

Count		EP – Final Understanding			
		Novice	Intermediate	Advanced	Total
EP – Initial Understanding	Novice	3	5	1	9
	Intermediate	0	0	2	2
Total		3	5	3	11

Table 6-45: Crosstabulation of initial and final understanding for Boundary Value Analysis.

Count		BVA – Final Understanding			
		Novice	Intermediate	Advanced	Total
BVA – Initial Understanding	Novice	2	4	2	8
	Intermediate	0	0	2	2
	Advanced	0	0	1	1
Total		2	4	5	11

Table 6-46: Crosstabulation of initial and final understanding for Syntax Testing.

Count		ST – Final Understanding			
		Novice	Intermediate	Advanced	Total
ST – Initial Understanding	Novice	3	5	1	9
	Intermediate	0	0	2	2
Total		3	5	3	11

Interestingly, a comparison of the group's self-rated understanding of these methods in the Initial Questionnaire (Table 6-47, col. 2 to 4) and the Reflect and Review Questionnaire (Table 6-47, col. 5 to 7) revealed that more participants felt they were initially at an 'intermediate' level of experience with EP and BVA before the experiment than at the end. This suggests some participants had 'unconscious incompetence' (also known as the 'Dunning–Kruger effect' (Kruger & Dunning 1999)), in that they 'did not know what they did not know' about the methods until after they learnt how to use them during the Atomic Rules testing phase.

Table 6-47: Comparison of self-rated understanding EP, BVA and ST between the Initial and the Reflect and Review Questionnaires (choose one rating for each method).

Rating	Initial Understanding of Black-Box Testing Methods					
	Initial Questionnaire			Reflect & Review Questionnaire		
	EP	BVA	ST	EP	BVA	ST
	Percentages (%)					
Never heard of method	18	18	**54**	0	0	0
None	0	0	18	0	0	0
Novice	9	9	9	**81**	**72**	**81**
Intermediate	**46**	**46**	9	18	18	18
Advanced	27	27	9	0	9	0
Expert	0	0	0	0	0	0
Missing	*0*	*0*	*0*	*0*	*0*	*0*

6.3.7.3 Understanding of the Atomic Rules Approach

On the Reflect and Review Questionnaire, the testers rated their understanding of the Atomic Rules approach, using a Likert scale of: 1 = excellent, 2 = very good, 3 = good, 4 = average, 5 = poor, 6 = very poor (Table 6-48). All participants reported having an average, good or very good understanding.

Table 6-48: Participant's self-rated understanding of the Atomic Rules approach (choose one).

Rating	Count	Percent
Average	**6**	**55**
Good	4	36
Very good	1	9
Excellent	0	0
Poor	0	0
Very poor	0	0

The participants were asked to comment on their answer (Table 6-49). Seven (64%) reported that having more experience with the Atomic Rules approach (e.g. by using it at work) would enable them to gain a better understanding of it. One participant stated that their previous job as a software developer had assisted them in understanding the terminology and structure of the approach. This is an insightful view, since the attributes *Set Type*, *Original Datatype* and *Test Datatype* that are used in the Atomic Rules schema are based on similar concepts in software development. Thus, this suggests that software development experience can assist with the understanding the Atomic Rules approach.

Table 6-49: Participant feedback on their understanding of the Atomic Rules approach.

Feedback: Understanding of the Atomic Rules approach
2 days in not enough for me to fully understand.
As before I still need to get my head around some of the wording and meanings. If we used this wording at work then of course it would make better sense.
Further training and practice will reinforce what I have learned.
Having developed previously the terminology was familiar and the structure is easily followed.
I feel I have gained a lot from this session.
I need more time to explore this method. On the surface it looks like a great tool to use.
I understand the concept and think it is very worthwhile but lack experience with testing.
If you go through all 3 checklists for each filed you couldn't miss anything in testing - it may get out of hand though - number of tests - risk management may have to take over.
Only learnt this approach today!!
With more experience I expect that this rating would improve.
Would need more time to become familiar with it.

6.3.7.4 Understanding of the Four-Step Test Case Design Process

Interestingly, only one participant applied the Atomic Rules approach in the order prescribed in the four-step test case design process, as follows:

1. Apply Data-Set Selection Rules to partition the input domain.
2. Apply Data-Item Selection Rules to each partition to select individual test data values.
3. Apply Data-Item Manipulation Rules to each test data value to select mutated test data values.
4. Apply Test Case Construction Rules to combine test data values into test cases.

After the experiment, it was realised that the Quick-Reference Guide used during Atomic Rules testing only listed rule numbers and names, not the rule application order prescribed above. This may have caused participants to disregard rule application order during testing. On the other hand, this may not be essential for experienced testers, as they may understand and apply the rules without recording the intermediate step of selecting equivalence classes before selecting and mutating individual test data values (e.g. this occurs when competent mathematicians skip intermediate steps when solving algebraic equations).

Thus, while rule application order may be important when teaching the Atomic Rules approach to novice testers, it might not be required when teaching the approach to experienced testers in industry.

6.3.8 Operability (H_{06}/H_{16})

Operability of the Atomic Rules approach was assessed by examining three factors: the ease of use of the approach (Section 6.3.8.1) and the advantages and disadvantage of using the approach, as reported by participants (Section 6.3.8.2), as well as by asking them whether it was likely they would use the approach in future (Section 6.3.8.3).

6.3.8.1 Ease of Use

One the Reflect and Review Questionnaire, the participants were asked how easy the Atomic Rules approach was to use, using a Likert scale of: 1 = very easy to use, 2 = easy to use, 3 = difficult to use and 4 = very difficult to use (Table 6-50). It was encouraging to see that 82% found it 'easy to use' or 'very easy to use,' suggesting that it is a practical representation for teaching black-box testing methods in industry. On the other hand, since the testers did not have access to any other test method representations to use as a basis for comparison, it is possible that their answer may differ if they had alternate representations to consider (e.g. Myers' definition of EP and BVA).

Table 6-50: Participant opinions on how easy the Atomic Rules approach is to use (choose one).

Rating	Count	Percent
Easy to use	**8**	**73**
Difficult to use	2	18
Very easy to use	1	9
Very difficult to use	0	0

6.3.8.2 Advantages and Disadvantages of Atomic Rules

On the Reflect and Review Questionnaire, the participants were asked what they felt were the biggest advantages (Table 6-51) and disadvantages (Table 6-52) of the Atomic Rules approach. Almost half (46%) felt it provided them with useful guidance on achieving 'good' program coverage. Other responses include having universal test case design rules and terminology and having an effective black-box testing checklist.

Table 6-51: Opinions on the biggest advantage of the Atomic Rules approach.

Feedback: Advantages of the Atomic Rules approach
A formal checklist to follow.
As a checklist it would help you to discover if there were any rules that you may have omitted from your test approach. I found that this helped me in that way.
Ease of use.
Ensuring good coverage.
Guidance & structure for coverage.
If have the time - thorough coverage.
It allows for a systematic approach which identifies core functionality.
Not too sure.
Scenarios are easy to adapt from the Atomic Rules. Quick Reference Guides.
Thorough coverage after rules defined which need to be tested.
Universal rules/terms.

Disadvantages that were reported include that the application of the Atomic Rules approach is "time consuming" and that it requires "lots of prep work." However, this is not surprising since prescriptive black-box testing does require more time spent in test case design, whereas "freestyle" Exploratory Testing (Copeland 2004) (which was performed during the PNTP testing phase) involves no preparatory steps. This

is one of the trade-offs between prescriptive and non-prescriptive testing. As one participant reported, "although initial setup is more time consuming, a better overall result is achieved."

Table 6-52: Opinions on the biggest disadvantage of the Atomic Rules approach.

Feedback: Disadvantages of the Atomic Rules approach
Don't really see any disadvantages.
It is a long process.
Lots of prep work.
N/A
Need to continually do it to become more familiar.
None. Although initial setup is more time consuming, a better overall result is achieved.
Not all tests have a rule that relates specifically. e.g. mismatched suburb/postcode.
Not so easy to put into practice without testing experience.
Time consuming to set up.
To determine when you have gained adequate test coverage & not go overboard on possible test case scenarios.
We use different language and that is going to take a long time to change our system.

6.3.8.3 Use Atomic Rules in Future

On the Reflect and Review Questionnaire, participants were asked whether it was likely they would use the Atomic Rules approach in future (Table 6-53). An equal number (45% each) felt it was either very likely/somewhat likely and neither likely/unlikely that they would use Atomic Rules in future.

Table 6-53: Opinions on how likely it is they will use Atomic Rules in future (choose one).

Response	Count	Percent
Neither likely nor unlikely	**5**	**45**
Very likely	3	27
Somewhat likely	2	18
Somewhat unlikely	1	9
Very unlikely	0	0

The participants were asked to explain their answers. Seven (64%) said they would use Atomic Rules in some capacity immediately or if the opportunity arose in future (Table 6-54). Of the participants who stated that they would not use the approach in future (27%), two said it depends on their job position and if it involves test case design.

Table 6-54: Feedback on how likely it is that the participants will use Atomic Rules in future.

Feedback: How likely is it that you will use the Atomic Rules approach in future?
Again it is possible that I may use this approach. However will be of benefit for my team of testers to learn this.
As stated earlier this is now a part of the approach I will use.
Depends on current position.
I'll use it Monday as I think it's excellent.
I am sure this information could be put to good use in [our organisation].
I will not be constructing test cases in the near future so again - not relevant.
I would use this approach if I was engaged as a tester in the future. I would endorse this approach to testers.
If I get to write tests - somewhat likely. In [our organisation] we rarely get to write and test ourselves.
In [our organisation], we may occasionally use a matrix but most cases the tests are written in [a different department of our organisation].
Most of our test writing is done in [a different department of our organisation]. I have been fortunate in being able to write tests for the application. I will definitely take these rules back with me and keep them to check against the next time I write tests.

The participants were also asked whether their participation in the experiment would impact on the way they perform black-box testing in future (Table 6-55). Almost two thirds (73%) answered "yes."

Table 6-55: Feedback on whether taking part in the experiment will impact how the participants perform black-box testing in future.

Response	Count	Percent
Yes	8	73
No	1	9
Undecided	1	9
Missing	*1*	*9*

They were then asked to explain their answer (Table 6-56). One felt the skills they gained during the experiment would enable them to do "better" testing, while another said that when they gave feedback to the test writers in their organisation (i.e. Business Analysts), they would include examples of how Atomic Rules could be applied to the applications under test.

These results suggest that learning the Atomic Rules approach can have a direct and positive impact on the way organisations conduct black-box testing.

Table 6-56: Participant comments on whether taking part in the experiment will impact on how they carry out black-box testing in future.

Feedback: Will taking part in this experiment impact on how you perform black-box testing in future?
Become more aware of different programs.
I have greater understanding.
If there is one.
It will allow me to write better tests with better coverage.
Learnt a few interesting points that may change the way I execute tests in the future.
Lets me see how others approach testing.
Mostly we are prescribed as to how we test. BA's in [our organisation] write the tests and we follow the instructions. Feedback to them on their tests will now include examples of the Atomic Rules applied.
New skills = better testing in future.
Not really as I don't test very often anymore. However, will assist for ideas for my team.
Only if I test.
Will go into maybe things like symbols etc - more creative in my testing.

6.3.9 Satisfaction (H_{07}/H_{17})

Satisfaction was assessed by assessing how satisfied they were with the effectiveness of the test methods they used during PNTP and Atomic Rules testing (Section 6.3.9.1) and their satisfaction towards participating in the experiment (Section 6.3.9.2).

6.3.9.1 Satisfaction with the Testing Approaches

Satisfaction was assessed by asking the group to rate the effectiveness of PNTP and the Atomic Rules approach, using a scale of: very effective, somewhat effective, somewhat ineffective or very ineffective (Table 6-57). The majority reported that the approaches were ‘somewhat’ effective. This indicates that testers in industry might not have a preference towards using their own ‘PNTP’ approach to testing over the Atomic Rules approach. As a Likert scale was not used for this question, significance testing could not be carried out. The results for this hypothesis are inconclusive.

Table 6-57: Effectiveness of the two testing approaches.

Rating	PNTP (%)	Atomic Rules (%)
Somewhat effective	**64**	**55**
Very Effective	9	27
Somewhat ineffective	18	9
Very ineffective	0	9
Missing	*9*	*0*

They were then asked to comment on these ratings (Table 6-58, Table 6-59). After the PNTP testing phase, many participants felt unsure of the coverage or effectiveness of their test cases (Table 6-58). Reasons that were reported include that the specification was ambiguous, their approach lacked test planning and they lacked knowledge in testing.

Table 6-58: Participant comments on the effectiveness of PNTP test case design.

Feedback: How effective do you feel your approach to testing was?
I am not sure of the effectiveness.
As it ways not made clear in the spec what/who was the end user and what it was used for – may well have been able to cut tests or completely missed others.
I was stretching to think of any other avenues to test in 'Addresses.'
I am sure that I missed some aspects. Had I order and a plan I think the outcomes would have been better. I put too many issues in the one test - it would have been better to separate them.
Ad hoc approach due to lack/level of knowledge of testing.
I do not think that I achieved enough coverage with this approach.
Being unsure of the product and being doubtful of abilities. Would hope that my approach was ok.
Learnt new things like terms, private industry.
I have not achieved adequate coverage & was not able to test several key functions of the program.
In the first case initial testing was slow due to generating the template to create the sequences. Once complete, testing was increased to approximately 10 tests per hour due to easily being able to generate and modify new data sets.

After Atomic Rules testing, some participants felt they achieved better coverage with Atomic Rules, while others found the vocabulary confusing (Table 6-59). For example, one participant stated that “Due to limited time I was unable to write test cases for flat, unit, RSD, country & C/o addresses. However, I do feel this was a more structured approach to yesterday's ‘feeble’ attempts!”

Table 6-59: Participant comments on the effectiveness of Atomic Rules test case design.

Feedback: How effective do you feel your approach to testing was?
If you fully understand the concepts.
The programme had numerous bugs and it would have taken a lot of structured testing and maybe would have been helped to have someone ad hoc as well.
I felt I had more control over the coverage of my tests yesterday doing it my way.
Terminology confusing. I got lost a few times with the terminology. I think this approach would work well when work-shopped with a team prior to test writing.
My efforts at testing hampered by general lack of knowledge/experience in testing.
It allowed me to achieve excellent coverage.
I was quite comfortable with this approach.
Still a little unsure of ability and effectiveness.
May not be relevant to current position but gives me an idea of how BA's work.
Due to limited time I was unable to write test cases for flat, unit, RSD, country & C/o addresses. However, I do feel this was a more structured approach to yesterday's "feeble" attempts!
It thoroughly covered a wide range of scenarios and was easy to ensure all tests were completed and analysed for all fields.

On the Reflect and Review Questionnaire, participants were also asked whether they felt learning the Atomic Rules approach enabled them to write more effective test cases, to which 55% agreed with this statement (Table 6-60).

Table 6-60: Opinions on whether Atomic Rules enables the design of more effective test cases.

Response	Count	Percent
Yes	**6**	**55**
No	3	27
Undecided	1	9
Missing	*1*	*9*

The testers were asked to explain their responses (Table 6-61). Most expressed positive views towards the Atomic Rules approach, suggesting that testers in industry would perceive the approach as effective for use in industry.

Table 6-61: Feedback on whether Atomic Rules enables the design of more effective test cases.

Feedback: Does the Atomic Rules approach enable the design of more effective test cases?
Absolutely! I had no idea what I was doing yesterday & although I have probably drilled down into more detail than required today I feel the rules helped a great deal.
I feel I am thorough with my old way of doing testing - cover all avenues.
I only wrote one as there was not enough time. It would be good to trail it at work with a knowledge base I can use.
I think so and the Atomic Rules approach has more structure.
Much better coverage was achieved.
Not all parts were relevant to current work/industry.
Quite an easy to understand approach.
The concept was well explained and the application of the concept would be very helpful.
What it did was make an effective checklist from the list that I had already made by going through the spec.

6.3.9.2 Satisfaction with Experiment Participation

On the Reflect and Review Questionnaire, participants were asked what they liked (Table 6-62) and disliked (Table 6-63) the most about taking part in the experiment. While six testers (55%) said there was nothing specific they disliked, two said they doubted their own testing abilities after the experiment. This could be due to their abilities being analysed alongside that of their peers. In contrast, it was encouraging to see 64% reporting that the aspect they liked most was they enjoyment of learning a new testing approach (Table 6-62). While the survey sample is extremely small, we feel justified in remarking that testers in industry seem to enjoy learning new approaches to testing.

Although subjective, this view is supported by the observations of the author during the teaching of software testing training courses to industry professionals, including K. J. Ross & Associates' Certified Software Test Processional course. Having taught over fifteen industry-based software testing training courses over the past three years, the author has seen very few students who did not enjoy the experience in learning about black-box testing methods, even when students found that part of the course challenging.

Table 6-62: Participant feedback on what they liked about participation in the experiment.

Feedback: What did you like the most about participating in this experiment?
Being challenged by experiencing something different.
Good group, great tutor, great food, break from work, gained new knowledge.
It was interesting and job specific.
Learning about testing.
Learning more to reinforce my knowledge of testing methodologies.
Learning new ways to test & test write cases. Friendly, helpful instructor.
Learnt valuable concepts to testing concepts. e.g. Atomic Rules.
Opened a new way to look at testing, i.e. formally having a checklist rather than an informal way.
Preparation.
The new perspective I gained on testing methodologies.
Was very interesting.

Table 6-63: Participant feedback on what they disliked about participation in the experiment.

Feedback: What did you dislike the most about participating in this experiment?
Feeling stupid!
I was happy with our progress over the 2 days. No dislikes.
No dislikes what so ever (got tired).
Not relevant to current position.
Nothing - enjoyed the experience.
Nothing - it was beneficial.
Nothing really - thanks Tafline.
Nothing.
Some concepts were a bit abstract for me. Took a while for the penny to drop.
Sometime I am not able to follow the instruction.
The unknown. Doubting of ones own ability.

6.3.10 Tester Motivation (H_{08}/H_{18})

To determine whether 'enthusiasm' was a threat to experiment validity (see Section 6.2.5.1), on the Initial and Post-Testing Questionnaires the testers rated their motivation levels using a Likert scale of: 1 = not motivated at all, 2 = very unmotivated, 3 = somewhat unmotivated, 4 = somewhat motivated, 5 = very motivated and 6 = exceptionally motivated (Table 6-64, Figure 6-17).

Although results indicate that motivation levels did change during the experiment, they followed a similar trend and the majority of the group felt "very motivated" throughout. Since the participants did not report feeling more motivated after using the Atomic Rules approach, enthusiasm did not appear to threaten validity. On the contrary, a t-test revealed a significant difference in their motivation levels in the Post-Testing Questionnaires, where the mean was higher after the PNTP testing phase than after Atomic Rules testing (Table 6-65).

Table 6-64: Participant motivation levels during the experiment (choose one) (tabular view).

Response	Initial (%)	After PNTP Testing (%)	After Atomic Rules Testing (%)
Very motivated	**55**	**73**	**55**
Somewhat motivated	0	18	27
Exceptionally motivated	18	9	0
Very unmotivated	9	0	18
Somewhat unmotivated	18	0	0
Not motivated at all	0	0	0

Figure 6-17: Participant motivation levels during the experiment (graphical view).

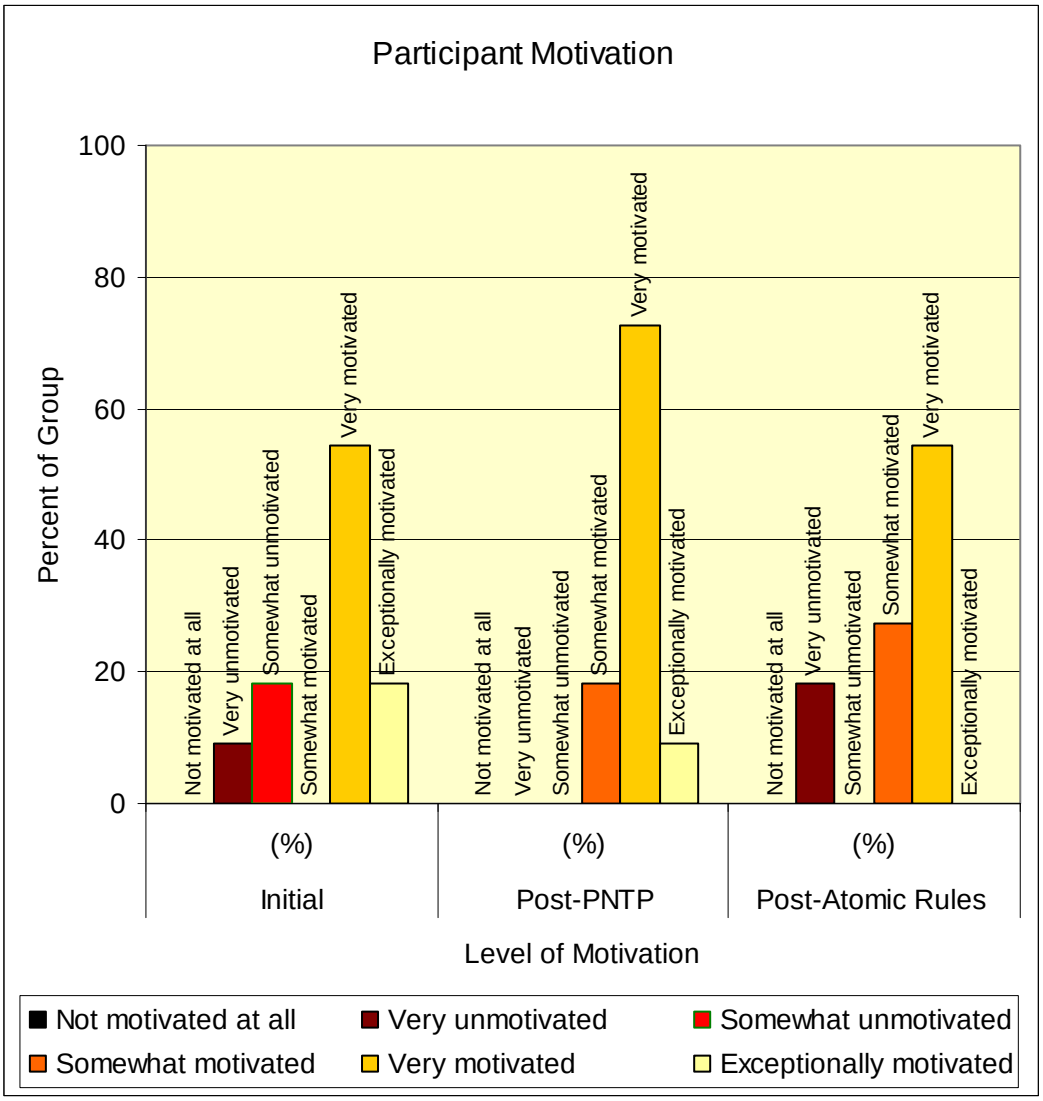


Table 6-65: Comparison of test motivation levels after PNTP and Atomic Rules testing.

N	Approach	Mean (%)	Std Dev	t-test	Cohen's Effect Size
11	PNTP	4.9	.53	$t(10) = 2.02$, $p < .05$	moderate
11	Atomic Rules	4.1	1.1		

The testers may have felt less motivated after Atomic Rules testing because they felt lethargic after participating in a two-day experiment. On the other hand, they may have felt more motivated after PNTP testing because they enjoyed Exploratory Testing (used during PNTP testing, see Section 6.3.2) more than they enjoyed using prescriptive testing methods (used during Atomic Rules testing).

This view is supported by Reid (2007), who investigated the job satisfaction of nine different roles of testing, including black-box test design, test execution and Exploratory Testing. Basing his research on Hackman and Oldham's Job Characteristics Model (1980), Reid proposed an equation for calculating the Motivating Potential Score (MPS) for roles in testing, which was based on five attributes:

1. skill variety (V; range of skills required);
2. task identity (I; degree of completing a job);
3. task significance (S, importance of the job);
4. autonomy (A, level of control of own time); and
5. feedback (F, degree of supervisory and results-based feedback on performance)

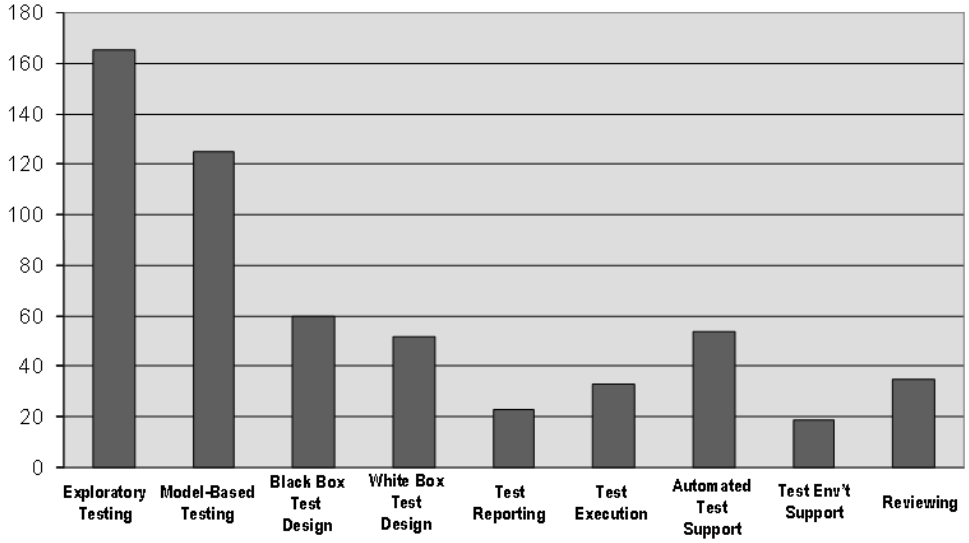
From this, Reid developed the following equation (Reid 2007).

$$MPS = \frac{(V + I + S)}{3} * A * F$$

Reid subjectively assigned ratings to this equation from 1 (low) and 7 (high) for nine roles in testing (Figure 6-18). Exploratory Testing achieved the highest MPS at just over 160, compared to only 60 for black-box test design, suggesting that testers feel greater satisfaction when they are conducting Exploratory Testing than when they are carrying out prescriptive black-box test case design and execution.

Thus, given that Exploratory Testing was used during the PNTP testing phase of this experiment (see Section 6.3.2), Reid's findings could explain why the testers in this experiment felt more motivated after the PNTP phase of testing (see Table 6-64 and Figure 6-17).

Figure 6-18: Motivating Potential Score (MPS) for software testing roles (Reid 2007). The highest rating role is Exploratory Testing, which was using during the PNTP phase of this experiment.



6.3.10.1 Individual Feedback

On the Initial Questionnaire and Post-Testing Questionnaires, the participants were asked to explain their level of motivation (Table 6-66, Table 6-67, Table 6-68). Only one participant provided negative feedback after PNTP testing, saying that the experience was "a bit tedious." Two participants provided negative feedback after Atomic Rules testing, with one reporting that they felt "tired and brain fogged" and the other commenting that the date of the experiment affected their motivation, stating that "It's Friday afternoon – no more explanation needed."

Table 6-66: Participant comments on their motivation levels at the start of the experiment.

Motivation	Feedback
Exceptionally motivated	Something new to look at and hopefully will help me in understanding of system testing broad spectrums.
Very motivated	Am interested in finding out where this is going.
Very motivated	Something new and challenging. Supporting my job with new knowledge and informed on how the industry works outside of [our organisation].
Exceptionally motivated	Always interested to do/learn new things and help someone along the way.
Very motivated	The experiment will give me an insight into testing. This will assist in my role in assessing learning needs for [our department's] staff.
Very unmotivated	I'm looking to learn as much as I can from this experiment.
Very motivated	I am always keen to learn more regarding testing methodologies in order to improve my testing skills.
Somewhat motivated	Am keen to learn new skills but unsure of the outcome and how I will go in all of this.
Very motivated	Sounds interesting, something different + may learn e.g. new terms etc.
Somewhat motivated	A little anxious due to the expectations & my limited testing knowledge.
Very motivated	As a new tester and previous developer any new knowledge in this area is of great benefit.

Table 6-67: Participant comments on their motivation levels after PNTP testing.

Motivation	Feedback
Very motivated	I like to know more if it can help me in my current job.
Very motivated	Want to have a go at Batch testing next to see what it is like.
Very motivated	This is different, not threatening, and keeps the mind on alert.
Exceptionally motivated	Great learning experience. Always happy to update my skills + knowledge.
Very motivated	Learning more about testing and seeing what skills/knowledge need to be learned, for reference in my function as [staff] coordinator
Very motivated	I have enjoyed the experience and I think it will be beneficial in relation to my work at [our organisation].
Very motivated	I find it interesting to see what I am able to achieve + the satisfaction I feel when I am able to detect defects.
Somewhat motivated	Am enjoying the experience. Somewhat doubting my abilities and knowledge. Think that this will build in time.
Somewhat motivated	Gets a bit tedious but that is probably across the board with system testing.
Very motivated	Am hoping I will be able to learn some basic concepts in approaching test writing.
Very motivated	Looking forward to phase 2 to see how I tackle the next one.

Table 6-68: Participant comments on their motivation levels after Atomic Rules testing.

Motivation	Feedback
Very motivated	I enjoyed in participate the program and have a very understanding tutor to help with all the questions. Good presenter.
Somewhat motivated	It's Friday afternoon - no more explanation needed.
Very motivated	No one wants to be in a rut and all these new techniques to me I found interesting.
Very motivated	Am tired and brain fogged. Not a bad feeling though.
Very motivated	I was pleased to be able to provide data that help testing professionals in their roles.
Very unmotivated	It was interesting and challenging.
Very motivated	I am always eager to gain information that will improve my testing knowledge.
Somewhat motivated	End of day 2. Good experience.
Somewhat motivated	*No comment provided*
Very unmotivated	Would actually have liked to have spent another day here learning other approaches to test writing/construction.
Very motivated	Same as yesterday, however more so as I can see the benefits of the new approach.

6.3.11 Test Method Representation (H_{09}/H_{19})

The possibility that the test case design rules utilised during the Exploratory Testing (carried out during the PNTP phase) could be described as Atomic Rules was investigated. If they could, this would indicate that Atomic Rules from EP, BVA and ST could support Exploratory Testing by providing testers with a checklist against which they can audit their test coverage.

To start, the mean and total number of test data values derived during PNTP testing and Atomic Rules testing was compared (Table 6-69), to obtain an overall view of the number of times test case design rules

were applied during the experiment. An average of 46 and total of 501 test data values were derived by the testers during PNTP testing compared to an average of 69 and a total of 760 during Atomic Rules testing.

Table 6-69: Number test data values derived during PNTP and Atomic Rules testing.

N	Approach	Mean (count)	Std Dev	Total (Count)
11	PNTP	46	26	501
11	Atomic Rules	69	38	760

Of the 501 test data values that were derived during the PNTP testing phase, only eight (1.6%) could not be derived by Atomic Rules from EP, BVA or ST. This suggests that the Atomic Rules approach could provide excellent support to practitioners in industry when they are carrying out Exploratory Testing, as it would provide them with a set of test case design rules on which they can base their 'on-the-fly' test design. Furthermore, it could be used by auditors to assess the completeness of black-box test sets that are designed during Exploratory Testing (e.g. see Chapter 3, Section 3.5) and could be used to describe the overall approach to testing that was in use. It also suggests that the Atomic Rules approach may be 'complete' in terms of its coverage of the types of black-box test case design rules that are used by practitioners.

Of these, four could be described as Atomic Rules as they were prescriptive and could be applied to individual input fields, while the other four could not be described as Atomic Rules as they were non-prescriptive and based on tester "application solution domain knowledge" (Reed 1990) (see below).

The four that *could* be described as Atomic Rules are as follows.

1. During the PNTP testing phase, two participants added characters to the middle of keywords.

    a) For the suburb *Greensbor**ou**gh*, insert an invalid character to create a test data value *Greensbor**oiu**gh*

    b) For the keyword ***RSD***, add white spaces to create a test data value ***R S D***.

    Although ST includes rules for removing and replacing characters from the start and end of keywords (see *ST1*, *ST2*, *ST4* and *ST5* in Appendix B), a new Atomic Rule *ST17: Add Middle Character* could be defined to add characters to the middle of keywords (see Appendix B).

2. One participant removed characters from middle of a keyword, as follows.

    a) For the suburb *G**reen**sborough*, select test data value *G**rn**sborough*.

    Although ST includes rules for adding characters to the start and end of keywords (see *ST3* and *ST6* in Appendix B), a new rule *ST18: Remove Middle Character* could be defined to remove characters from the middle of keywords (see Appendix B).

3. One participant selected the second item from a list, as follows.

a) Select the second suburb *Box Hill North* from the list of suburbs (see Table 6-3).

Although BVA includes Atomic Rules for selecting the second and second last values from *range-based* fields (see *BVA2* and BVA5 in Appendix B), two new rules *BVA12: Second List Item Selection* and *BVA13: Second Last List Item Selection* could been defined to select the inside boundary values for list-based fields (Appendix B).

4. Two participants misunderstood *ST14: Select All List Alternatives in Reverse Order*, resulting in a new Atomic Rule. *ST14* is meant to be applied to list-based fields to select a test data value containing all elements from the field in the reverse order to which they were specified. For example, if *ST14* was applied to <*street_type*>, it would select the invalid test data value *"Place Ln Lane Grv Grove Crt Court Ave Avenue Rd Road St Street"*. Two participants used this rule to reverse the order of fields in the test case, e.g. creating the invalid test.*3088 Greensborough Street Main 100*. Thus, a new Atomic Rule *ST19: Reverse All Fields* has been defined.

On the other hand, two of the four test case design rules that could not be described as Atomic Rules were based on "application solution domain knowledge" (Reed 1990), as follows.

5. Two participants tested the Address Parser by replacing the directions '*North'* and '*South'* with abbreviations '*Nth'* and '*Sth'*, which were (correctly) rejected by the program They used "application solution domain knowledge" (Reed 1990) to select these values, which originated from their understanding of English language abbreviations. Since the number of abbreviations in any language is generally very large, it would be impractical to define a generic Atomic Rule to cover all possibilities. An automated test data selection tool would likely be required to support the prescriptive identification of such abbreviations. Further, an Exploratory Tester who is familiar with the application solution domain of the system under test is likely able to identify such test data more efficiently than through the use of a testing tool. Thus, this test case design rule should not, in the author's opinion, be defined as an Atomic Rule.

6. One participant tested the Address Parser suburb field by removing the word '*West'* from the (valid) suburb '*Geelong West'*. Similarly, one participant tested suburbs from the state of Queensland, even though the program only accepted Victorian suburbs. These tests were identified through application solution domain knowledge, which originated from their understanding of valid Australian suburbs. It would be more efficient to hire a tester to generate such test data than defining and using a generic Atomic Rule for it.

This evidence suggests that test case design rules that rely heavily on application solution domain knowledge might not be definable as Atomic Rules. Further investigation and experimentation of industrial approaches to testing is required to confirm or disprove this.

The two remaining rules that could not be described as Atomic Rules were not within the scope of EP, BVA and ST, as follows.

7. Two participants tested the Address Parser with non-matching suburb/postcode pairs, while another tested the Batch Processor for matching 'sbatch' and 'ebatch' identifiers. As discussed in Chapter 3, one limitation of the Atomic Rules approach is that it does not support testing dependencies between input fields (see Section 3.8). Therefore, an Atomic Rule cannot be defined to cover this form of test case design rule.

8. During PNTP testing, after running many invalid test cases, one participant re-ran their first valid test to "try and clear environment to make sure there are no environment issues as so many bugs have been occurring." This was not necessary for the programs under test, as all output files were overwritten each time the program was re-executed. In research conducted by Peri-Salas and Krishnan, if variables that define program behaviour are identified in the system specification, it is possible to define black-box test cases that exercise such constraints (Salas 2007). If information about the behaviour of program memory could be formally described (e.g. the times at which it is cleared), then BVA rules could potentially be applied to systematically test such boundaries. Nonetheless, this is not currently covered by the Atomic Rules approach.

6.4 Discussion

6.4.1 Results of Hypothesis Testing

Nine hypotheses were defined for this experiment, covering: completeness (effectiveness) (H_{01}/H_{11}), failure-detection effectiveness (H_{02}/H_{12}), efficiency (H_{03}/H_{13}), errors made (effectiveness - accuracy) (H_{04}/H_{14}), understandability (H_{05}/H_{15}), operability (H_{06}/H_{16}), satisfaction (H_{07}/H_{17}), tester motivation (H_{08}/H_{18}) and test method representation (H_{09}/H_{19}). The results for these are as follows.

The participants produced significantly more complete test data values for BVA and ST during Atomic Rules testing. Thus, the null hypothesis for completeness could be rejected in favour of the alternate hypothesis for BVA and ST (Table 6-70). On the other hand, the results for EP completeness were inconclusive, which suggests that testers in this group may have already had the ability to carry out EP purely from their own individual knowledge and experience prior to participating in the experiment.

Table 6-70: Outcomes of hypothesis testing for Completeness (H_{01}/H_{11}).

Hypothesis: Completeness ***H_{01} = Null, H_{11} = Alternate***	**Test Method Coverage**		
	EP	**BVA**	**ST**
H_{01}: The completeness of black-box test sets derived by industry-based testers is independent of the approach used.	Fail to reject	Reject	Reject
H_{11}: Industry-based testers using the Atomic Rules approach derive more complete test sets in terms of EP, BVA and ST coverage compared to those using their own method for black-box test case design.	Fail to accept	Accept	Accept

It was interesting to find a significant difference in the mean failure-detection effectiveness achieved by the testers, where the mean was higher during Atomic Rules testing. Thus, the null hypothesis for

completeness could be rejected in favour of the alternate hypothesis (Table 6-71). Furthermore, with the exclusion of the space fields in the Address Parser, the group as a whole achieved the same or improved levels of specification field coverage (i.e. input domain coverage) during Atomic Rules testing (see Section 6.3.3.4). These results suggest that learning the Atomic Rules enables industry-based testers to write more effective test cases both in terms of the failure-detection effectiveness of the test cases and in terms of the input domain coverage achieved by the testing group.

Table 6-71: Outcomes of hypothesis testing for Failure-Detection Effectiveness (H_{02}/H_{12}).

Hypothesis: Failure-Detection Effectiveness ***H_{02} = Null, H_{12} = Alternate***	**Failure-Detection Effectiveness**
H_{02}: There is no difference between the failure-detection effectiveness of the Atomic Rules approach compared to black-box testing approaches used by industry-based testers.	Reject
H_{12}: Industry-based testers detect more failures using the Atomic Rules approach then when using their own approaches to black-box test case design.	Accept

At a confidence interval of 94%, the results for efficiency (productivity) indicated a significant difference between the number of EP equivalence classes and BVA and ST test data values that were derived by the testers, where the mean was higher during Atomic Rules testing. Thus, the null hypothesis for efficiency can be rejected in favour of the alternate hypothesis (Table 6-72). This suggests testers in industry are likely to be more productive when using the Atomic Rules approach.

Table 6-72: Outcomes of hypothesis testing for Efficiency (Productivity) (H_{03}/H_{13}).

Hypothesis: Efficiency ***H_{03} = Null, H_{13} = Alternate***	**Efficiency[3]**
H_{03}: The efficiency of black-box test case derivation by industry-based testers is independent of the approach used.	Reject
H_{13}: Industry-based testers using the Atomic Rules approach derive test cases more efficiently compared to those using their own approach to test case design.	Accept

The results for the number of mistakes made during test case derivation were inconclusive, as no significant difference was found between the results for PNTP or Atomic Rules testing. Thus, the null and alternate hypothesis could not be accepted nor rejected for this attribute (Table 6-73).

Table 6-73: Outcomes of hypothesis testing for Errors Made (Completeness – Accuracy) (H_{04}/H_{14}).

Hypothesis: Errors Made ***H_{04} = Null, H_{14} = Alternate***	**Errors Made**
H_{04}: The number of errors made by novice testers during black-box test case derivation is independent of the approach used.	Inconclusive
H_{14}: Industry-based testers using the Atomic Rules approach make fewer errors during test case derivation compared to those using their own approaches to test case design	Inconclusive

[3] This result is based on a confidence interval of 94%.

The results for understandability indicated a significant difference in the level of understanding participants in EP, BVA and ST before and after learning the Atomic Rules approach, where the mean was higher after learning Atomic Rules. Thus, the null hypothesis could be rejected for this attribute in favour of the alternate hypothesis (Table 6-74). This indicates that the Atomic Rules approach is an effective representation for teaching black-box testing methods to testers in industry.

Table 6-74: Outcomes of hypothesis testing for Understandability (H_{05}/H_{15}).

Hypothesis: Understandability ***H_{05} = Null, H_{15} = Alternate***	**Understandability**		
	EP	**BVA**	**ST**
H_{05}: Learning the Atomic Rules approach has not affect on a tester's understanding of black-box testing methods.	Reject	Reject	Reject
H_{15}: Testers improve their understanding of black-box testing methods by learning the Atomic Rules approach.	Accept	Accept	Accept

A similar result was found for operability, in which a significant difference was found in the participant's opinions as to whether the Atomic Rules approach was easy or difficult to use, where the majority of the group found the new approach 'easy to use' or 'very easy to use.' Thus, the null hypothesis could be rejected for this attribute in favour of the alternate hypothesis (Table 6-76). This indicates that testers in industry would find the Atomic Rules approach easy to use.

Table 6-75: Outcomes of hypothesis testing for Operability (H_{06}/H_{16}).

Hypothesis: Operability ***H_{06} = Null, H_{16} = Alternate***	**Operability**
H_{06}: Testers using the Atomic Rules approach do not find it easy or difficult to use.	Reject
H_{16}: Testers using the Atomic Rules approach find it an easy to use.	Accept

The results for satisfaction in terms of whether participants would have a preference towards using the Atomic Rules were inconclusive. Thus, the null and alternate hypothesis could not be accepted nor rejected for this attribute (Table 6-76).

Table 6-76: Outcomes of hypothesis testing for Satisfaction (H_{07}/H_{17}).

Hypothesis: Satisfaction ***H_{07} = Null, H_{17} = Alternate***	**Satisfaction**
H_{07}: The preference of industry-based testers towards the use of black-box testing methods is independent of the representation used.	Inconclusive
H_{17}: Industry-based testers prefer to use the Atomic Rules approach for black-box test case design compared to using their own approaches.	Inconclusive

Tester motivation was compared after PNTP testing and Atomic Rules testing to determine whether their motivation for learning a new testing approach affected their results, which was a threat to validity (see Section 6.2.5.1). A significant difference was found in the participant's motivation after both phases of testing, where the mean was higher after the PNTP testing phase. Thus, the null hypothesis could be

rejected for this attribute in favour of the alternate hypothesis (Table 6-77). This suggests that testers in industry find Exploratory Testing more enjoyable than prescriptive testing, which supports research by Reid (Reid 2007) that indicated that testers get experience more enjoyment when performing Exploratory Testing than when using prescriptive black-box testing methods (see Section 6.3.10).

Table 6-77: Outcomes of hypothesis testing for Motivation (H_{08}/H_{18}).

Hypothesis: Motivation ***H_{08} = Null, H_{18} = Alternate***	**Motivation**
H_{08}: Testers feel more motivated when using a new technique simply because it is new.	Reject
H_{18}: Testers using the Atomic Rules approach do not find it more motivating to use simply because it is a new technique.	Accept

For test method representation, a thorough analysis was conducted to determine whether the test case design rules utilised during PNTP testing could be described as Atomic Rules. All test data values derived during PNTP testing except eight (1.6%) could be derived by Atomic Rules from EP, BVA and ST. Of these, four lead to the definition of new Atomic Rules for BVA and ST. Only four test case design rules could not be described as Atomic Rules, due to the rules being non-prescriptive and based on the application solution domain knowledge of the testers (see Section 6.3.11). Thus, the null hypothesis could be rejected for this attribute in favour of the alternate hypothesis (Table 6-78). This indicates that the Atomic Rules approach could be effective in supporting Exploratory Testing in industry, by allowing them to audit their own test coverage (e.g. using the Quick Reference Guide provided in Appendix C). It also suggests that the Atomic Rules approach may be 'complete' in terms of its coverage of black-box testing approaches that are used by practitioners in the software testing industry.

Table 6-78: Outcomes of hypothesis testing for Test Method Representation (H_{09}/H_{19}).

Hypothesis: Test Method Representation ***H_{09} = Null, H_{19} = Alternate***	**Test Method Representation**
H_{09}: Test case design rules used by experienced testers in industry cannot be described by any black-box test method representation.	Reject
H_{19}: Test case design rules used by experienced testers in industry can be described as Atomic Rules.	Accept (for some rules)

6.4.2 Black-Box Testing in Industry

As discussed in Chapter 2 (Section 2.5), a number of experiments have investigated the application of specific black-box testing methods by experienced testers in industry (e.g. see (Basili & Selby 1987, Laugherbach & Randall 1989, Vegas et al. 2003, Itkonen & Rautiainen 2005, Wood et al. 1997)). In these experiments, participants are usually asked to apply specific static or dynamic testing methods. One of the goals of this experiment was to examine how testers in industry perform testing when they are not required to use a specific test method. Interestingly, all participants used Exploratory Testing during the PNTP testing phase. Even the tester who started PNTP testing by developing and applying a random test case generator supplemented this with Exploratory Testing. On the other hand, by the end of the experiment

almost half the group stated that if they were to test the programs again, they would carry out prior test planning and design (see Section 6.3.2).

As has been mentioned earlier, this suggests that when testers are unfamiliar with an application, they may initially choose to carry out Exploratory Testing to 'shake-out' the program, allowing them to gain an understanding of its functionality while they explore potential defects, after which they will opt for a prescriptive testing method that allows them to plan and track test coverage. The Atomic Rules approach is a natural candidate for achieving this, as a simple Test Matrix can be used to trace the application of Atomic Rules to program input and output fields (see Chapter 3, Sections 3.4 and 3.5).

6.4.3 Effects of Domain Knowledge on Testing Effectiveness

A number of publications of Exploratory Testing suggest that there is no system to the seemingly 'intuitive' process that takes place during test design and execution. Kaner maintains that "in complex situations, your intuition will often point you toward a tactic that was successful (you found bugs with it) under similar circumstances. Sometimes you won't be aware of this comparison. You might not even consciously remember the previous situations. This is the stuff of expertise" (Kaner 1988). As Agruss and Johnson (2000) state, "much of what experienced software testers do is highly intuitive, rather than strictly logical." Nonetheless, Barber (2007) argues that "the more we know about what a system or application is supposed to do, the more intuitive we believe it is".

The Oxford English Dictionary (1970) defines intuition as "The immediate apprehension of an object by the mind without the intervention of any reasoning process." Regardless of whether a tester is consciously aware of the process they follow during Exploratory Testing, there is still likely to be a pattern to the test case design rules they use. As Kaner et al. observe (2001), a tester's skill with Exploratory Testing increases as they become familiar with the system under test, including its market, the risks associated with developing it and failures previously detected. The domain knowledge and experience that guides a tester during Exploratory Testing can be influenced by their understanding of prescriptive testing methods (Craig & Jaskiel 2002), testing heuristics (Watkins 2001), tests that previously detected faults (Watkins 2001), program implementation and design (Bertolino 2004, Watkins 2001), hardware (Mosley 1993), platforms (Bertolino 2004) and programmer assumptions (Myers 1979).

Thus, it is likely that the 'intuition' a tester employs during Exploratory Testing is logical and procedural "application solution domain knowledge" (Reed 1990) they have gained over time. There may be information about the application solution domain of the system under test that gives experienced "pathological testers" (Reed 2007) clues on how to test it. This view is supported by the resulted discussed in Section 6.3.4.3, where the failure-detection effectiveness achieved when using PNTP testing against the Address Parser was higher than for the Batch Processor during PNTP testing, but this was not the case during Atomic Rules testing. This indicates that while a tester can require extensive domain knowledge to carry out effective Exploratory Testing, prescriptive testing methods like those represented by the Atomic Rules approach can fill the gap when domain knowledge is not present.

In fact, many prescriptive black-box testing methods are based on domain knowledge of specific software fault classes. As Wild et al. attest (1992), faults are often caused by programmers misunderstanding the problem domain of the program under development. BVA is based on the (implementation level) domain knowledge that programmers often make 'off-by-one' errors, which can be classed as application solution domain knowledge. Error Guessing targets error-prone situations such as divide by zero errors and calculating the square root of a negative number (Mosley 1993), which are also forms of application solution domain knowledge.

Thus, the knowledge an experienced tester draws upon during Exploratory Testing includes:

- knowledge of prescriptive testing methods, such as EP, BVA and ST, as this supports in the selection of an optimised, high-yield test set;
- knowledge of the program under test and of similar programs, including requirements, design, source code (structure and contents), previous test cases that were effective and previously detected defects, as this assists in the design of test cases that cover the requirements, specifications and likely defects in the system under test (which Reed referred to as "implementation domain knowledge" (Reed 1990)); and
- general software development knowledge and experience, as this assists in the design of test cases that target certain types of program structures or program faults (e.g. buffer overflows) (which can also be considered as implementation domain knowledge (Reed 1990)).

Thus, experienced testers may be capable of designing high-yield test sets that achieve high failure-detection effectiveness without applying all Atomic Rules from all black-box testing methods to all input fields. There may also be many effective black-box test case design rules used by experienced testers that have not yet been published in software testing literature. If such rules could be defined as Atomic Rules (e.g. see Section 6.3.11), this domain knowledge could be shared with other testers.

6.4.4 Limitations of the Experiment

There are two main limitations to this experiment: sample size and participant experience.

Due to the small number of testers who participated in the experiment (see Section 6.2.5.1), the results cannot be generalised across the entire population of professional testers. As such, they are considered to be indicative results (not conclusive), providing an incentive to both ourselves and other researchers to replicate the experiment (following the practice of other disciplines).

The participants' experience was a limitation, as some were novices who had not designed test cases before. Although they were working as Testers and Test Leads, at the time of the experiment 72% of the group were not responsible for test case design in their organisation, as this was typically the responsibility of Business Analysts (see Section 6.3.1.4). Although this may have reduced the effectiveness of the testing that was conducted during the experiment, the results obtained still provide insight into the likelihood that the Atomic Rules approach could one day be adopted in industry.

Thus, replication of this experiment with a broader sample size, demographic and set of test programs would determine whether or not these results apply to the software testing industry.

One factor of usability that was not assessed in this experiment but that also warrants investigation is the level of confidence a tester feels when they are applying a black-box testing method, and whether this has any affect on the completeness or failure-detection effectiveness of their test cases.

6.4.5 Teaching Atomic Rules in Future

Based on the experiment results, the following four aspects of the teaching materials for the Atomic Rules approach will be improved.

In future, ***Test Matrices*** will be demonstrated as an effective approach for planning and tracking test coverage. A number of participants reported that if they were to test the programs used in this experiment again, they would use a more structured approach to testing, such as using Test Matrices (see Section 6.3.5.2). Test matrices also made data analysis in this experiment very efficient when determining which participants had applied each Atomic Rules to which program field. Thus, they would be a valuable tool to demonstrate when teaching the Atomic Rules approach.

Data analysis indicated that ***special characters*** such as spaces and symbols were not well tested (see Section 6.3.3.4). In future, additional examples that demonstrate the application of Atomic Rules to special character fields will be provided in the teaching materials.

Atomic Rules names will be revised to improve future training material**.** Although it was encouraging to find that the participants liked the structure and format of the Atomic Rules approach, particularly the Quick Reference Guide (Appendix C), the intended usage of a number of Atomic Rules were misunderstood by some participants during this experiment (see Section 6.3.6). This should be resolved by an improved naming scheme.

During the experiment, only one participant applied the ***four-step test case design process*** in the prescribed order (see Section 6.3.7.4). Although the process was demonstrated during a lecture on the approach, this may have been caused by the Quick Reference Guide not mentioning rule application order. Since the participants understood how to select test cases without the use of this process, this did not have a detrimental effect on the quality of the resulting test cases. Nonetheless, in future training this process will be added to the Quick Reference Guide, as it will likely be useful for novice testers. The four-step test selection process is also required for automation of the Atomic Rules approach.

6.5 Summary

The aim of this experiment was to compare the usability and failure-detection effectiveness of the black-box testing methods that are ‘typically’ used by testers in industry to that of the Atomic Rules representation of EP, BVA and ST. The experiment also looked at how experience and domain knowledge can affect black-box testing effectiveness.

During the PNTP testing phase, rather than doing any test planning or design prior to test execution, all participants conducted Exploratory Testing (see Section 6.3.2). Although one tester developed a random input generator in Microsoft Excel for testing the Batch Processor during this phase of testing, they supplemented this with manual Exploratory Testing. These results, combined with participant feedback gathered on questionnaires, suggests that testers in industry may initially choose to use Exploratory Testing when they are unfamiliar with the program under test, as this allows them to learn about the program and the nature of defects present, before adopting more prescriptive testing methods that facilitate better understanding and improvement of test coverage.

Of the 501 test data values that were derived by the participants during PNTP testing, only eight (1.6%) could not be derived by existing Atomic Rules from EP, BVA and ST (see Section 6.3.1.1). Thus, the Atomic Rules approach could be an excellent facilitator of 'on-the-fly' test case design during Exploratory Testing (e.g. by using the Quick Reference Guide in Appendix C). This also suggests that the Atomic Rules approach may be 'complete' in terms of its coverage of the black-box test case design rules that are used by practitioners in industry. Therefore, practitioners could use Atomic Rules to describe the approach to black-box testing they follow during Exploratory Testing. It also indicates that the Atomic Rules approach could be useful to auditors and test managers to assess the completeness of the black-box test sets that are designed by testers during Exploratory Testing (if test cases are recorded) and prescriptive black-box testing (e.g. checking that a tester's described coverage matches actual coverage). Furthermore, test managers could use Atomic Rules to assess whether testers understand each black-box testing method, to identify weak areas in a tester's knowledge that require improvement.

It was also interesting to discover that of the eight test data values that were derived during PNTP testing that were not derivable by existing Atomic Rules, four gave rise to the definition of new Atomic Rules (see Section 6.3.1.1). Since this is the aim of creation-based Systematic Method Tailoring and is supported by that approach (see Chapter 3, Section 3.2.10), this result suggests that SMT could be useful for capturing new test case design rules in industry.

The experiment results also indicate that Exploratory Testing can be an effective approach for defect detection (see Section 6.3.4.2), particularly when a tester has application solution domain knowledge in the program under test (see Section 6.3.4.3). This was evidenced by the fact that the participants, who had domain knowledge in the Address Parser but not in the Batch Processor, designed significantly more (failure-detection) effective test cases during PNTP testing of the Address Parser than of the Batch Processor. Conversely, the testers did just as well against both programs during Atomic Rules testing. This suggests that to detect significant numbers of program failures, testers either need application solution domain knowledge in the program (if they are conducting Exploratory Testing) or they need to use a prescriptive black-box testing method. This also raises the question of whether there is information about the application domain of the program under test that gives experienced 'pathological' testers clues on how to test it effectively (see Section 6.4.3), which would be an interesting topic for future research.

A comparison of the number of known program failures that were detected in general during PNTP and Atomic Rules testing indicated that significantly more failures can be detected by using the Atomic Rules

approach. Specifically, the testers in this experiment were able to detect more failures and achieve greater levels of BVA and ST coverage when they used the Atomic Rules approach. Thus, despite its potential complexity, this indicates that the Atomic Rules representation of EP, BVA and ST is more effective than, or at least comparable to, Exploratory Testing.

Interestingly, the number of years the participant's had worked as testers in industry and their current role in testing did not have any affect the level of EP, BVA and ST coverage they achieved during PNTP or Atomic Rules testing (see Section 6.3.3.5). This may have been due to the participants not having recent experience in test design (see Section 6.3.1.4). The length of their testing experience also did not affect their failure-detection effectiveness during either phase of testing (see Section 6.3.4.5). However, their role did affect the number of failures they detected during the PNTP testing phase, with Testers detecting more failures than Test Leads (see Section 6.3.4.5). This may have been due to the Testers having more recent familiarity with test case design and execution than the Test Leads (who would be likely to have more recent experience in test planning, strategies and management). Conversely, no significant difference was found in the failure-detection effectiveness achieved by Testers and Test Leads during Atomic Rules testing, suggesting that learning the Atomic Rules approach can fill the knowledge gap for Test Leads who are not currently involved in test design or execution.

It was also interesting to find that Atomic Rules from EP were capable of detecting 93% of the known failures in the Address Parser if the method was applied 'completely' (i.e. every Atomic Rule being applied to every applicable field), compared to only 54% of failures in the Batch Processor (see Section 6.3.4.1). BVA was capable of detecting 67% of Address Parser failures and 58% of Batch Processor failures. ST was equally effective against both programs, detecting 67% of known failures in each one. An interesting area for future research is to examine why some testing methods are more effective at detecting certain types of program failures and whether programming styles and program architecture have any influence on this.

Another interesting finding is that there were certain classes of input fields that were not well tested during both phases of testing, including non-alphanumeric characters and white spaces. Since the testers said they understood the importance of testing punctuation (see Section 6.3.4.3) and as they were taught how to test such fields during a presentation on the Atomic Rules approach, it is possible that they either did not recognise the requirement to test these fields, they did not know how to test them or they did not have enough time to test them. Further research is required to determine why this was the case and whether this is a common problem in industry.

Chapter 7

Conclusions and Future Work

"When you have eliminated all which is impossible, then whatever remains, however improbable, must be the truth."

Sherlock Holmes, by Sir Arthur Conan Doyle, The Adventure of the Blanched Solider, 1926.

7.1 Conclusions

The aim of this thesis was to improve the usability and failure-detection effectiveness of prescriptive and non-prescriptive approaches to black-box testing. This included identifying and resolving the following seven problems with existing definitions of black-box testing methods:

1. definition by exclusion;
2. multiple versions;
3. method overlap;
4. notational and terminological differences;
5. reliance on domain knowledge;
6. difficult to automate; and
7. difficult to audit.

Items 1 to 4 above are inherent problems with existing definitions of black-box testing methods. They are, as we and others have shown demonstrable weaknesses that are due to the manner in which the methods were defined prior to this work.

Item 5 arises since the methods currently rely on the existence of ‘perfect’ requirements that leave no gaps to be filled through the acquisition of domain knowledge. In one sense, that the test methods should be ‘application domain knowledge agnostic’ seems not unreasonable. On the other hand, the work reported here suggests that the effectiveness of black-box testing can be improved if prescriptive black-box testing methods are supported by the capture and evaluation of domain knowledge.

Items 6 and 7 are ‘derivative’ problems, in that they are a direct result of a lack of precision in existing descriptions of black-box testing methods and are therefore related to the problems with test method descriptions that are covered under items 1 to 4 above. In the authors view they are important, as they represent important uses that testing professionals may have for black-box testing methods that cannot easily be met by existing method definitions.

The Atomic Rules approach, Goal/Question/Answer/Specify/Verify and Systematic Method Tailoring were proposed as solutions to these problems. As such, they represent the main contribution of this thesis to the field of software testing. Additional contributions include the definition of *test method usability* and metrics for evaluating usability. A prototype testing tool called the Atomic Rules Testing Tool was also presented, to demonstrate that the Atomic Rules approach makes black-box testing methods more precise and easier to automate.

The Atomic Rules approach provides precise definitions of eleven different black-box testing methods, including Equivalence Partitioning, Boundary Value Analysis, Syntax Testing and combinatorial methods Each Choice, Base Choice, Orthogonal Array Testing, Heuristic Pair-Wise, All Combinations, Specification-Based Mutation Testing and the combined approaches Base Choice/Orthogonal Array Testing and Base Choice/Heuristic Pair-Wise Testing (see Appendix B). Individual Atomic Rules from EP, BVA and ST were also shown to aid test data selection for State Transition Testing, Use Case Testing and the Category Partition Method. The Atomic Rules approach, GQASV and SMT were evaluated through two university experiments, an industrial experiment and a proof-of-concept assessment.

The seven problems with black-box testing methods that are listed above were resolved as follows.

Definition by exclusion was resolved by defining explicit datatypes (e.g. integer, real, alpha) that allow the 'universe of discourse' for program inputs to be explicitly defined. This allowed prescriptive Atomic Rules to be defined for EP that select invalid equivalence classes by datatype (see *EP4* to *EP10* in Appendix B). This also partially resolved ***reliance on domain knowledge***, by ensuring that each Atomic Rule was defined to a level of detail that facilitates the design of effective and predictable black-box test sets, regardless of each tester's domain knowledge and experience.

Reliance on domain knowledge was further resolved by SMT and GQASV. SMT allows experienced testers to define new Atomic Rules during non-prescriptive (e.g. Exploratory) testing, allowing new test case design rules to be shared with other testers and for them to be reused (e.g. during Regression Testing). GQASV guides testers in the creation of precise program input field specifications (when such specifications are not readily available) and in recording domain knowledge that is used during the process, allowing that knowledge to be shared and reused. GQASV enables more effective application of the Atomic Rules approach by identifying the minimum information that is required to apply Atomic Rules to each input field under test.

Notational and terminological differences between black-box testing methods were resolved through the uniform notation of the Atomic Rules characterisation schema and four-step black-box test case design process.

Multiple versions of each black-box testing method were resolved by defining one set of Atomic Rules that cover the various versions of each method, creating a single repository of Atomic Rules. ***Method overlap*** could then be resolved by locating and eliminating redundant rules that appear in more than one method, or more than once within a method.

Difficulties with auditing the 'completeness' of black-box testing were resolved through the granularity of the Atomic Rules approach, by enabling simplified analysis of the set of Atomic Rules that were (or were not) applied to a program. The Atomic Rules approach could be used by auditors or test managers to assess the completeness of black-box test sets that are designed during Exploratory Testing (if test cases are recorded) or prescriptive black-box testing (i.e. checking that described coverage matches actual coverage) (see further discussion under Section 7.3).

Difficulties with automation were also resolved through the precise definitions provided by the Atomic Rules approach. This eventually resulted in the development of the Atomic Rules Testing Tool (see Chapter 4). ARTT currently supports automatic generation of equivalence classes and test data values for EP, BVA and ST from specifications that are input by the user. It also supports domain knowledge capture through GQASV and the creation of new Atomic Rules through creation-based SMT.

7.2 Interesting Results of the University Experiments

Two university experiments were conducted to determine whether the Atomic Rules approach improves the usability of black-box testing methods for novice testers. This compared Myers' original definition of EP and BVA (Myers 1979) to the corresponding Atomic Rules. Usability was assessed in terms of completeness, accuracy, efficiency, learnability, understandability and satisfaction. Although there were limitations to the experiment, including some inconclusive results, it still provided evidence that Atomic Rules could be an effective approach for teaching black-box testing methods to novice testers.

For example, in the first university experiment, the Atomic Rules representation of EP and BVA allowed novice testers to write more complete and accurate black-box test sets with higher levels of productivity, compared to those who used Myers' representation. More students in that year also preferred learning the Atomic Rules approach and felt they gained a better understanding of it than Myers' representation. Students in the second experiment who chose to use the Atomic Rules approach in their class assignment achieved significantly higher grades (86% on average) than who used Myers' representation (68% on average) (i.e. the difference between a 'C' and an 'A' grade).

During data analysis it became evident that the Atomic Rules approach can stifle tester creativity. Since the Atomic Rules approach is more prescriptive than Myers' definition, it did not allow the students to derive test cases based on their own domain knowledge and experience. Some of the participants in Myers' group created test cases that were not derivable from Myers' representation. Although Systematic Method Tailoring was developed as an approach for guiding testers in the definition of new Atomic Rules, it was not taught to the testers in these experiments, as it had not been invented when the first experiment was run. However, this was a feature of the industrial experiments (see Section 7.3).

The limitations of these experiments mean that the results cannot be generalised to all novice software testers. To determine whether the Atomic Rules approach improves the usability of black-box testing methods for all novice testers, these experiments would need to be rerun with more novice testers, ideally from both academia and industry and with industry-developed software.

7.3 Interesting Results of the Industry Experiment

In the industrial experiment, the usability and failure-detection effectiveness of the black-box testing methods that are 'typically' used by professional software testers (called the Practitioner Normal Testing Practice or 'PNTP') were compared to that of the Atomic Rules representations of EP, BVA and ST. Usability was evaluated in terms of completeness, efficiency (i.e. productivity), errors made (i.e. accuracy), understandability, operability, satisfaction and motivation. The experiment also looked at whether the test case design rules that are typically used by practitioners can be described as Atomic Rules.

The main limitations of this study were that the sample size was relatively small and the programs that were used in the experiment were not developed in industry. In addition, while all participants in the experiment were currently working as Test Leads, Testers or (Tester) Learning and Development Managers, none had recent experience in test design. Furthermore, test case design took place in less than a day, whereas test case design in industry can often take weeks or months (depending on the requirements and scale of the program being tested). While the results of the experiment cannot be generalised across the entire software testing industry, they do provide useful insight into how the testers in this group carry out black-box testing and evidence that the test cases industry practitioners design can described by and audited by the Atomic Rules approach.

Patterns in the derived test cases derived by the testers indicated that they conducted Exploratory Testing during the PNTP testing phase. During this phase the participants did not do any test planning or test design prior to test execution. This result, coupled with feedback from the participants, suggests that testers in industry may chose to use Exploratory Testing when they are unfamiliar with a program, as this allows them to learn about the program and the nature of defects present, before using prescriptive testing methods, which then facilitate understanding and improvement of test coverage.

It was encouraging to find that during PNTP test case generation, out of 501 (Exploratory) test data values that were derived, only eight (1.6%) could not be derived by Atomic Rules from EP, BVA and ST (see Chapter 6, section 6.3.11). Thus, the Atomic Rules approach facilitated auditing of black-box test set completeness in 98.4% of cases. This also suggests that the Atomic Rules approach could in fact be 'complete' in terms of its coverage of the types of test case design rules that are used by practitioners in industry (although this can only be confirmed through further experimentation).

The industry experiment also indicated that significantly more program failures could be detectable on average through using the Atomic Rules approach than through Exploratory Testing. They also indicated that it was capable of producing significantly more complete BVA and ST test sets. Thus, despite its potential complexity, when used by testers in industry the Atomic Rules approach could be more effective than, or at least comparable to, Exploratory Testing.

On the other hand, the results suggested that Exploratory Testing could be made more effective by using testers who have domain knowledge in the program under test. During the PNTP (i.e. Exploratory) testing phase, participants who had domain knowledge in the program under test detected significantly more

failures than those who were unfamiliar with the domain. On the other hand, when participants without domain knowledge used the Atomic Rules approach, they were able to detect just as many failures as those using Exploratory Testing with domain knowledge. This suggests that to detect significant numbers of failures, testers either need program domain knowledge (during Exploratory Testing) or the use of a prescriptive black-box testing method. A recommendation these results offer industry is that while Exploratory Testing can be a useful approach for detecting program failures when domain knowledge is present, prescriptive black-box testing methods could allow more program failures to be detected and could fill the knowledge gap for testers without domain knowledge. An interesting area for future research is whether there is information about the domain of a program that gives experienced 'pathological' testers clues on how to test it effectively.

However, the existence of the Atomic Rules approach allows the tests produced by pathological testers, or, for that matter, by exploratory testers, to be analysed. In principal, this may allow the test selection methods that are being used to be exposed and shared, with the possibility that totally new prescriptive black-box testing methods may be described and used.

Interesting, the number of years that the participants had worked as testers in industry, and their current role in testing, did not affect their coverage of Atomic Rules from EP, BVA or ST during PNTP or Atomic Rules testing. The length of their experience also did not affect their failure-detection effectiveness. On the other hand, their current role in testing did affect the number of failures they detected during PNTP testing, with Testers detecting more than Test Leads. This may be due to Testers having more recent experience with test design and execution. No significant difference was found in the failure-detection effectiveness achieved by Testers and Test Leads during Atomic Rules testing, suggesting that the Atomic Rules approach could fill the knowledge gap for Test Leads who do not have recent experience with black-box testing.

It was very encouraging to find that Atomic Rules from EP were capable of detecting 93% of the known failures in the Address Parser, compared to 54% in the Batch Processor. BVA and ST were capable of detecting between 67% of failures in the Address Parser and 58% to 67% in the Batch Processor. An interesting topic for future research would be why some Atomic Rules are more effective at detecting program failures than others and what effect program design has on this. The ideal outcome will be a mapping of program design characteristics to Atomic Rules that are most effective.

From the test cases derived by the participants on this experiment, five new Atomic Rules could be defined through SMT. They were *BVA12* and *BVA13* (select the inside boundary values of lists), *ST14* (select list values in reverse), *ST17* (add middle character to keywords) and *ST19* (reverse all fields) (see Appendix B). New Atomic Rules were also defined during the proof-of-concept evaluation of SMT. This suggests that SMT could be useful for capturing new test case design rules in industry. It would be ideal to conduct further experiments with industry practitioners, to assess whether the test cases they derive during Exploratory Testing can be described by existing Atomic Rules or whether they give rise to the definition of new Atomic Rules through SMT. In addition, it would be useful to know whether new Atomic Rules provides explicit 'skill-based' improvements to existing black-box testing methods. This could also include

an evaluation of whether practitioners can understand how to use SMT to define their own Atomic Rules, and whether they can audit the completeness of their own black-box test sets.

Future work could assess tester confidence levels when they are using prescriptive or non-prescriptive testing methods, to determine whether their confidence affects the completeness and/or failure-detection effectiveness of their resulting test cases. Another aspect to consider is whether the acquisition of domain knowledge allows testers to feel more confident during test design, and whether this in turn allows them to write more effective test cases or to do so in a more productive manner.

It would be interesting to examine whether the Atomic Rules approach and GQASV do provide a good substitute for domain knowledge in industry. For example, experimentation could be used to determine whether experienced testers find GQASV to be a useful approach for specifying program input and output fields and how much domain knowledge is required to produce 'adequate' specifications. It would also be useful to discover the most useful sources of domain knowledge (e.g. past experience in testing similar systems, programming experience, textbooks, websites; see Chapter 2, Section 2.6.2 and Chapter 3, Section 3.10.1) that enable testers to write the most effective test cases. Furthermore, after domain knowledge is collected by experienced testers using GQASV, it would be interesting to know whether novice testers (or those who are unfamiliar with the domain) could reuse that knowledge to design 'complete' test sets with the Atomic Rules approach. ARTT could provide support for this kind of investigation.

An additional area for experimentation could be to examine whether the Atomic Rules approach makes it easier for test managers to audit the completeness of the black-box test sets that are produced by testers in their teams, in order to identify when test sets are 'complete.'

7.3.1 Additional Uses of the Atomic Rules Approach in Industry

Since the Atomic Rules approach provides a single definition of each black-box testing method, it could be used by both national and international standardisation bodies to publish precise definitions of each black-box testing method. Having one standard, prescriptive definition of these methods would make it easier for organisations to demonstrate compliance to such standards and would make it easier for auditors to carry out accurate compliance assessments against testing standards. For example, the British Standard BS-EN 50128 'highly recommends' the use of EP and BVA for testing certain classes of safety critical systems (BS 50128:2001). Since that standard refers to Myers' definition of these methods, which suffers from considerable ambiguities, it is likely that both the black-box testing approach taken by organisations claiming compliance against the standard, and the compliance assessment approach that is subsequently taken by auditors, could miss certain crucially important test cases. Regulation of black-box testing may become more important in industry, as more rigorous testing standards are developed (such as the new ISO/IEC 29119 Software Testing standard[1]).

[1] As the author is a co-editor of the new ISO/IEC 29119 Software Testing standard, and is responsible for the development of part 4 (Testing Techniques), which will include definitions of black-box testing methods, it is her intention to introduce the concept of more auditable black-box testing methods descriptions into that standard.

Software testing certification boards like the International Software Testing Qualifications Board (ISTQB) and local software testing course providers (e.g. K. J. Ross & Associates) could use the Atomic Rules definition of these methods to improve their training and certification materials. Test managers could use the Atomic Rules approach to assess whether the testers in their organisation understand how to use each black-box testing method and to identify weak areas in their knowledge of the methods.

7.4 Future Improvements to the Atomic Rules Approach

A number of improvements are already planned for the Atomic Rules approach, including the following.

- The names of EP Atomic Rules that were misunderstood during the industry experiment will be enhanced, ideally through consultation with testers in industry.
- When the Atomic Rules approach is taught to testers in future, Test Matrices will be demonstrated as a useful approach for planning and tracking test coverage.
- Additional worked examples will be provided in teaching materials, to demonstrate the application of Atomic Rules to special character fields, as this was a problem area during all experiments.
- The Quick Reference Guide will be enhanced to include the four-step test case design process, as this was not followed by testers during the industrial experiment.
- Future investigation will be carried out to determine whether Atomic Rules can be defined to test input field dependencies, as this is a known limitation of the Atomic Rules approach.

7.5 Future Improvements to the Atomic Rules Testing Tool

A variety of improvements to the Atomic Rules Testing Tool were proposed in Chapter 4. These are summarised as follows.

- Automatic generation of program source code for input data validation in various languages, as this reduces the need for black-box testing by ensuring that programs only accept valid inputs.
- Implementation of Test Case Construction Rules from EP, BVA and ST, as this will enable automatic generation of complete black-box test cases.
- Enhancement of the Specification Editor to enable automatic import of BNF specifications, as this will enable more efficient specification creation.
- Enhancement of the Specification Editor to provide feedback on input field definitions for new specifications, based on the names of previously defined input fields, and automatic advice on the use of domain knowledge that was previously defined for existing specifications.
- Automatic production of abstract syntax trees for specifications, as this will provide users with a visual representation of the hierarchy in each specification they create.
- Enhancement of Atomic Rules from BVA to generate test cases for fields that repeat.

- Removal of redundant test data generated by ARTT, as this will result in more efficient testing.
- Automatic generation of test data values for testing output field partitions.
- Automatic derivation of unrestricted specifications for the CPM in the Test Specification Language, via the automatic combination of test data values that are generated through EP, BVA and ST, as well as documentation of expected results of specific combinations of test data values.
- Integration with unit testing tools like JUnit, as well as code coverage analysis tools like JCover.

It would also be beneficial to carry out experiments with ARTT with professional testers, to determine whether this tool could be useful to practitioners in industry and to determine whether it improves the efficiency of test case derivation in reality. Comparisons of ARTT to other testing case generation tools such as CaseMaker (see Chapter 2, Section 2.7.3) would also determine whether automation of the Atomic Rules approach does result in more complete and effective test sets than that which is currently supported by other black-box test case generation tools in industry.

7.6 Future Experimentation with GQASV and SMT

As GQASV and SMT only underwent a preliminary evaluation, future experimentation must be carried out before the techniques are used by the software testing industry, to determine whether the potential benefits of these approaches can be realised by professional testers. For GQASV, an experiment could be conducted to determine whether more precise input/output field definitions can be produced by professional testers that are using GQASV, as compared to the approaches they would normally use, in order to understand and properly define program input/output domains, and to determine whether the domain knowledge that is captured during the process is useful to testers in current and future projects. For SMT, a separate experiment could be carried out to identify whether new and useful Atomic Rules can be defined by professional testers through using the SMT process.

7.7 Final Word

In the words of Dikstra (1969), “Program testing can be used to show the presence of bugs, but never to show their absence!” We hope that the contributions made by this thesis provide software testers everywhere with an even better ability to detect the types of program faults that would otherwise prevent the users of their software from experiencing a ‘bug-free’ existence every time they interact with a program.

Chapter 8

Appendices

"The last project generated a ton of paper and it was still a disaster, so this project will have to generate two tons."

DeMarco and Lister, "Peopleware", 1999

Appendix A. Demonstration of the Category Partition Method

In the following, the Category Partition Method (CPM) (see Chapter 2, Section 2.2.8) is applied to a specification of a 'find' command (Figure 8-1), which was originally published by Ostrand and Balcer (1988). This specification defines one function that can be used to locate instances of a particular string within a file.

Figure 8-1: Natural language specification of a 'find' command (Ostrand & Balcer 1988).

Command:

find

Syntax:

find <pattern> <file>

Function:

The find command is used to locate one or more instances of a given pattern in a text file. All liens in the file that contain the pattern are written to standard output. A line containing the pattern in written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

Examples:

find john myfile

displays lines in the file **myfile** which contain john **john**

find "john smith" myfile

displays lines in the file **myfile** which contain **john smith**

find "john"" smith" myfile

displays lines in the file **myfile** which contain **john" smith**

The CPM consists of six steps, as follows.

In step one, the specification is decomposed into individual functional units that can each be tested separately. Since this specification contains one functional unit, this step is already complete.

In step two, input fields called 'categories' and equivalence classes called 'choices' for each category are defined[1]. The specification mentions two input fields, <*pattern*> and <*file*>. The characteristics of <*pattern*>, which are explicitly defined in the specification, are as follows:

1. length of <*pattern*> must not exceed maximum line length, assumed to be 80 characters;
2. if <*pattern*> contains a white space it must be enclosed in quotes; and
3. if <*pattern*> contains an embedded quote it must be replaced with two quotes in a row.

Other characteristics of <*pattern*> that were not described in the specification, but that an experienced tester may wish to consider, are whether:

4. quoted patterns always have to include blank characters;
5. several successive quotes are permitted in a pattern.

The <*file*> field can be tested as an 'environmental variable' *file contents* or as an 'input parameter' *file name*. As an environmental variable, the following characteristics can be defined for *file contents*:

6. the number of occurrences of <*pattern*> in the file;
7. the number of occurrences of <*pattern*> on a line that contains it (called the 'target' line);
8. the maximum length of the file;
9. the file type (e.g. text, binary, executable);
10. whether <*pattern*> overlaps itself in a line of the file; and
11. whether <*pattern*> extends over more than one line.

If <*file*> is considered to be an input parameter *file name*, the following choices (which were not identified by (Ostrand & Balcer 1988)) could be identified through the application of EP and BVA:

12. whether any specific ASCII characters are not allowed in the file name, such as / \ : * ? " < > | as these are restricted characters in many operating systems; and
13. the maximum length of the file name (i.e. 80 characters or greater than 80 characters).

In step three, 'constraints' can be added to each choice, which dictate how a choice in one category can restrict choices in another. This reduces the number of test frames that are generated in step four and prevents 'contradictory test frames' (i.e. impossible combinations of choices) from being created, such as combining an empty pattern with a pattern that is quoted (e.g. see Figure 8-2). Constraints are defined by annotating contradictory choices with a '*property*' statement, and by adding '*selector expressions*' that limit whether a particular choice can be included in a test case, based on the values of other choices. A property statement is expressed as *[property A, B, …]*, where *A, B, …* are property names, while *selector expressions* are expressed as *[if A]* or *[if A and B]*. For example, if a test contains a pattern which is empty (i.e. [property Empty]) then the pattern cannot be quoted (i.e. [if NonEmpty]). To limit the scope of their

[1] The Goal/Question/Answer/Specify/Verify approach could be used at step 2, when defining the contents of each input field.

example, in step 3 Ostrand and Balcer only included choices and categories that were mentioned in the original specification (i.e. pattern size (choice 1), quoting (2), embedded white spaces (3), embedded quotes (4 and 5) and file name (6 to 11)). Thus, choices 12 and 13 are not covered in the example below.

Figure 8-2: Example of a contradictory test frame (Ostrand & Balcer 1988).

Pattern size: empty
Quoting: pattern is quoted
Embedded blanks: several embedded white spaces
Embedded quotes: no embedded quotes
File name: good file name
Number of occurrences of pattern in file: none
Pattern occurrences on the target line: one

In step four, categories, choices and constraints are documented in a 'restricted test specification' that is expressed in the Test Specification Language (TSL) (Figure 8-3). Step 3 can be skipped, resulting in an 'unrestricted test specification' (see Figure 8-4), which does not prevent contradictory test frames.

Figure 8-3: Restricted test specification for the find command, expressed in the Test Specification Language (Ostrand & Balcer 1988).

Parameters:	
Pattern size:	
empty	[property Empty]
single character	[property NonEmpty]
many characters	[property NonEmpty]
longer than any line in the file	[property NonEmpty]
Quoting:	
pattern is quoted	[property Quoted]
pattern is not quoted	[if NonEmpty]
pattern is improperly quoted	[if NonEmpty]
Embedded white spaces:	
no embedded white spaces	[if NonEmpty]
one embedded white space	[if NonEmpty and Quoted]
several embedded white spaces	[if NonEmpty and Quoted]
Embedded quotes:	
no embedded quotes	[if NonEmpty]
one embedded quote	[if NonEmpty]
several embedded quotes	[if NonEmpty]
File name:	
good file name	
no file with this name	
omitted	
Environments:	
Number of occurrences of pattern in the file:	
none	[if NonEmpty]
exactly one	[if NonEmpty] [property Match]
more than one	[if NonEmpty] [property Match]
Pattern occurrences on target line:	
# assumes line contains the pattern	
one	[if Match]
more than one	[if Match]

Figure 8-4: Unrestricted test specification for the 'find' command, expressed in the Test Specification Language (# denotes comments) (Ostrand & Balcer 1988).

```
Parameters:
    Pattern size:
        empty
        single character
        many characters
        longer than any line in the file
    Quoting:
        pattern is quoted
        pattern is not quoted
        pattern is improperly quoted
    Embedded white spaces:
        no embedded white spaces
        one embedded white space
        several embedded white spaces
    Embedded quotes:
        no embedded quotes
        one embedded quote
        several embedded quotes
    File name:
        good file name
        no file with this name
        omitted
Environments:
    Number of occurrences of pattern in the file:
        none
        exactly one
        more than one
    Pattern occurrences on target line:
    # assumes line contains the pattern
        one
        more than one
```

In step four, an initial set of 'test frames' are designed by taking the Cartesian product of choices in the test specification, excluding those that cannot be combined due to constraints that were placed on them in step three. The restricted specification depicted in Figure 8-3 results in 678 test frames, which Ostrand and Balcer considered too high for the (relatively simple) find command (Ostrand & Balcer 1988).

Thus, in step five, the restricted specification can be further refined by adding two additional tags: *[error]* and *[single]* (see Figure 8-5). The *[error]* tag reduces the number of redundant test frames by identifying choices that result in invalid test cases, ensuring that only one test frame is derived for each such choice (this is synonymous with the one-to-one test case design approaches for EP and BVA that are defined in (BS 7925-2)). For example, Ostrand and Balcer recognised that the choice "no file with this name" would result in an invalid test frame that the program should reject, regardless of any other valid or invalid choices; thus, it could be tagged with *[error]* so that it was not tested in combination with other choices. In addition, *[single]* can be used to mark choices that do not have to be combined with other choices, which can be used to reduce the number of tests executed against certain choices that are of a lower risk of affecting program correctness.

Applying the *[error]* tag to the restricted specification for the find command reduces the number of frames to 125, while tagging three choices with *[single]* reduces this to 40 frames (Figure 8-5).

Figure 8-5: Refined version of the restricted test specification for the 'find' command, expressed in the Test Specification Language (Ostrand & Balcer 1988).

```
Parameters:
    Pattern size:
        empty                                   [property Empty]
        single character                        [property NonEmpty]
        many characters                         [property NonEmpty]
        longer than any line in the file        [property NonEmpty]
    Quoting:
        pattern is quoted                       [property Quoted]
        pattern is not quoted                   [if NonEmpty]
        pattern is improperly quoted            [error]
    Embedded white spaces:
        no embedded white spaces                [if NonEmpty]
        one embedded white space                [if NonEmpty and Quoted]
        several embedded white spaces           [if NonEmpty and Quoted]
    Embedded quotes:
        no embedded quotes                      [if NonEmpty]
        one embedded quote                      [if NonEmpty]
        several embedded quotes                 [if NonEmpty] [single]
    File name:
        good file name
        no file with this name                  [error]
        omitted                                 [error]
Environments:
    Number of occurrences of pattern in the file:
        none                                    [if NonEmpty] [single]
        exactly one                             [if NonEmpty] [property Match]
        more than one                           [if NonEmpty] [property Match]
    Pattern occurrences on target line:
    # assumes line contains the pattern
        one                                     [if Match]
        more than one                           [if Match] [single]
```

In step six, a final set of test frames are generated by again taking the Cartesian product of the choices that are defined in the test specification, excluding those that cannot be combined due to constraints that were placed on them in steps three and five. Each test frame is then populated with test data, resulting in one test case. The test cases can be annotated with additional information, such as a test case identification number, a key listing the choices covered from each category, any commands that are required as preconditions to set up the test, the test data that will be used during test execution and the expected result of the test (Figure 8-6).

Figure 8-6: Example test case generated from the restricted specification for the find command (Ostrand & Balcer 1988).

Test Frame:

Test Case 28: (Key = 3.1.3.2.1.2.1)

Pattern size: many characters

Quoting: pattern is quoted

Embedded blanks: several embedded white spaces

Embedded quotes: one embedded quote

File name: good file name

Number of occurrences of pattern in file: exactly one

Pattern occurrences on the target line: one

Command to set up the test:

copy /testing/sources/case_28 testfile

find command to perform the test:

find “has “” one quote” testfile

Instructions for checking the test:

The following lien should be displayed:

This line has “ one quote on it

Appendix B. Atomic Rules for Black-Box Testing

This section contain Atomic Rules for Equivalence Partitioning (Table 8-1 to Table 8-4), Boundary Value Analysis (Table 8-6 to Table 8-9), Syntax Testing (Table 8-10 to Table 8-14) and combinatorial testing methods All Combinations, Each Choice, Base Choice, Orthogonal Array Testing and Specification-Based Mutation Testing, including Single-Substitution Mutation, Multiple-Substitution Mutation, All-Permutations Mutation and All-Combinations Mutation (Table 8-16 to Table 8-25).

B.1 Equivalence Partitioning

The following tables contain Atomic Rules that have been defined for Equivalence Partitioning.

Table 8-1: Atomic Rules for Equivalence Partitioning.

Attribute	Values	Values	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning
Number	EP1	EP2	EP3	EP4
Identifier	LLBS	GUBS	LUBS	IR
Name	Less Than Lower Boundary Selection	Greater Than Upper Boundary Selection	Lower to Upper Boundary Selection	Integer Replacement
Description	Select an equivalence class containing values below the lower boundary of a field	Select an equivalence class containing values above the upper boundary of a field	Select an equivalence class containing values between the boundaries of a field (including the on-boundary values)	Select an equivalence class containing every integer value (other than those in the valid set, if applicable)
Source	(Myers 1979)	(Myers 1979)	(Myers 1979)	N/A
Rule Type	DSSR	DSSR	DSSR	DSSR
Set Type	Range	Range	Range	List or Range
Valid or Invalid	Invalid	Invalid	Valid	Invalid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	All
Test Datatype	Same as original	Same as original	Same as original	Integer
Test Data Length	Same as original	Same as original	Same as original	Max
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-2: Atomic Rules for Equivalence Partitioning (continued).

Attribute	Values	Values	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning
Number	EP5	EP6	EP7	EP8
Identifier	RNR	SAR	MAR	MANR
Name	Real Number Replacement	Single Alpha Replacement	Multiple Alpha Replacement	Multiple Alphanumeric Replacement
Description	Select an equivalence class containing every real value (other than those in the valid set, if applicable)	Select an equivalence class containing every single alpha value (other than those in the valid set, if applicable)	Select an equivalence class containing multiple alpha values (other than those in the valid set, if applicable)	Select an equivalence class containing multiple alphanumeric values (other than those in the valid set, if applicable)
Source	(BS 7925-2)	(BS 7925-2)	(BS 7925-2)	N/A
Rule Type	DSSR	DSSR	DSSR	DSSR
Set Type	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Invalid
Original Datatype	All	All	All	All
Test Datatype	Real	Single Alpha	Multiple Alpha	Multiple Alphanumeric
Test Data Length	Max	1	Max	Max
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-3: Atomic Rules for Equivalence Partitioning (continued).

Attribute	Values	Values	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning
Number	EP9	EP10	EP11	EP12
Identifier	SNAR	MNAR	NIR	VLS
Name	Single Non-Alphanumeric Replacement	Multiple Non-Alphanumeric Replacement	Null Item Replacement	Valid List Selection
Description	Select an equivalence class containing a single non-alphanumeric value (other than those in the valid set, if applicable)	Select an equivalence class containing multiple non-alphanumeric values (other than those in the valid set, if applicable)	Select an equivalence class containing a Null value	Select an equivalence class containing all values in the specified list
Source	N/A	N/A	N/A	(Myers 1979)
Rule Type	DSSR	DSSR	DSSR	DSSR
Set Type	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Valid
Original Datatype	All	All	All	All
Test Datatype	Single Non-Alphanumeric	Multiple Non-Alphanumeric	Null	Same as original
Test Data Length	1	Max	0	Max
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-4: Atomic Rules for Equivalence Partitioning (continued).

Attribute	Values	Values	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning	Equivalence Partitioning
Number	EP13	EP14	EP15	EP16
Identifier	RDVS	VTCCMin	ITCCMax	ITCCMin
Name	Random Data Value Selector	Valid Test Case Constructor – Minimised	Invalid Test Case Constructor – Maximised	Invalid Test Case Constructor – Minimised
Description	Selects a random value from an equivalence class	Construct the minimum number of tests required to cover all valid values from all valid equivalence classes	Construct one test for each invalid test data value (i.e. one field is assigned an invalid value selected from an invalid equivalence class while all others are assigned nominal values)	Construct the minimum number of test cases reuiqred to cover all invalid values (i.e. all fields in each test are assigned invalid values)
Source	N/A	(Myers 1979)	(Myers 1979)	(Myers 1979)
Rule Type	DISR	TCCR	TCCR	TCCR
Set Type	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Depends on whether the class is valid or invalid	Valid	Invalid	Invalid
Original Datatype	All	All	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	Max	Max	Max	Max
# Fields Populated	1	*n*, where n is the number of input fields	*n*, where n is the number of input fields	*n*, where n is the number of input fields
# Tests Derived	0	*1 to m*, where *m* = # valid classes selected	*m*, where *m* = # invalid classes selected	*m*, where *m* = # invalid classes selected

Table 8-5: Atomic Rules for Equivalence Partitioning (continued).

Attribute	Values	Values
Test Method	Equivalence Partitioning	Equivalence Partitioning
Number	EP17	EP18
Identifier	NOM	VTCCMax
Name	Nominal Data Value Selector	Valid Test Case Constructor - Maximised
Description	Selects the nominal (i.e. mid-point) value from an equivalence class	Construct one test for each valid test data value (i.e. one field is assigned a valid value selected from a valid equivalence class while all others are assigned nominal values)
Source	N/A	(Myers 1979)
Rule Type	DISR	TCCR
Set Type	List or Range	List or Range
Valid or Invalid	Depends on whether the class is valid or invalid	Valid
Original Datatype	All	All
Test Datatype	Same as original	Same as original
Test Data Length	Max	Max
# Fields Populated	1	*n*, where n is the number of input fields
# Tests Derived	0	*1 to m*, where *m* = # valid classes selected

B.2 Boundary Value Analysis

The following tables contain definitions of Atomic Rules for Boundary Value Analysis.

Table 8-6: Atomic Rules for Boundary Value Analysis.

Attribute	Values	Values	Values	Values
Test Method	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis
Number	BVA1	BVA2	BVA3	BVA4
Identifier	LBM	LB	LBP	UBM
Name	Lower Boundary – Selection	Lower Boundary Selection	Lower Boundary + Selection	Upper Boundary – Selection
Description	Select value just below the lower boundary of an equivalence class	Select a value on the lower boundary of an equivalence class	Select a value just above the lower boundary of an equivalence class	Select a value just below the upper boundary of an equivalence class
Source	(BS 7925-2)	(Myers 1979)	(Myers 1979)	(Myers 1979)
Rule Type	DISR	DISR	DISR	DISR
Set Type	Range	Range	Range	Range
Valid or Invalid	Depends on partition validity	Depends on partition validity	Depends on partition validity	Depends on partition validity
Original Datatype	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original	Same as original
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-7: Atomic Rules for Boundary Value Analysis (continued).

Attribute	Values	Values	Values	Values
Test Method	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis
Number	BVA5	BVA6	BVA7	BVA8
Identifier	UB	UBP	FLIS	LLIS
Name	Upper Boundary Selection	Upper Boundary + Selection	First List Item Selection	Last List Item Selection
Description	Select a value on the upper boundary of an equivalence class	Select a value just above the upper boundary of an equivalence class	Select the first item in a list	Select the last item in a list
Source	(Myers 1979)	(BS 7925-2)	(Myers 1979)	(Myers 1979)
Rule Type	DISR	DISR	DISR	DISR
Set Type	Range	Range	List	List
Valid or Invalid	Depends on partition validity	Depends on partition validity	Valid	Valid
Original Datatype	Integer, Real, Single Alpha, Single Non-Alphanumeric	Integer, Real, Single Alpha, Single Non-Alphanumeric	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original	Same as original
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-8: Atomic Rules for Boundary Value Analysis (continued).

Attribute	Values	Values	Values	Values
Test Method	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis	Boundary Value Analysis
Number	BVA9	BVA10	BVA11	BVA12
Identifier	MIR	AFLLM1	ALLM1	SLIS
Name	Missing Item Replacement	Attempt First List Item – Selection	Attempt Last List Item + Selection	Second List Item Selection
Description	Replace a field with null	Attempt to select a list item before the first item in a list	Attempt to select a list item after the last item in a list	Selects the second item in a list
Source	(Myers 1979)	N/A	N/A	N/A
Rule Type	DISR	DISR	DISR	DISR
Set Type	List or Range	List	List	List
Valid or Invalid	Invalid	Invalid	Invalid	Valid
Original Datatype	All	All	All	All
Test Datatype	Null	Same as original	Same as original	Same as original
Test Data Length	0	Same as original	Same as original	Same as original
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-9: Atomic Rules for Boundary Value Analysis (continued).

Attribute	Values
Test Method	Boundary Value Analysis
Number	BVA13
Identifier	LLIS
Name	Second Last List Item Selection
Description	Selects the second last item in a list
Source	N/A
Rule Type	DISR
Set Type	List
Valid or Invalid	Valid
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	1
# Tests Derived	0

B.3 Syntax Testing

The following tables contain definitions of Atomic Rules for Syntax Testing.

Table 8-10: Atomic Rules for Syntax Testing.

Attribute	Values	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST1	ST2	ST3	ST4
Identifier	RMLC	RPLC	AECE	RMFC
Name	Remove last character	Replace last character	Add extra character to end of field	Remove first character
Description	Remove the last character of an input string	Replace the last character of a string with an invalid value	Add an extra character to the end of a string	Remove the first character of a string
Source	(Beizer 1990, Marick 1995)	(Marick 1995)	(Beizer 1995, Marick 1995)	N/A
Rule Type	DISR	DISR	DISR	DISR
Set Type	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Invalid
Original Datatype	All	All	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	m - 1, where m is the original field length	Same as original	m + 1, where m is the original field length	m - 1, where m is the original field length
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-11: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST5	ST6	ST7	ST8
Identifier	RPFC	AECS	ULL	LUL
Name	Replace first character	Add extra character to start of field	Uppercase a lowercase letter	Lowercase an uppercase letter
Description	Replace the first character of a string with an invalid value	Add an extra character to the start of a string	Change the case of a uppercase letter to lowercase	Change the case of a lowercase letter to uppercase
Source	N/A	N/A	(Marick 1995)	(Marick 1995)
Rule Type	DISR	DISR	DISR	DISR
Set Type	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Invalid
Original Datatype	All	All	Alpha	Alpha
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	m + 1, where m is the original field length	Same as original	Same as original
# Fields Populated	1	1	1	1
# Tests Derived	0	0	0	0

Table 8-12: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST9	ST10	ST11	ST12
Identifier	NAI	DF	AAF	SELA
Name	Null all input	Duplicate Field	Add a field	Select Each List Alternative
Description	Construct a test case that is empty	Construct a test case that has one field duplicated (all other fields are assigned their nominal value)	Construct a test case that contains a new field (contents of new field must be defined, possibly using GQAS)	For a specification with a list, create test cases where each alternative in each list is selected once (all other fields are assigned nominal values)
Source	(Beizer 1990)	(BS 7925-2, Beizer 1995)	(BS 7925-2, Beizer 1995)	(Marick 1995)
Rule Type	TCCR	TCCR	TCCR	TCCR
Set Type	List or Range	All	All	All
Valid or Invalid	Invalid	Invalid	Invalid	Valid
Original Datatype	All	All	All	All
Test Datatype	Null	Same as original	Same as original	Same as original
Test Data Length	0	Same as original	Same as original	Same as original
# Fields Populated	n, where n is the number of specification fields	n, where n is the number of specification fields	n, where n is the number of specification fields	n, where n is the number of specification fields
# Tests Derived	1	1	1	p, where p is the number of alternatives

Table 8-13: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST13	ST14	ST15	ST16
Identifier	SALA	SALAR	RR	SC
Name	Select All List Alternatives	Select All List Alternatives in Reverse Order	Reference Replacement	Syntax Cover
Description	Select every alternative from a list in the one test	Select every alternative from a list in the reverse order in the one test	For non-terminal fields that references other terminals, create a test case in which the non-terminal references itself	Construct a set of test cases which link-cover the syntax graph of the specification under test
Source	(Marick 1995)	(Marick 1995)	(Marick 1995)	(Beizer 1995, Hetzel 1988)
Rule Type	DISR	DISR	TCCR	TCCR
Set Type	All	All	All	All
Valid or Invalid	Invalid	Invalid	Invalid	Valid
Original Datatype	All	All	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original	Same as original
# Fields Populated	1	1	n, where n is the number of specification fields	n, where n is the number of specification fields
# Tests Derived	0	0	q, where q is the number of references to other non-terminals	r, where r is the number of basis paths (Pressman 1992)

Table 8-14: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values
Test Method	Syntax Testing	Syntax Testing	Syntax Testing
Number	ST17	ST18	ST19
Identifier	AMC	RMC	RF
Name	Add Middle Character	Remove Middle Character	Reverse All Fields
Description	Adds a character to the middle of a test data value	Removes a character from the middle of a test data value	Constructs a test case in which all input fields are reversed
Source	N/A	N/A	N/A
Rule Type	DIMR	DIMR	TCCR
Set Type	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All	All	All
Test Datatype	Any	Any	Same as original
Test Data Length	m - 1, where m is the original field length	m + 1, where m is the original field length	Same as original
# Fields Populated	1	1	1
# Tests Derived	0	0	1

B.4 State Transition Testing

The following Data-Set Selection Rule was defined for supporting State Transition Testing (see Chapter 3, Section 3.6.1).

Table 8-15: An Atomic Rule for State Transition Testing.

Attribute	Values
Test Method	State Transition Testing
Number	STT1
Identifier	ILS
Name	Invalid List Selection
Description	Selects an equivalence class containing the set of all input values for a state transition diagram that are valid at all other states other than the current state
Source	N/A
Rule Type	DSSR
Set Type	List or Range
Valid or Invalid	Invalid
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Max
# Fields Populated	1
# Tests Derived	0

B.5 Combinatorial Testing Methods

The following tables contain definitions of Atomic Rules for various combinatorial testing methods.

Table 8-16: An Atomic Rule for the combinatorial testing method All Combinations.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT1
Identifier	AC
Name	All Combinations
Description	Construct every possible combination of test data values, which may be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Grindal, Lindström, Offutt & Andler 2004)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	Approximately $\prod_{i=1}^{N} V_i$ test cases, where N is the number of parameters in the input string and where each parameter has V_i values.

Table 8-17: An Atomic Rule for the combinatorial testing method Each Choice.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT2
Identifier	EC
Name	Each Choice
Description	Construct a test of test cases by including test data values for each input field in at least one test case. The test data values may be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Ammann & Offutt 1994)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	$Max_{i=1}^{N} V_i$ test cases, where N is the number of parameters in the input string and where each parameter has V_i values.

Table 8-18: An Atomic Rule for the combinatorial testing method Base Choice.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT3
Identifier	BC
Name	Base Choice
Description	Select a 'base' test case in which each input field is assigned one test data value. Construct new test cases by varying the test data values of each field of the base test case one at a time. The values for the base test case and the varying values may be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Ammann & Offutt 1994)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	At least $1+\sum_{i=1}^{N}(V_i-1)$ test cases, where N is the number of parameters in the input string and where each parameter has V_i values.

Table 8-19: An Atomic Rule for the combinatorial testing method Orthogonal Array Testing (also known as Pair-Wise Testing).

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT4
Identifier	OA
Name	Orthogonal Array Testing
Description	Construct an orthogonal array of test data values, which may be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Mandl 1985)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	Approximately V_i^2 test cases, where $V_i = Max_{j=1}^{N} V_j$ where N is the number of parameters in the input string and where each parameter has V_i values.

Table 8-20: Atomic Rules for the combined combinatorial testing method Base Choice + Orthogonal Array Testing.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT5
Identifier	BC+OAT
Name	Combined Strategy: Base Choice + Orthogonal Array Testing
Description	Apply the base-choice rule first to select a set of base-choice test cases and then create an orthogonal array of the values that were selected for each field of each test case.
Source	(Grindal, Lindström, Offutt & Andler 2004)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	Approximately V_i^2 test cases, where $V_i = Max_{j=1}^{N} V_j$ where N is the number of parameters in the input string and where each parameter has V_i values that were selected using the base-choice approach.

Table 8-21: Atomic Rules for the combined combinatorial testing method Base Choice + Heuristic Pair-Wise Testing.

Attribute	Definition
Test Method	Combinatorial Testing
Number	CT6
Identifier	BC+HPW
Name	Combined Strategy: Base Choice + Heuristic Pair-Wise
Description	Apply the base-choice TCCR *CT3* to select a set of base-choice test cases and then use the 'heuristic pair-wise' algorithm to them to create a new set of test cases from them.
Source	(Grindal, Lindström, Offutt & Andler 2004)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	Approximately V_i^2 test cases, where $V_i = Max_{j=1}^{N} V_j$ where N is the number of parameters in the input string and where each parameter has V_i values that were selected using the base-choice approach.

Table 8-22: An Atomic Rule for the combinatorial testing method Specification-Based Mutation Testing.

Attribute	Definition
Test Method	Specification-Based Mutation Testing
Number	SBMT1
Identifier	SSM
Name	Single-Substitution Mutation
Description	Construct a base test case by selecting one a valid test data value for each input field. Then, construct a series of mutant test cases by substituting one field for another field, one substitution per test case. Repeat this process until every field has been substituted for every other field. The test data values in the base test case can be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Murnane & Reed 2001)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameter in the input string
# Tests Derived	$n(n-1)$ test cases, where *n* is the number of parameters in the input string.

Table 8-23: An Atomic Rule for the combinatorial testing method Specification-Based Mutation Testing.

Attribute	Definition
Test Method	Specification-Based Mutation Testing
Number	SBMT2
Identifier	MSM
Name	Multiple-Substitution Mutation
Description	Construct a base test case by selecting one a valid test data value for each input field. Then, construct a series of mutant test cases by substituting one field for another *n* other field, with *n* substitutions per test case, where *n* is the number of fields in the test case. Repeat this field until every pair of fields has been substituted for every other pair of fields. The test data values in the base test case can be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Murnane & Reed 2001)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	$O(n^m)$ test cases, where *n* is the number of parameters in the input string and m is the number of parameters substituted per mutant.

Table 8-24: An Atomic Rule for the combinatorial method Specification-Based Mutation Testing.

Attribute	Definition
Test Method	Specification-Based Mutation Testing
Number	SBMT3
Identifier	APM
Name	All-Permutations Mutation
Description	Construct a base test case by selecting one a valid test data value for each input field. Then, construct all permutations of the base test case. The test data values in the base test case can be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Murnane & Reed 2001)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	$P(n,r) = n*(n-1)*(n-2)*...*(n-r+1)$ $= \frac{n!}{(n-r)!}$ where *n* is the number of parameters in the entire input string, and *r* is the number of parameters that are included in each test case. *n* = *r* iff all parameters are included in each test case.

Table 8-25: An Atomic Rule for the combinatorial testing method Specification-Based Mutation Testing.

Attribute	Definition
Test Method	Specification-Based Mutation Testing
Number	SBMT4
Identifier	ACM
Name	All-Combinations Mutation
Description	Construct a base test case by selecting one a valid test data value for each input field. Then, construct all combinations of the base test case. The test data values in the base test case can be selected by the Data-Item Selection Rules and Data-Item Manipulation Rules of other black-box testing methods.
Source	(Murnane & Reed 2001)
Rule Type	TCCR
Set Type	List or Range
Valid or Invalid	Depends on whether rule is applied to valid or invalid values
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	*n*, where *n* is the number of parameters in the input string
# Tests Derived	$C(n,r) = \frac{n!}{(n-r)!r!}$ where *n* is the number of parameters in the entire input string, and *r* is the number of parameters that are included in each test case. *n* = *r* iff all parameters are included in each test case.

Appendix C. Atomic Rules' Quick Reference Guide

The 'Quick Reference Guide' that lists Atomic Rules from EP, BVA and ST, which was presented to participants of the industrial experiment, are shown below (Table 8-26, Table 8-27 and Table 8-28). Five Atomic Rules from ST are not included, as these additional Atomic Rules were added to this method after the experiment was complete.

Table 8-26: Atomic Rules for Equivalence Partitioning.

#	Rule ID	Rule Name	Rule Type	Valid or Invalid
EP1	LLBS	Less Than Lower Boundary Selection	DSSR	Invalid
EP2	GUBS	Greater Than Upper Boundary Selection	DSSR	Invalid
EP3	LUBS	Lower to Upper Boundary Selection	DSSR	Valid
EP4	IR	Integer Replacement	DSSR	Invalid
EP5	RNR	Real Number Replacement	DSSR	Invalid
EP6	SAR	Single Alpha Replacement	DSSR	Invalid
EP7	MAR	Multiple Alpha Replacement	DSSR	Invalid
EP8	MANR	Multiple Alphanumeric Replacement	DSSR	Invalid
EP9	SNAR	Single Non-Alphanumeric Replacement	DSSR	Invalid
EP10	MNAR	Multiple Non-Alphanumeric Replacement	DSSR	Invalid
EP11	NIR	Null Item Replacement	DISR	Invalid
EP12	VLIS	Valid List Item Selection	DISR	Valid
EP13	RDVS	Random Data Value Selector	DISR	Depends
EP14	VTCC	Valid Test Case Constructor	TCCR	Valid
EP15	ITCMax	Invalid Test Case Constructor – Maximised	TCCR	Invalid
EP16	ITCMin	Invalid Test Case Constructor – Minimised	TCCR	Invalid

Table 8-27: Atomic Rules for Boundary Value Analysis.

#	Rule ID	Rule Name	Rule Type	Valid or Invalid
BVA1	LBM	Lower Boundary – Selection	DISR	Invalid
BVA2	LB	Lower Boundary Selection	DISR	Valid
BVA3	LBP	Lower Boundary + Selection	DISR	Valid
BVA4	UBM	Upper Boundary – Selection	DISR	Valid
BVA5	UB	Upper Boundary Selection	DISR	Valid
BVA6	UBP	Upper Boundary + Selection	DISR	Invalid
BVA7	FLIS	First List Item Selection	DISR	Valid
BVA8	LLIS	Last List Item Selection	DISR	Valid
BVA9	MIR	Missing Item Replacement	TCCR	Invalid

Table 8-28: Atomic Rules for Syntax Testing.

#	Rule ID	Rule Name	Rule Type	Valid or Invalid
ST1	RMLC	Remove last character	DIMR	Invalid
ST2	RPLC	Replace last character	DIMR	Invalid
ST3	AECE	Add extra character to end	DIMR	Valid
ST4	RMFC	Remove first character	DIMR	Invalid
ST5	RPFC	Replace first character	DIMR	Invalid
ST6	AECS	Add extra character to start	DIMR	Invalid
ST7	ULL	Uppercase a lowercase letter	DIMR	Invalid
ST8	LUL	Lowercase an uppercase letter	DIMR	Invalid
ST9	NAI	Null all input	TCCR	Depends
ST10	DF	Duplicate field	TCCR	Invalid
ST11	AAF	Add a field	TCCR	Invalid
ST12	SELA	Select each list alternative	TCCR	Valid
ST13	SALA	Select all list alternatives	TCCR	Invalid
ST14	SALAR	Select all list alternatives in reverse order	TCCR	Invalid

Appendix D. Known Failures in the Address Parser

The table below lists all known failures in the Address Parser program, which was used in the industrial experiment. This includes a description of the failure, an example input that is capable of detecting a failure, and a mapping to the Atomic Rules that can be used to detecting the failure.

Table 8-29: Known defects in the Address Parser program.

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
1	Address with missing unit/flat/rsd identifier and missing space before unit number is accepted as correct	100 / 200 Main Road Eltham 3095.	*Following address appears in Addresses-Correct.txt:* 100 / 200 Main Rd Eltham 3095	BVA9 or EP11 with EP16
2	Missing unit/flat/rsd at start of address and missing separator (/ or ,), or invalid symbol in place of street name or street type, or replacing street name with symbol results, all result in error messages of street not found, invalid suburb and no full stop at end of address	100 200 Main Road Eltham 3095. 100 # Road Eltham 3095. 100 Main # Eltham 3095.	*Following errors reported to error log:* Error... Street not found. Error... There was no full stop at the end of the address. Error... Invalid suburb entered.	EP1, EP2, EP4, EP5, EP6, EP7, EP8, EP9, EP10, EP11, BVA1, BVA6, BVA9, ST1, ST2, ST3, ST4, ST5, ST6, ST10, ST11, ST13, ST14
3	Correct RSD addresses that include a unit and house number are output to correct address file without the RSD tag	RSD 100 / 100 Main Street Eltham 3095.	*Following address appears in Addresses-Correct.txt:* 100 / 100 Main St Eltham 3095.	EP12, ST12
4	Missing unit or house number results in error message of incorrect spaces after house number	UNIT 100 / Main Road Eltham 3095.	*Following errors reported to error log:* Error... Incorrect spaces after unit or flat symbol. Error... Number has too few digits. Must have 3. Error... Incorrect spaces after house or unit number.	EP1, EP11, BVA9
5	Multiple spaces or invalid data after C/- or C/o results in error message of illegal spaces before address	C/-Main Road Eltham 3095. C/o1 Main Road Greensborough 3088. C/oa Main Road Greensborough 3088. C/o Main Road Greensborough 3088.	*Following error reported to error log:* Error... Illegal spaces before address.	EP1, EP2, EP11, BVA3, BVA9, ST3, ST6, ST10
6	Forward slash and hyphen are accepted in place of full stop at end of address, which sit just above and below full stop in ASCII (this is a seeded fault)	100 Main Street Eltham 3095/	*Following addresses appears in Addresses-Correct.txt:* 100 Main St Eltham 3095. 100 Main St Eltham 3095.	EP9 or BVA1 and BVA6
7	Valid postcode Ivanhoe East is rejected (this is a seeded fault). Error message is also incorrectly reported when multiple spaces are entered after suburb, or missing postcode, or missing postcode and extra spaces between suburb and postcode.	100 Main Road Ivanhoe East 3079. 100 Main Road Eltham 3095. 100 Main Road Eltham .	*Following error reported to error log:* Error... Postcode does not match suburb.	EP2, EP3, EP4, EP5, EP6, EP7, EP11, EP12, EP13, BVA3, BVA9, ST10, ST12
8	Address with missing street name or street type is accepted as valid, correct address output file is missing street name and type, and no error is reported	100 Road Eltham 3095. UNIT 100 / 100 Main Greensborough 3088.	*Following address appears in Addresses-Correct.txt:* 100 Eltham 3095. UNIT 100 / 100 Main Greensborough 3088.	EP1, EP11, BVA1, BVA4, BVA9

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
Continued from previous page…				
9	Accepts greater than 40 characters in street name	100 aaaaaaaaaabbbbbbbbbbccccccccccddddddddde St Eltham 3095.	*Following address appears in Addresses-Correct.txt:* 100 Aaaaaaaaaabbbbbbbbbbccccccccccddddddddde St Eltham 3095.	EP2, BVA6
10	The street *Road St* is rejected as invalid, while other variants such as *Court St*reet are accepted. This error is also detected by multiple spaces after street and missing street name.	*Rejected:* 100 Road St Eltham 3095. *Accepted:* 100 Court St Eltham 3095. Also detected by: 100 Main Road Eltham 3095. 100 Main Eltham 3095.	*Following error reported to errror log:* Error... Invalid suburb entered.	EP1, EP2, EP3, EP5, EP6, EP7, EP11, EP12, BVA1, BVA2, BVA3, BVA5, BVA7, BVA8, BVA9, ST1, ST2, ST3, ST4, ST5, ST6, ST10, ST12, ST14
11	A large number of spaces between street name and street type results in an error message of street not found	C/o Main Road Greensborough 3088.	*Following error reported to error log:* Error... Street not found.	EP1, EP2, EP11, BVA1, BVA3, BVA9 ST1, ST2, ST3, ST4, ST5, ST6, ST10, ST12, ST14
12	Inserting extra spaces between suburb and postcode results in error message of spaces required after suburb	UNIT 100 / 100 Main Road Greensborough 3088.	*Following errors reported to error log:* Error... Space required after suburb.	EP3, EP12, ST2, ST3, ST5, ST6, ST11, ST13, ST14
13	Program is not case sensitive - it does not report errors for lowercase at start of street name, street type or suburb, or lowercase unit, flat, rsd, c/o or c/	100 main Road Eltham 3095. 100 Main rd Eltham 3095.	*Following address appears in Addresses-Correct.txt:* 100 Main Rd Eltham 3095. 100 Main Rd Eltham 3095.	EP3, EP12, EP13, ST3, ST4, ST7, ST8, ST10, ST12
14	Address strings of 99 characters or more are not processed by the program and prevent all other addresses from being processed.	100 aaaaaaaaaabbbbbbbbbbccccccccccddddddddddeeeeeeeeeeehhhhhhhhhhiiiiiiiiiijj St Greensborough 3088.	All output files are empty, including the error log	EP2, BVA4, BVA5, BVA6, BVA8, ST13, ST14
15	Street types are abbreviated in the correct address output file, regardless of which form they were input in	100 Main Road Eltham 3095. 100 Main Rd Eltham 3095.	*Following address appears in Addresses-Correct.txt:* 100 Main Rd Eltham 3095. 100 Main Rd Eltham 3095.	EP12, ST12
16	Replacing the unit number with a symbol results in extra error messages of incorrect flat/unit symbol and incorrect spaces after house or unit number	UNIT [/ 200 Main Road Greensborough 3088.	*Following errors reported to error log:* Error... Number has too few digits. Must have 3. Error... Incorrect flat/unit symbol. Error... Number has too few digits. Must have 3. Error... Incorrect spaces after house or unit number.	EP9, BVA1
17	Replacing space after UNIT or C/o identifier, removing space after unit number or C/o, inserting extra spaces between the UNIT identifier and the unit number, or adding an invalid character after the UNIT tag, all result in addresses being accepted as correct	UNITa100 / 200 Main Road Greensborough 3088. UNIT 100/ 200 Main Road Greensborough 3088. C/oMain Road Greensborough 3088. UNIT 100 / 200 Main Road Greensborough 3088. UNIT1 100 / 200 Main Road Greensborough 3088.	*Following addresses appear in Addresses-Correct.txt:* UNITa100 / 200 Main Road Greensborough 3088. UNIT 100/ 200 Main Road Greensborough 3088. C/o Ain Road Greensborough 3088. UNIT 100 / 200 Main Road Greensborough 3088. UNIT 100 / 200 Main Road Greensborough 3088.	EP1, EP6, EP7, EP9, EP11, EP12, BVA1, BVA3, BVA6, BVA9, ST1, ST2 , ST3, ST4, ST5, ST6

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
Continued from previous page…				
18	Invalid symbol in place of postcode results in error message of missing full stop being reported	100 Main Road Eltham %.	*Following error reported to error log:* Error... There was no full stop at the end of the address.	EP1, EP4, EP5, EP6, EP7, EP8, EP9, EP10, EP11, BVA9, ST2, ST10
19	Missing space after UNIT/FLAT/RSD tag results in error message of number having too few digits	UNIT100 / 200 Main Road Greensborough 3088.	*Following error reported to error log:* Error... Number has too few digits. Must have 3.	EP1, EP5, EP9, EP11, BVA1, BVA4, BVA9, ST1, ST4
20	Replacing house number with a Real results in error messages incorrectly reporting there were incorrect spaces after the house number, space required after suburb, street not being found and invalid suburb being entered	100.11 Main Road Eltham 3095.	*Following errors reported to error log:* Error... Incorrect spaces after house or unit number. Error... Space required after suburb. Error... Street not found. Error... Invalid suburb entered.	EP2, EP5
21	Missing or invalid unit or house number results in error message of incorrect spaces after house number	a Main Road Eltham 3095. UNIT 100 / a Main Road Greensborough 3088.	*Following errors reported to error log:* Error... Number has too few digits. Must have 3. Error... Incorrect spaces after house or unit number.	EP6, EP7, ST2, ST1, ST2, ST3, ST4, ST5, ST6, ST11, ST14
22	Invalid characters in house number correctly results in error message of invalid house number, but also that spaces after house number and suburb are incorrect, street was not found, invalid suburb and missing full stop	a12C Main Road Greensborough 3088. [Main Road Greensborough 3088. RSD A99 Main Grove Yan Yean 3755.	*Following errors reported to error log:* Error... Number has too few digits. Must have 3. Error... Incorrect spaces after house or unit number. Error... Space required after suburb. Error... Street not found. Error... There was no full stop at the end of the address. Error... Invalid suburb entered.	EP3, EP4, EP5, EP6, EP7, EP8, EP9, EP10, EP12, BVA1, BVA3, BVA6, ST1, ST2, ST3, ST5, ST6, ST11
23	Missing house number results in two messages reporting that there were illegal spaces before the address	Main Road Greensborough 3088.	*Following errors reported to error log:* Error... Illegal spaces before address. Error... Illegal spaces before address. Error... Number has too few digits. Must have 3. Error... Incorrect spaces after house or unit number.	EP11, BVA9
24	Inserting an integer between the street type and suburb results in error message of missing full stop and invalid suburb	100 Main Road 5555 Greensborough 3088.	*Following errors reported to error log:* Error... There was no full stop at the end of the address. Error... Invalid suburb entered.	EP4, EP9, EP10, ST5, ST6
25	Removing the UNIT/FLAT/RSD identifier results in error message of illegal spaces before address	100 / 200 Main Road Greensborough 3088.	*Following errors reported to error log:* Error... Illegal spaces before address. Error... Illegal spaces before address.	EP1, EP11, BVA9
26	Program does not detect data after the full stop field.	100 Main Street Eltham 3095.VIC	*Following address appears in Addresses-Correct.txt:* 100 Main Street Eltham 3095.	ST3, ST10, ST11
27	Street type Place is not accepted as correct and error message also reports invalid suburb.	100 Main Place Eltham 3095.	Error... Street not found. Error... Invalid suburb entered.	ST12

Appendix E. Known Failures in the Batch Processor

The table below lists all known failures detectable in the Batch Processor program, which was used in the industrial experiment. This includes a description of the failure, an example input that is capable of detecting a failure, and a mapping to the Atomic Rules that can be used to detecting the failure.

Table 8-30: Known defects in the Batch Processor program.

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
1	Averages across all batches are rounded, resulting in invalid results (averages were stored as integers, not Reals/doubles)	sbatch 11AAA A11A 30, AA11A 31 ebatch 11AAA lbatch	Average computed as 60	EP2, EP3, EP12, BVA2, BVA3, BVA4, BVA5, BVA6, BVA7, BVA8
2	Did not diagnose non-matching sbatch and ebatch id's	sbatch 11AAA A11A 30 ebatch 11BBB lbatch	No error recorded in error log	EP1, EP11, BVA2 BVA4, BVA5, BVA6, BVA7, BVA8, BVA9, ST12
3	Missing character from lbatch tag correctly diagnosed, but with extra error message of invalid batchid being reported	sbatch 11AAA A11A 50 ebatch 11AAA lbatc	Line # 4: BatchIDCheck FAILED(format should be <d><d><l><l><l>) Line # 5: incorrect end of file, no lbatch detected	ST1, ST4
4	Did not diagnose data after sbatch, ebatch, lbatch or after a recordline	sbatch 11AAA 11AAA A11A 30 30 ebatch 11AAA 11AAA lbatch abcde	No error recorded in error log	ST11
5	Did not diagnose missing sbatch id	sbatch A11A 30 ebatch 11AAA lbatch	No error recorded in error log	EP11, BVA9
6	Did not diagnose missing or additional partid	sbatch 11AAA 30 ebatch 11AAA lbatch sbatch 11AAA A11A A11A 50 ebatch 11AAA lbatch	No error recorded in error log	EP11, BVA9, ST11
7	Did not diagnose missing ebatch tag	sbatch 11AAA A11A 30 11AAA lbatch	No error recorded in error log	EP1, EP11, BVA9
8	Did not diagnose missing ebatch id	sbatch 11AAA A11A 30 ebatch lbatch	No error recorded in error log	EP11, BVA9
9	Did not diagnose spaces before sbatch	sbatch 11AAA A11A 30 ebatch 11AAA lbatch	No error recorded in error log	ST11
10	Did not diagnose spaces before ebatch	sbatch 11AAA A11A 30 ebatch 11AAA lbatch	No error recorded in error log	ST11
11	Did not diagnose spaces before lbatch	sbatch 11AAA A11A 30 ebatch 11AAA lbatch	No error recorded in error log	ST11

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
Continued from previous page…				
12	Did not diagnose extra spaces between partid and value	sbatch 11AAA A11A 30 ebatch 11AAA lbatch	No error recorded in error log	EP1, EP2, EP11, BVA3, BVA6, BVA9
13	Missing input file is not reported as an error to the error log	No input.txt file exists in folder	No error recorded in error log	EP11, BVA9
14	Program crashes with a "divide by zero" error when the input file does not contain at least one batch with one valid recordline. This includes a batch with a missing or uppercase sbatch tag, missing or invalid values (e.g. alpha instead of integer or values < -99 or > 98), missing spaces between sbatch tag and id, missing spaces or invalid characters between partid and value, and on empty file	11AAA A11A 30 ebatch 11AAA lbatch SBATCH 11AAA A11A 30 ebatch 11AAA lbatch sbatch 11AAA A11A=30 ebatch 11AAA lbatch	Program crashed	EP1, EP2, EP4, EP5, EP6, EP7, EP8, EP9, EP10, EP11, BVA1, BVA2, BVA3, BVA4, BVA6, BVA7, BVA9, ST1, ST2, ST3, ST4, ST5, ST6, ST7, ST10, ST11, ST12, ST13, ST14
15	Misdiagnoses missing letters from partid and missing letters in sbatch and ebatch id's	sbatch 11 A11A 30 ebatch 11AAA lbatch sbatch 11AAA A11A 30 ebatch 11 lbatch sbatch 11AAA 11A 30 ebatch 11 lbatch	Line # 2: Incorrect Data Type for Value Entered, please ensure it is an integer	EP1, EP11, BVA9, ST1, ST4, ST5, ST10
16	Did not diagnose missing space or extra space after comma on a recordline	sbatch 11AAA A11A 30,B22B 25 ebatch 11AAA lbatch	No error recorded in error log	BVA1, BVA4, BVA9, ST1, ST4
17	Missing digit in sbatch id or value of -100 (when at least one recordline is valid) is reported as an invalid partid	sbatch 1AAA A11A 50 B11B -100 ebatch 11AAA lbatch	Line # 1: PartIDCheck FAILED(format should be <l><d><d><l>)	ST1, ST2, ST3, ST6, ST10, ST11
18	Misdiagnoses invalid partid as an invalid sbatch or ebatch identifier	sbatch 11AAA 1111A 50 ebatch 11AAA lbatch sbatch 11AAA ?A11A 50 ebatch 11AAA lbatch	Line # 2: BatchIDCheck FAILED(format should be <d><d><l><l><l>)	ST1, ST2, ST3, ST4, ST5, ST6, ST10
19	Did not diagnose invalid sbatch or ebatch id, and output calculations are incorrect	sbatch 11AAA A11A 30 ebatch 11 lbatch sbatch 11 A11A 30 ebatch 11AAA lbatch	No error recorded in error log and output calculations are incorrect	EP1, EP11, BVA9

#	Failure Description	Example Input	Example Output	Atomic Rules Capable of Detecting the Failure
Continued from previous page…				
20	Did not diagnose missing comma between two parts	sbatch 11AAA A11A 30 B22B 25 ebatch 11AAA lbatch	No error recorded in error log	EP1, EP11, BVA9, ST1, ST4
21	Did not diagnose extra comma between two parts	sbatch 11AAA A11A 30,, B22B 25 ebatch 11AAA lbatch	No error recorded in error log	ST10
22	Program does not diagnose uppercase S in sbatch tag when there is at leats one other valid batch in the file	Sbatch 11AAA A11A 30, B22B 25 ebatch 11AAA lbatch	No error recorded in error log	ST7
23	Program does not diagnose partid and value in the reverse order	sbatch 11AAA 50 A11A ebatch 11AAA lbatch	No error recorded in error log	Incorrect use of ST14, could be detected using SBMT substitution rule
24	Does not diagnose missing value when there are more than two recordlines in a file	sbatch 11AAA A11A 50 B11B ebatch 11AAA lbatch	No error recorded in error log	EP11, BVA9

Appendix F. Functional Specification of the Atomic Rules Testing Tool

F.1 Overview

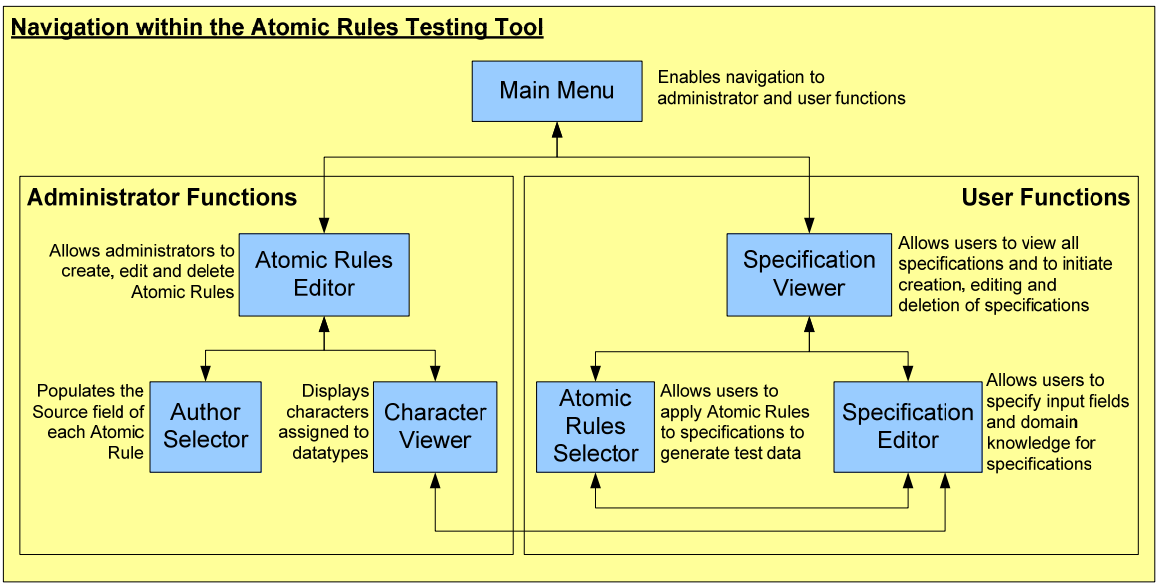
This section provides a detailed, low-level functional specification for the Atomic Rules Testing Tool. It starts by providing an overview of the 'screens' (i.e. GUI components) of ARTT (Section F.2) and illustrating the process of creating Atomic Rules, creating specifications and generating test data (Section F.3). This is followed by a detailed explanation of the functionality of each GUI screen (Section F.4), pseudo code that explains how ARTT generates test data (Section F.6) and a definition of the datatypes that are covered by ARTT (Section F.5).

F.2 High-Level Screen Design and Navigation

As outlined in Chapter 4, ARTT functionality is divided into two main areas: administrator and user functionality. The screens that are available in the system are as follows (Figure 8-7).

- **Main Menu**. This screen allows the user to navigate either to user functions or administrator functions (see Section F.4.1).
- **Atomic Rules Editor**. This screen allows administrators to create, edit and delete Atomic Rules (see Section F.4.2).
- **Author Selector**. This screen is accessed from the Atomic Rules editor. It allows administrators to select the sources (i.e. authors of textbooks, standards and papers on software testing) that have published existing Atomic Rules (see Section F.4.3).
- **Character Viewer**. This screen allows administrators to view the individual characters that are included in each datatype (e.g. Integer, Real, Alpha), which are used within the Atomic Rules Editor to specify the Original Datatype and Test Datatype of each Atomic Rule, and within the Specification Editor to specify the datatype of each input field under test (see Section F.4.4).
- **Specification Viewer**. This screen allows users to view all specifications that have been created, and to initiate creation, editing and deletion of specifications (see Section F.4.5).
- **Specification Editor**. This screen allows users to define the input fields of each specification, assign domain knowledge to the specification, attach files to the specification and view a BNF representation of the specification that is automatically generated by ARTT (see Section F.4.6).
- **Atomic Rules Selector**. This screen allows users to apply a chosen set of Atomic Rules from EP, BVA and ST to a specification to automatically generate black-box test data (see Section F.4.7).

Figure 8-7: Screens and navigation within the Atomic Rules Testing Tool (from Chapter 4).



F.3 Activity Diagrams

The two activity diagrams below illustrate how a user (Figure 8-8) and an administrator (Figure 8-9) can interact with ARTT[2]. Nodes in these diagrams that are prefixed with a 'U' represent actions that can be carried out by a user or an administrator, while notes prefixed with 'S' represent system actions. The two scenarios that are represented in Figure 8-8 and Figure 8-9 are as follows.

(a) The user starts the system (from U1).

(b) The user chooses to navigate to user area of functionality within the system (from U2.1).

   a. The user creates a new specification (from U3.1).

   b. The user edits an existing specification (from U3.2).

   c. The user deletes an existing specification (from U3.4).

   d. The user applies a set of Atomic Rules to a specification to generate black-box test data (from U3.3).

(c) The administrator chooses to navigate to administrator functionality (from U2.2).

   a. The administrator creates a new Atomic Rule (from U9.1).

   b. The administrator edits an existing Atomic Rule (from U9.2).

   c. The administrator deletes an existing Atomic Rule (from U9.2).

   d. The administrator views the character set covered by a particular datatype (from U9.4).

[2] Flow of event diagrams are useful diagramming can be used in use case testing (CSTP Module 2 2007).

Figure 8-8: Activity diagram depicting a user interacting with ARTT.

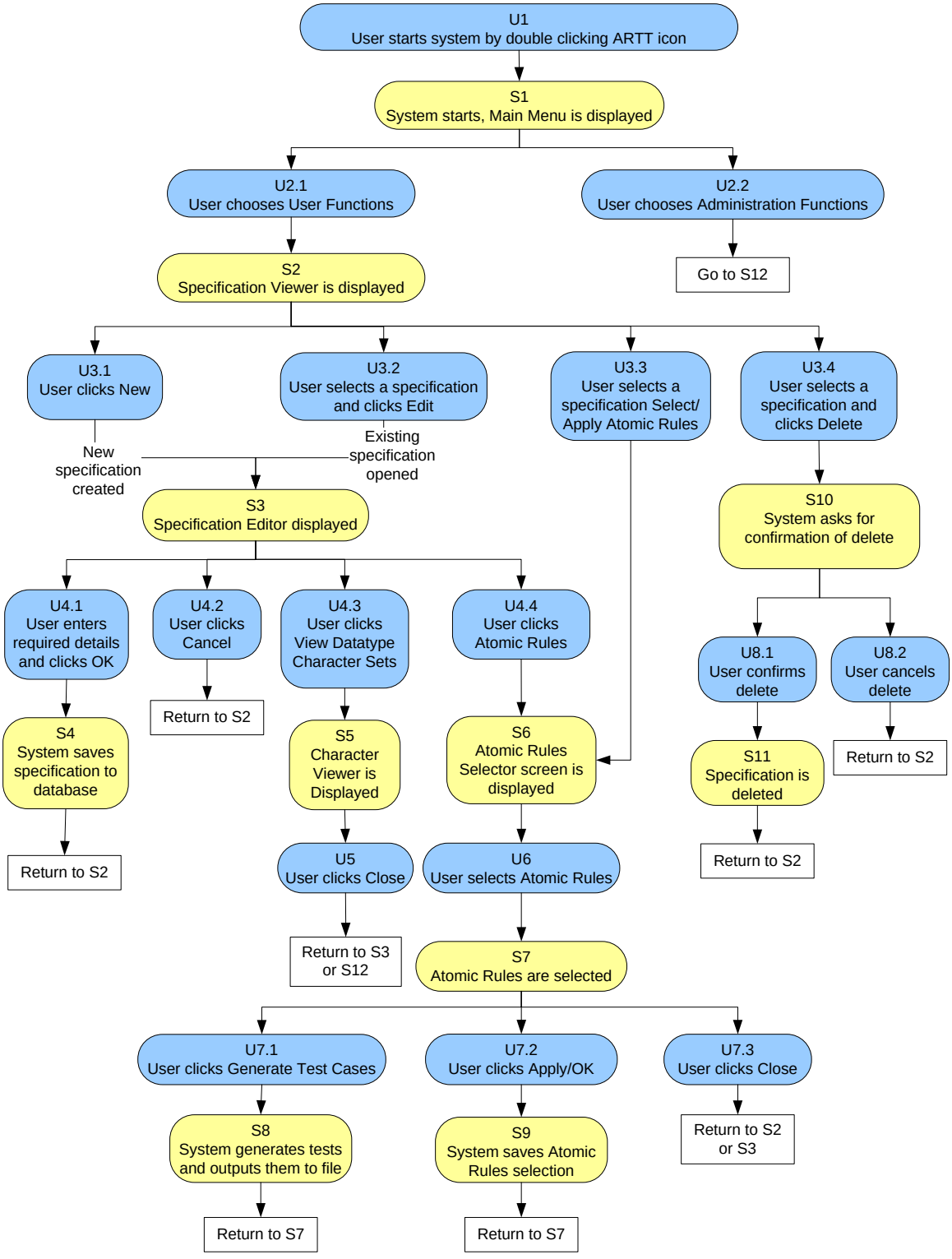
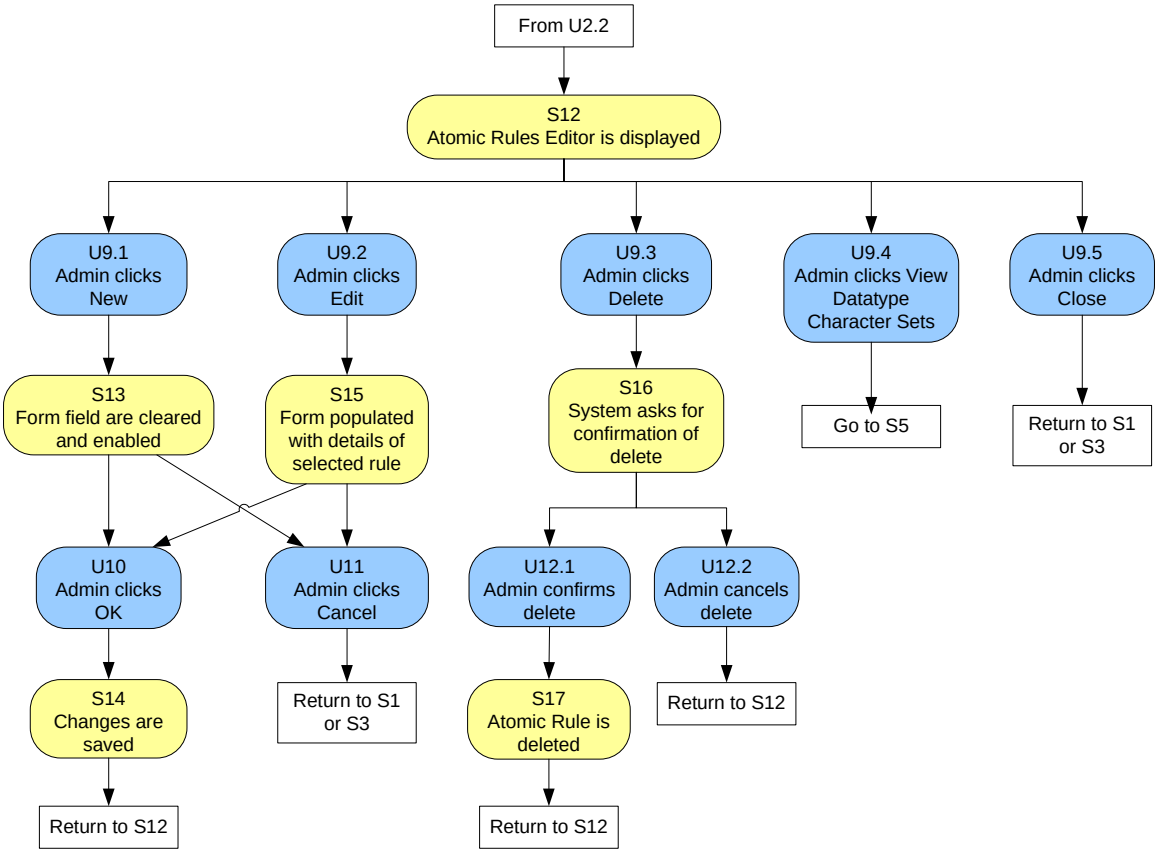


Figure 8-9: Activity diagram depicting an administrator ('Admin') interacting with ARTT.



F.4 Graphical User Interface Screens and their Associated Functionality

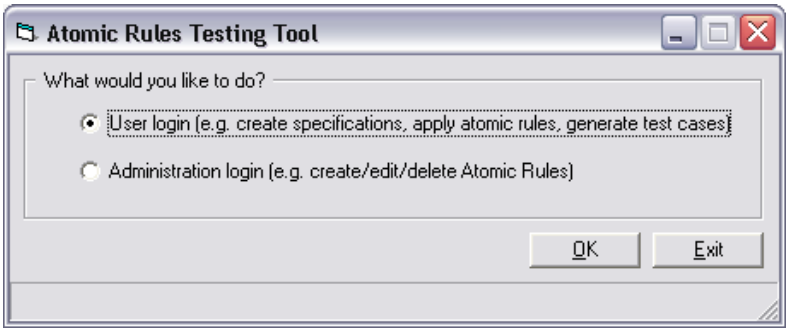
The functionality of each screen within ARTT is described in the following subsections (Section F.4.1 to F.4.7). Each is described using a tabular format, containing the following information:

- **Purpose**: a brief statement that explains what the screen aims to achieve;
- **Screen Capture**: a screen capture that illustrates the screen's GUI; and
- **Fields**: a detailed description of each field within each screen, including the field's type. As some fields are more complex than others, some are described in more detail.

F.4.1 The Main Menu

The Main Menu (Table 8-31) allows *users* and *administrators* to navigate to the *user* functions area of the tool, which enable the creation of specifications and the generation of test cases, or to the *administration* functions area, which enable the creation, editing or deletion of Atomic Rules.

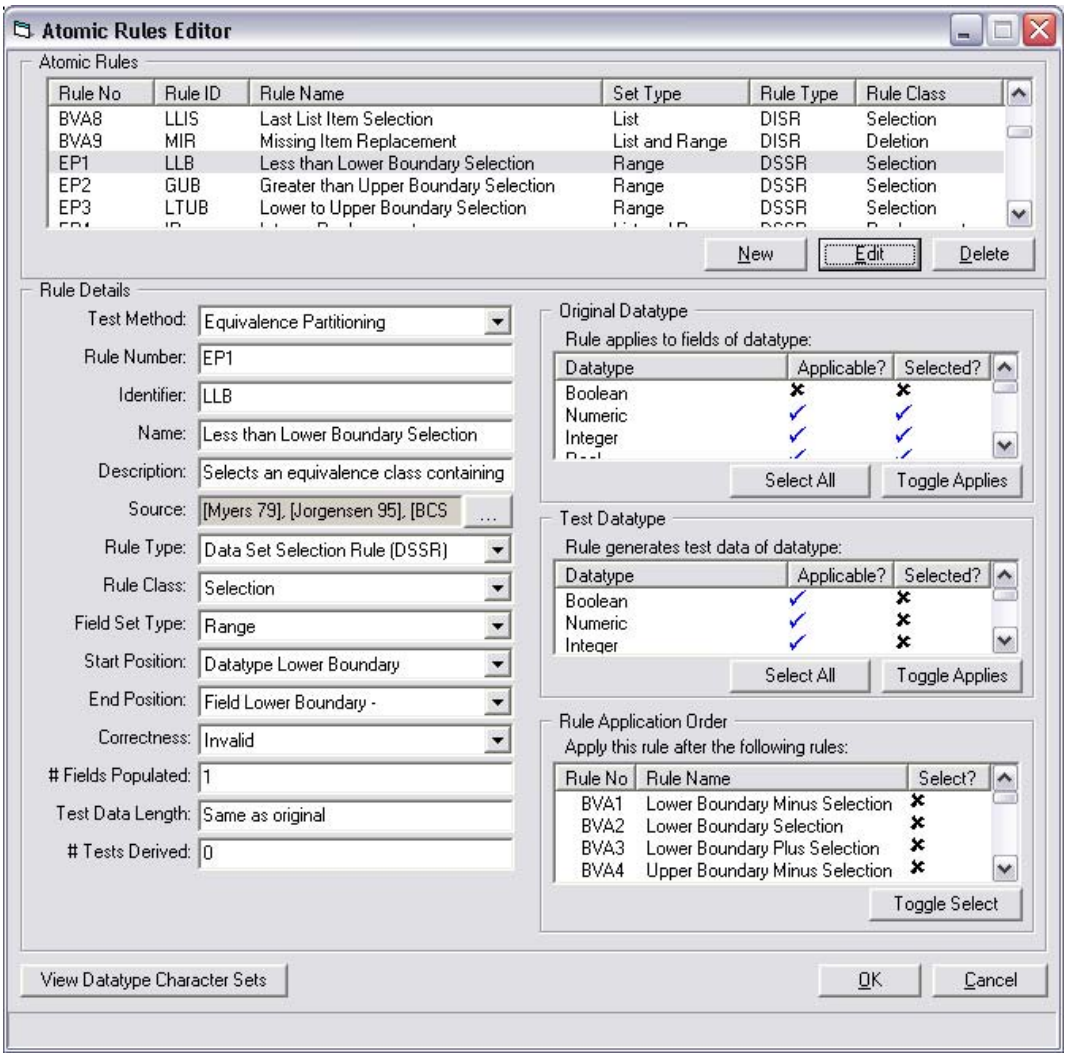
Table 8-31: The Main Menu.

Main Menu		
Purpose: this is the first screen that is displayed when the system starts. It allows the user to navigate to: • the *user* screens of the system, such as to create specifications or to generate test cases, or • to the *administration* screens of the system, such as to create, edit or delete Atomic Rules.		
User Interface		
		
Field	**Field Type**	**Functionality**
Administration Login	Radio Button	Selecting this option and clicking OK opens the *Atomic Rules Editor* (Table 8-32).
User Login	Radio Button	Selecting this option and clicking OK opens the *Specification Viewer* (Table 8-36).
OK	Button	Opens the *Atomic Rules Editor* or *Specification Viewer*, depending on which option is selected.
Exit	Button	Closes the *Main Menu* and exits the system.

F.4.2 The Atomic Rules Editor

The Atomic Rules Editor (Table 8-32) allows administrators to create, edit and delete Atomic Rules.

Table 8-32: The Atomic Rules Editor.

Atomic Rules Editor
Purpose: This screen is accessed from the Main Menu (Table 8-31). It allows users to view, create, edit and delete Atomic Rules. Most fields on this screen correspond directly with attributes of the Atomic Rules schema (see Chapter 3, Section 3.2.2). On this screen a user can: • View a rule by single left clicking one in the Atomic Rules list at the top of the screen • Create a new rule by clicking *New*, completing all form fields and clicking *OK* • Edit a rule by selecting one clicking *Edit*, making the changes and clicking *OK* • Delete a rule by selecting one and clicking *Delete*
User Interface


Field	Field Type	Functionality
Atomic Rules	Record List	Lists all Atomic Rules defined in the system. Clicking on a rule populates all fields with data for that rule.
New	Button	Initiates creation of a new Atomic Rule by clearing all form fields.
Edit	Button	Initiates editing of the currently selected Atomic Rule.

Field	Field Type	Functionality
Continued from previous page…		
Delete	Button	Initiates deletion of the currently selected rule.
Test Method	Combo Box	Lists black-box methods Atomic Rules are currently defined for. Options are: • Equivalence Partitioning • Boundary Value Analysis • Syntax Testing
Rule Number	Text Box	Corresponds to the *Number* attribute of the Atomic Rules schema.
Identifier	Text Box	Corresponds to the *Identifier* attribute of the Atomic Rules schema.
Name	Text Box	Corresponds to the *Name* attribute of the Atomic Rules schema.
Description	Text Box	Corresponds to the *Description* attribute of the Atomic Rules schema.
Source (…)	Text Box & Button	Corresponds to the *References* attribute of the Atomic Rules schema. Clicking the "**…**" button (right of the field) opens the *Author Selector* (Table 8-34), enabling the user to identify the authors that have published the current rule.
Rule Type	Combo Box	Corresponds to the *Rule Type* attribute of the Atomic Rules schema. Options are: • Data Set Selection Rule (DSSR) • Data Item Selection Rule (DISR) • Data Item Manipulation Rule (DIMR) • Test Case Construction Rule (TCCR)
Rule Class	Combo Box	Describes the overall functionality of the current rule. Options are: • Selection – rule *selects* data (e.g. EP1) • Insertion – rule *inserts* values into a partition or test data value (e.g. ST3) • Deletion – rule *removes* values from a partition or test data value (e.g. ST1) • Replacement – rule replaces a field with an invalid partition (e.g. EP4) • Combinatorial – rule constructs test cases (e.g. EP16)
Field Set Type	Combo Box	Corresponds to *Set Type* attribute of the Atomic Rules schema. Options are: • List • Range • List and Range • Neither (e.g. non-terminal fields do not have a set type)
Start Position & End Position	Combo Box	• Describe the first and last positions from which a rule selects test data. Six classes of start and end positions can appear, as follows (see Table 8-33): • Datatype start and end positions relate to the boundaries of the datatype of the field under test, resulting in the selection an equivalence class. For example, a DSSR could use these to selected a partition from the lower to upper boundaries of the integer datatype [-32768 – 32767] • *Field* start and end positions can be used by a DSSR to select a partition of value from a range-based field. For example, they could be used to select values between the lower to upper boundary of an *age* field, selecting the partition *[0 - 150]*. They can also be used by a DISR to select one test data value from a partition, such as selecting the value *150* from the partition *[0 - 150]*. • *First* and *Last Field Value* can be used by a DSSR to select a partition of values from a list field. For example, they could be applied to a *colour* field to select the partition *[Red | Blue | Green]*. They can also be used by a DISR to select one test data value from a list, such as selecting *Red* from this partition. • *Nominal* can be used by a DSSR, DISR or DIMR to select the mid-point value of a field, partition or test data value. • *Random* can be used by a DSSR, DISR or DIMR to select a randomly chosen value from a field, partition or test data value. • *First* and *Last Character* can be used by a DIRM to alter a test data value, such as selecting "R" from the colour "Red" or selecting "reen" from "Green." Therefore, the values that appear in the Start and End Position fields depend on

		the value of *Rule Type*, as follows. If *Rule Type* = *DSSR* then rule selects a partition. Start and End Positions are: • Datatype Lower Boundary – (e.g. ASCII A – 1 = @) • Datatype Lower Boundary (e.g. ASCII A = A) • Datatype Lower Boundary + (e.g. ASCII A + 1 = B) • Datatype Upper Boundary – (e.g. ASCII Z – 1 = Y) • Datatype Upper Boundary (e.g. ASCII Z = Z) • Datatype Upper Boundary + (e.g. ASCII Z + 1 = [ ) • Field Lower Boundary – (just below lower boundary of a range) • Field Lower Boundary (on the lower boundary of a range) • Field Lower Boundary + (just above the lower boundary of a range) • Field Upper Boundary – (just below the upper boundary of a range) • Field Upper Boundary (on the upper boundary of a range) • Field Upper Boundary + (just above the upper boundary of a range) • First Field Value (first value in a list) • Second Field Value (second value in a list) • Second Last Field Value (second-last value in a list) • Last Field Value (last value in a list) • Nominal Value (middle value of a range or list) • Random Value (random value from a range or list) If *Rule Type* = *DISR* then rule selects a test data value, so End Position will be disabled. Start Positions are: • Field Lower Boundary – • Field Lower Boundary • Field Lower Boundary + • Field Upper Boundary – • Field Upper Boundary • Field Upper Boundary + • First Field Value • Last Field Value • Nominal Value • Random Value • Not Applicable If *Rule Type* = *DIMR* then the rule mutates test data values. Start and End Positions are: • Nominal Value • Random Value • First Character – – (e.g. add two chars to start of a data value) • First Character – (e.g. add one char to start of a data value) • First Character (e.g. mutate first character of a data value) • First Character + (e.g. mutate second character of a data value) • First Character ++ (e.g. mutate third character of a data value) • Last Character – – (e.g. mutate third last character of a data value) • Last Character – (e.g. mutate second last character of a data value) • Last Character (e.g. mutate last character of a data value) • Last Character + (e.g. add one char to end of a data value) • Last Character ++ (e.g. add two chars to end of a data value) • Not Applicable (does not select a particular character or value) If *Rule Type* = *TCCR* then these fields will be empty, because TCCRs do not select test data, they create test cases, so this field is disabled.
Correctness	Combo Box	Corresponds to the *Valid or Invalid* attribute of the Atomic Rules schema. Options are: Valid, Invalid, Valid or Invalid, Depends on field.

Field	Field Type	Functionality
Continued from previous page…		
# Fields Populated	Text Box	Corresponds to the # *Fields Populated* attribute of the Atomic Rules schema.
Test Data Length	Text Box	Corresponds to the *Test Data Length* attribute of the Atomic Rules schema.
# Tests Derived	Text Box	Corresponds to the # *Tests Derived* attribute of the Atomic Rules schema.
Original Datatype & Test Datatype	Record List	Defines the *Original Datatype* of fields that rules can be applied to, and the *Test Datatype* selected by the rule, which correspond to attributes of the same name in the Atomic Rules schema. For example, *BVA1: lower boundary - 1* and *BVA6: upper boundary + 1* can only be applied to range-based datatypes such as *Integer* (see Table 8-35); they cannot be applied to list-based datatypes like *Alphanumeric* as there is no way to choose an outside boundary value. The datatypes defined in ARTT extend the base set defined for EP, BVA and ST (see Chapter 3, Section 3.2.2). The datatypes defined are: • Integer (all integers from -32768 to 32767) • Integer+ (all positive integers from 0 to 32767) • Integer- (all negative integers from -32768 to -1) • Boolean (i.e. 1 and 0) • Numeric (i.e. ASCII 48 to ASCII 57) • Real (all Reals from -32768.00 to 32767.00) • Real+ (all positive Reals from 0.00 to 32767.00) • Real- (all negative Reals from -32768.00 to -1.00) • Alpha (all alphabetical characters from A-Z and a-z) • Lowercase Alpha (all lowercase alphas from ASCII 97 to ASCII 122) • Uppercase Alpha (all uppercase alphas from ASCII 65 to ASCII 90) • Alphanumeric (Alpha ∪ Numeric) • Control Character (all control characters from ASCII 1 to ASCII 31) • Symbol (Set 1) (all special characters from ASCII 32 to ASCII 47) • Symbol (Set 2) (all special characters from ASCII 58 to ASCII 64) • Symbol (Set 3) (all special characters from ASCII 91 to ASCII 96) • Symbol (Set 4) (all special characters from ASCII 123 to ASCII 127) • Symbol (Symbol 1 ∪ Symbol 2 ∪ Symbol 3 ∪ Symbol 4) • Null (empty) (ASCII 0) • Non-Alphanumeric (Symbol 1 ∪ Symbol 2 ∪ Symbol 3 ∪ Symbol 4) • ASCII (all characters in the ASCII table) • Same as original (applies to Test Datatype only)
Rule Application Order	Record List	Specifies the order in which rules can be applied, based on the four-step test selection process (see Chapter 3, Section 3.2.1). For example, a DIMR can only be applied *after* a DISR, while a DISR can only be applied *after* a DSSR.
View Datatype Character Sets	Button	Opens the *Character Viewer* (Table 8-35).
OK	Button	Saves all changes that have been made.
Cancel/Close	Button	Closes the screen and returns the user to the previous screen. If this is clicked during edit or new, it will cancel the action before closing the screen.

Table 8-33: Definition of the Rule Types and Rule Classes that dictate the start and end position sets that are possible for each Atomic Rule.

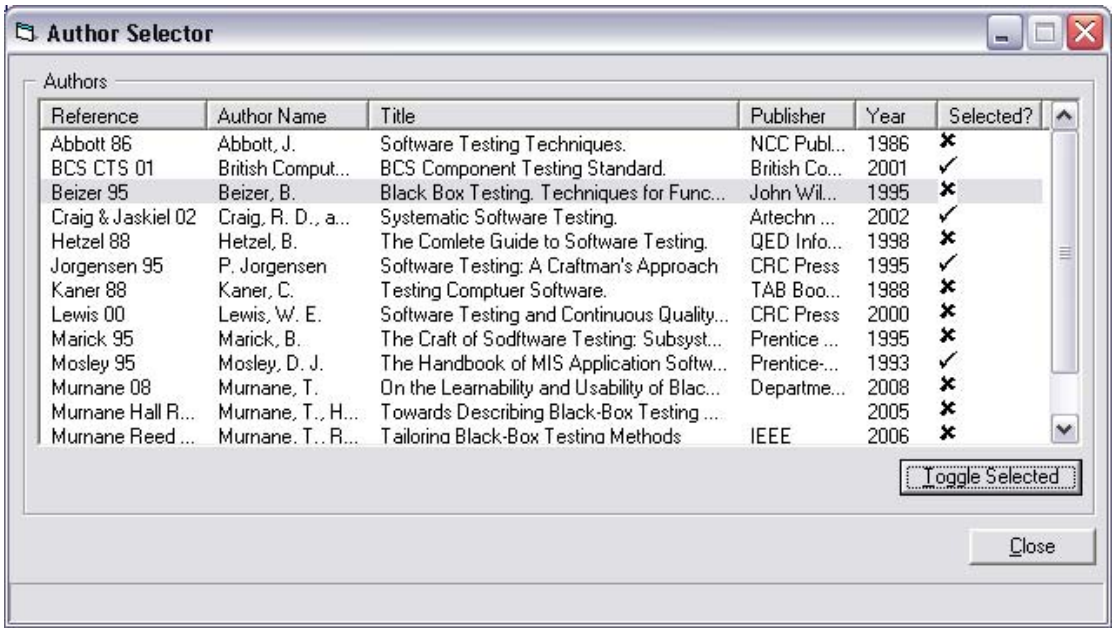
Rule Type	Rule Class	Type & Class Combination Possible?	Start / End Positions	Example
Data Set Selection Rule (DSSR)	Selection	Yes	List-based fields: • first value • last value • random value • nominal value	Selects a partition from a list. E.g. for field *<colour> ::= [Red | Green | Blue]* this could select a valid partition from the first to last value, by applying *EP12: valid list selection*, which would select *[Red | Green | Blue]*.
			Range-based fields: • min- • min • min+ • max- • max • max+ • nominal • random	Selects a partition of test data from a range of values. For the valid field *<age> ::= [0 – 150]* this could select a valid partition from min to max, such as applying *EP3: lower to upper boundary selection*, selecting the partition *[0 – 150]*.
	Insertion	No – rule class does not apply to DSSRs. While it would be possible to insert invalid values into a valid partition, this is already covered by EP "replacement" rules *EP4* to *EP10*.		
	Deletion	No – rule class does not apply to DSSRs. While it would be possible to delete data values from a valid partition, this is already covered by rules like *EP11: null item replacement*.		
	Replacement	Yes	List-based fields: • first value • last value	Replaces a field entirely with an invalid data partition, such as applying *EP4: integer replacement*, to entirely replace any field with the integer range [-32768 – 32767].
			Range-based fields: • min • max	
	Combinatorial	No – the combinatorial rule class only applies to TCCRs.		
Data Item Selection Rule (DISR)	Selection	Yes – All DISRs are selection rules. Most have the same start and end position, e.g. *BVA1: lower boundary selection*. Some select more than 1 test data item, such as *ST13: select all list alternatives*, which selects every item from a list as one test data item	List-based fields: • first field value • last field value • random value • nominal value	Selects one value from a partition, such as by applying *EP13: data value selector* to select a random value from field *<colour> ::= [Red | Green | Blue]* to select the colour Red.
			Range-based fields: • min- • min • min+ • max- • max • max+ • nominal • random	Selects one value from a partitio, such as by applying *EP13: data value selector* to selecting the nominal value from field *<age> ::= [0 – 150]* to select the number 75.
	Insertion	No – this rule class does not apply to DISRs		
	Deletion	No – this rule class does not apply to DISRs		
	Replacement	No – this rule class does not apply to DISRs		
	Combinatorial	No – the combinatorial rule class only applies to TCCRs.		

Rule Type	Rule Class	Type & Class Combination Possible?	Start / End Positions	Example
Continued from previous page…				
Data Item Manipulation Rule (DIMR)	Selection	Yes - selects characters between the specified start and end positions	Character position: • First Character – – • First Character – • First Character • First Character + • First Character ++ • Last Character – – • Last Character – • Last Character • Last Character + • Last Character ++ • Nominal Value • Random Value	Selects one or more characters from a data value, such as applying a rule to select the first character of the data item *Red*, resulting in the mutated test data value *R*. Atomic Rules that perform this type of function are not defined in existing literature. Thus, a new rule could be defined for this combination of Rule Type and Rule Class through the Atomic Rules Editor.
	Insertion	Yes - inserts randomly chosen characters between the specified start and end positions		Adds one or more characters to a data item, such as by applying *ST3: add extra character to end* to data value *Red* to select the mutated value *RedA*.
	Deletion	Yes - deletes characters between the specified start and end positions		Deletes one or more characters from a data item, such as by applying *ST1: remove last character* to data item *Red* to select the mutated value *Re*.
	Replacement	Yes - replaces characters between the start and end positions with randomly chosen characters		Replaces one or more characters from a data value, such as applying *ST2: replace last character* to the data item *Red*, resulting in the mutated test data value “1ed”.
	Combinatorial	No – the combinatorial rule class only applies to TCCRs.		
Test Case Construction Rule (TCCR)	Selection	No – does not apply to TCCRs.		
	Insertion	No – does not apply to TCCRs.		
	Deletion	No – does not apply to TCCRs.		
	Replacement	No – does not apply to TCCRs.		
	Combinatorial	Yes	There are no specific start or end positions for TCCRs.	NA

F.4.3 The Author Selector

The Author Selector screen (Table 8-34) allows *administrators* to assign particular authors to each Atomic Rule, allowing the reference source of existing Atomic Rule to be recorded.

Table 8-34: The Author Selector.

Author Selector		
Purpose: This screen is loaded when the user clicks the '…' button the Atomic Rules Editor (Table 8-32). It allows users to select authors that have published each Atomic Rule that is defined through the Atomic Rules Editor.		
User Interface		
		
Field	**Field Type**	**Functionality**
Authors	Record List	This lists all authors defined in the system.
Toggle Selection	Button	Selects (✓) or unselects (✗) the currently selected author.
Close	Button	Save the author selection, closes the screen and returns to the *Atomic Rules Editor*.

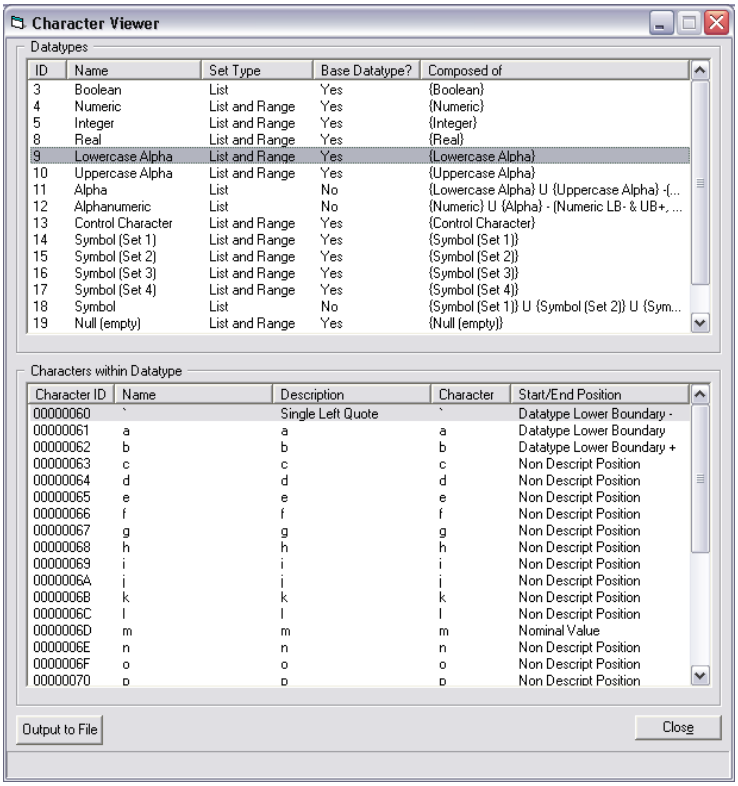
F.4.4 The Character Viewer

The Character Viewer (Table 8-32) allows administrators and users to view the contents of datatype sets that have been defined within the tool. The datatypes currently defined in ARTT were, with the exclusion of integers and Reals, chosen by dividing the ASCII table into sets of equivalent data, chosen to automate Atomic Rules representations of EP, BVA and ST.

For example, the tool differentiates between Lowercase Alphas and Uppercase Alphas, as this allows Syntax Testing rules *ST7* and *ST8* to be implemented, which swap the case of alphabetical fields to select invalid test data. In addition, ARTT divides the ASCII table into contiguous sets for integers, alphabetical characters and non-alphanumeric (i.e. special) characters, enabling outside boundary values to be selected for certain datatypes. For example, the At "@" symbol sits just below "A" in the ASCII table, allowing @ to be selected as an invalid outside boundary value for uppercase alpha fields.

Depending on the implementation language used to develop ARTT, there are other datatype sets that could have been implemented that are based on the "development environment domain" (Reed 1990) that could have been used to define datatype sets. Since ARTT is currently a prototype, future implementations may include definition of other datatype sets.

Table 8-35: The Character Viewer.

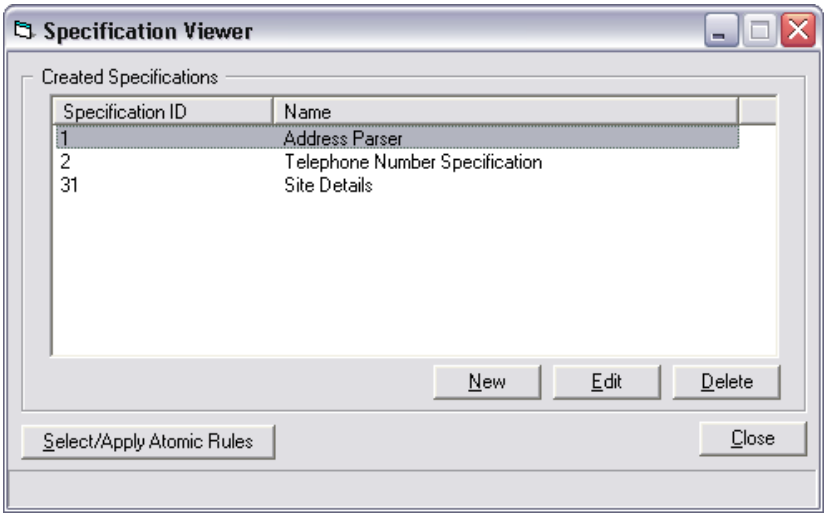
Character Viewer
Purpose: This screen can be accessed either from the Atomic Rules Editor (Table 8-32) or the Specification Editor (Table 8-37). It allows users to view all characters that are defined within a particular a datatype.
User Interface


Field	Field Type	Functionality
Continued from previous page…		
Datatypes	Record List	Displays all datatypes defined in the system. These extend the datatypes defined for EP, BVA and ST (see Chapter 3, Section 3.2.2) by decomposing the ASCII table further than that which is required for these methods (see Appendix F for complete list of characters and datatypes). Clicking a datatype populates the *Characters within Datatype* list with characters from that datatype. The columns of the table are: • *ID*: a unique identifier that is assigned to each datatype • *Name*: a unique name given to each datatype • *Set Type*: identifies whether the datatype is a range (e.g. integers or ASCII characters) or a list (e.g. the complete Symbol set) • *Base Datatype*: identifies whether the datatype is defined in its own right (e.g. integer) or is composed of other datatypes (e.g. Alpha, see below) • *Composed of*: identifies the other datatype each datatype is composed of (e.g. Alpha = Lowercase Alpha ∪ Uppercase Alpha)
Characters within Datatype	Record List	Lists characters within the selected datatype and shows characters outside datatype boundaries (e.g. ` is just below the lower boundary of Lowercase Alpha in ASCII). The columns of this table are: • *Character ID*: unique identifier for each character (ASCII characters are assigned their decimal identifier) • *Name*: character name (e.g. smallest integer named “Integer – Lower Boundary”) • *Description*: character description (e.g. smallest integer has the description “Integer – Smallest”) • *Character*: the actual character value • *Start/End Position*: used by Atomic Rules during test data generation (see *Start & End Position* in Table 8-32).
Output to File	Button	Outputs all datatypes and characters to a comma separated file (csv), allowing the user to view them all at the same time.
Close	Button	Closes the screen and returns the user to the previous screen.

F.4.5 The Specification Viewer

The Specification Viewer (Table 8-36) allows users to view all specifications that have been created previously in the tool and also to create new specifications and edit and delete existing specifications.

Table 8-36: The Specification Viewer.

Specification Viewer		
Purpose: This screen is accessed from the Main Menu (Table 8-31). It allows users to: • Create a new specification by clicking *New* (opens the *Specification Editor*) • View and edit existing specifications by selecting one and clicking *Edit* (opens the *Specification Editor*) • Delete a specification by selecting one and clicking *Delete* • Generate test data/cases by selecting a specification and clicking *Select/Apply Atomic Rules* (opens the *Atomic Rules Selector* screen)		
User Interface		
		
Field	**Field Type**	**Functionality**
Created Specifications	Record List	Lists all existing specifications that exist in the system.
New	Button	Opens the *Specification Editor* (Table 8-37) with all fields cleared, ready for the creation of a new specification.
Edit	Button	Opens the *Specification Editor* (Table 8-37) will all fields populated with values corresponding to the selected specification.
Delete	Button	Initiates deletion of the currently selected specification.
Select / Apply Atomic Rules	Button	Opens the *Atomic Rules Selector* (Table 8-43) for currently selected specification, allowing the user to generate test data and test cases.
Close	Button	Closes the screen and returns the user to the *Main Menu* (Table 8-31).

F.4.6 The Specification Editor

The Specification Editor consists of four main tabs: Fields, Domain Knowledge, Specification Files and Backus-Naur Form Specifications. The functionality of these tabs is as follows.

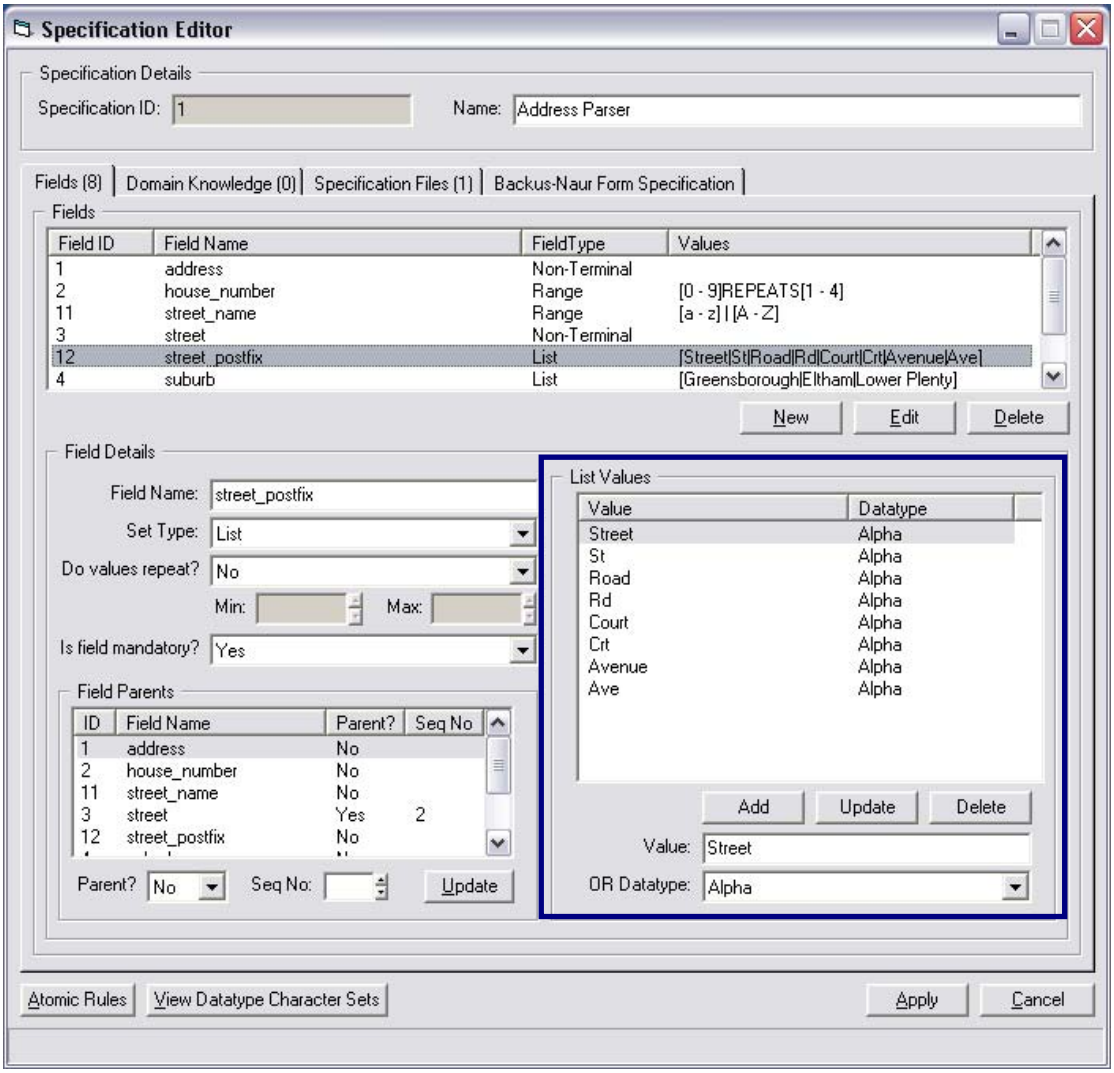
- **Fields:** this tab allows the user to define the characteristics of input fields, which will eventually be used for test data generation. Since there are two Field Types in the Atomic Rules schema (i.e. Lists and Ranges), this screen allows the creation of both list-based (Section F.4.6.1) and range-based (Section F.4.6.2) fields. The number shown to the right of the tab name (see Table 8-37) is a count of the number of input fields that have been created for the current specification.
- **Domain Knowledge:** this tab allows the user record domain knowledge against the current specification that is captured through the use of GQASV. The number shown to the right of the tab name (see Table 8-37) is a count of the number of domain knowledge records that have been defined for the specific input field that is currently being selected for the current specification.
- **Specification Files**: this screen allows users to attach program specification documents to the current specification. The number shown to the right of the tab name (see Table 8-37) is a count of the number of specification files that have been linked to the current specification.
- **Backus-Naur Form Specification:** this tab allows users to view the BNF representation of input fields in the current specification. The BNF is automatically produced by ARTT.

Each of these tabs is explained in detail in the subsections below.

F.4.6.1 The Specification Editor "Fields" Tab for List-Based Input Fields

The GUI and functionality of the Fields tab of the Specification Editor, when list-based fields are being created or edited, is explained below (Table 8-37).

Table 8-37: The Specification Editor 'Fields' tab for a list-based field.

Specification Editor "Fields" tab (List Values frame displayed)
Purpose: This screen is accessed from the Specification Viewer (Table 8-36). This screen has four tabs, as follows: • Fields – allows users to view, create, edit and delete fields for a specification • Domain Knowledge – allows users to store domain knowledge for fields on the Fields tab (Table 8-39) • Specification Files – allows users to attach Rich Text Document versions of their specification (Table 8-41) • Backus-Naur Form Specification – generates a Backus-Naur Form representation of the specification, based on the fields defined on the Fields tab (Table 8-42) This screen capture shows the *List Values* frame for defining a list-based field (see Table 8-38 for an example of the *Range Values* frame for defining range-based fields). The tab name includes the number of fields defined in the current specification (e.g. the tab name is "*Fields (8)*" in the screen capture below).
User Interface


Field	Field Type	Functionality
Specification ID	Text Box	Unique number that is assigned by the system to identify each specification.
Name	Text Box	Name of the specification assigned by the user.
Fields (Record List)	Record List	Contains fields created against the current specification. Clicking on a field loads data related to that field on the *Fields* and *Domain Knowledge* tabs.
New	Button	Initiates the creation of a new field by clearing all fields on the form.
Edit	Button	Initiates editing of the currently selected field.
Delete	Button	Initiates deletion of the currently selected field.
Field Name	Text Box	Name of the field.
Set Type	Combo Box	The Set Type of the field. Options are: • List e.g. <street_type> ::= [St | Street | Road | Road…] • Range e.g. <age> ::= [0 – 150] • AND e.g. <name> ::= <first_name> **&** <surname> • OR e.g. <number> ::= [1 – 500] **|** [600 – 1000] • Non-terminal e.g. <*street*> *::=* <*street_name*> <*street_type*> If *Set Type* = *List*, the *List Values* frame is shown (see screen capture above). If *Set Type* = *Range*, the *Range Values* frame is displayed (see Table 8-38).
Do values repeat?	Combo Box	Identifies whether the values in the field repeat, e.g. <*house_number*> *::= [0-9]*$^{1-4}$ is composed of integers from 0 to 9 that repeat 1 to 4 times. Options are Yes and No. If *Do value Repeat? = Yes* then fields *Min* and *Max* are enabled.
Min	Text Box	Stores the minimum number of repetitions for fields that repeat.
Max	Text Box	Stores the maximum number of repetitions for fields that repeat.
Is field mandatory?	Combo Box	Records whether a field is a mandatory or optional. Options are Yes and No. If *Is field mandatory = No* then Atomic Rules that select Null (e.g. *EP11: null item replacement*) produce valid test data.
Field Parents	Record List	Lists fields that are parents of the current field, e.g. <*street*> *::=* <*street_name*> <*street_type*>, so <*street*> is a parent of <*street_name*> and <*street_type*>.
Parent?	Combo Box	• Identifies if a field is a parent of the current field. Options are: Yes and No
Seq No	Text Box	If several fields are parented by one field, each child field has a sequence number identifying where it sits in the field order, e.g. for the field <*street*> *::=* <*street_name*> <*street_type*>, <*street_name*> comes first with sequence number 1 and <*steet_type*> comes second with sequence number 2.
Update	Button	Saves the values of the *Parent?* and *Seq No* to the *Field Parents* list.
List Values	Record List	Displays all list values for the currently selected field.
Value	Text Box	Allows the user to add one list value to the current field.
OR Datatype	Text Box	Instead of choosing a *Value*, a user can add *Datatypes* to a list. Options are: • Numeric • Integer • Real • Lowercase Alpha • Uppercase Alpha • Control Characters • Symbol Set 1 • Symbol Set 2 • Symbol Set 3 • Symbol Set 4
Add	Button	Adds the *Value* and/or *Datatype* chosen by the user to the *List Values* list.

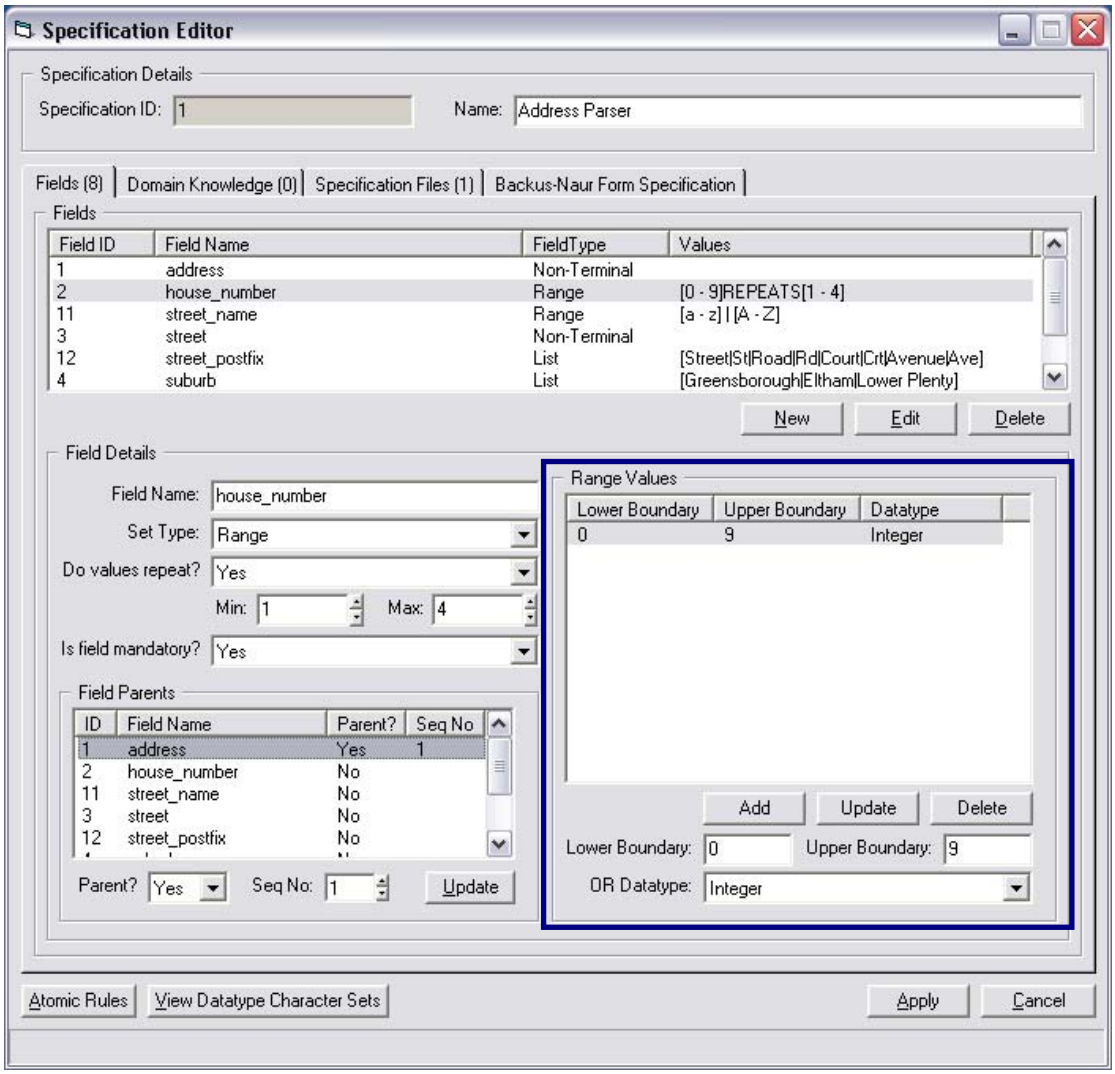
Field	Field Type	Functionality
Continued from previous page…		
Update	Button	Updates the value selected in the *List Values* list with the values defined in the *Value* and *Datatype* fields.
Delete	Button	Deletes the currently selected list value.
Atomic Rules	Button	Opens the *Atomic Rules Selector* (Table 8-43) to generate test data/cases.
View Datatype Character Sets	Button	Opens the *Character Viewer* (Table 8-35).
Apply	Button	Saves all changes to the database.
Cancel/Close	Button	Cancels all changes, closes the form and returns the user to the *Specification Viewer* (Table 8-36).

F.4.6.2 The Specification Editor "Fields" Tab for Range-Based Input Fields

The GUI and functionality of the Fields tab of the Specification Editor, when range-based fields are being created or edited, is explained below (Table 8-38).

Table 8-38: The Specification Editor 'Fields' tab for a range-based field.

Specification Editor "Fields" tab (Range Values frame displayed)
Purpose: This screen is accessed from the Specification Viewer (Table 8-36). The purpose of this tab is the same as the screen sown in Table 8-37. However, this screen capture shows the *Range Values* frame for defining range-based fields.
User Interface



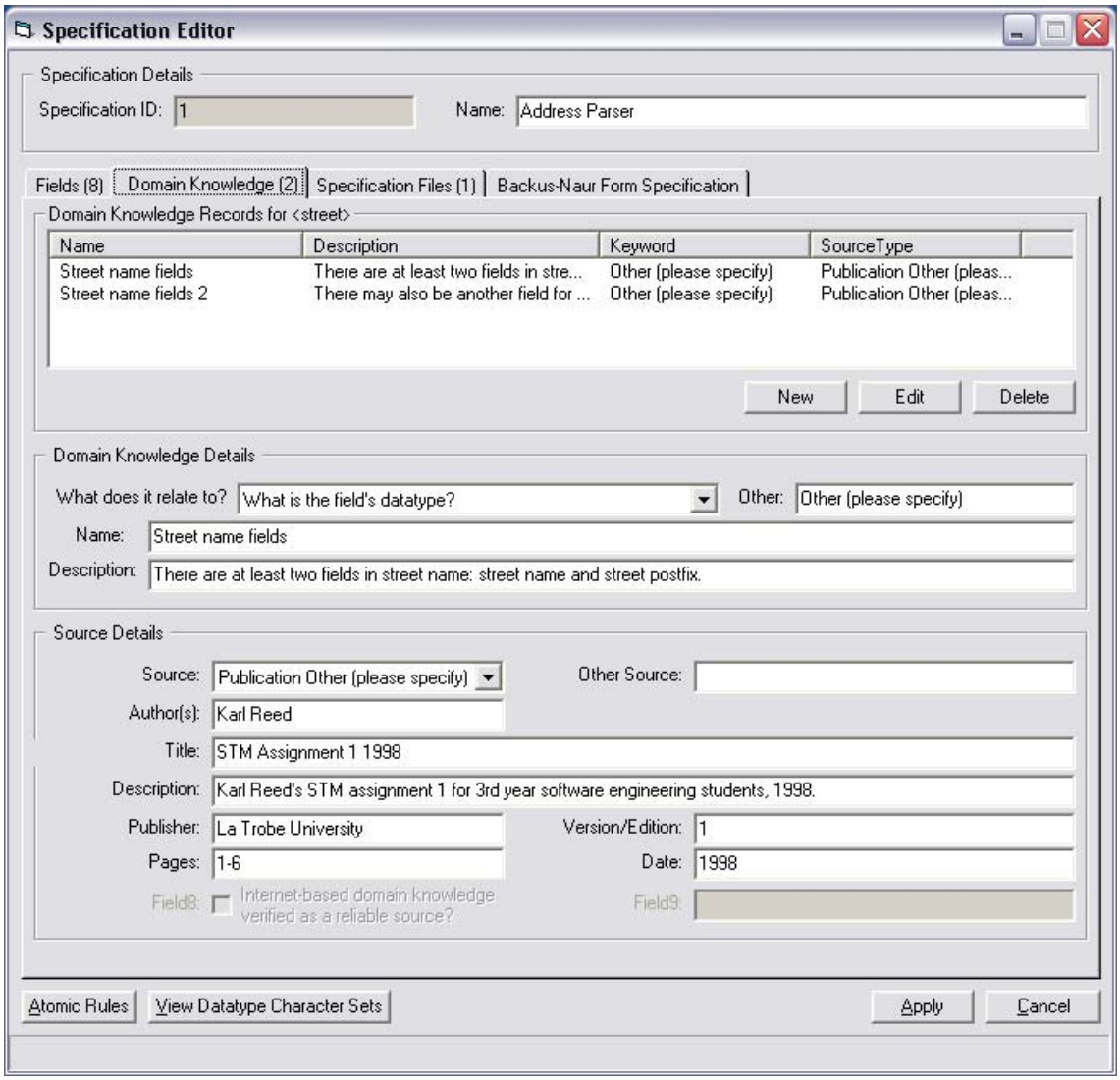
Field	Field Type	Functionality
Range Values	Record List	Displays boundary values for the currently selected field. When a record is selected in the *Range Values* list, *Lower Boundary*, *Upper Boundary* and *Or Datatype* are automatically populated.
Lower Boundary	Text Box	Allows the user to enter a lower boundary for the current field.
Upper Boundary	Text Box	Allows the user to enter an upper boundary for the current field.

Field	Field Type	Functionality
Continued from previous page…		
OR Datatype	Combo Box	If the user does not enter anything in the *Lower* and *Upper Boundary* fields they can add a *Datatype* to the list instead. Options are: • Numeric • Integer • Real • Lowercase Alpha • Uppercase Alpha • Control Characters • Symbol Set 1 • Symbol Set 2 • Symbol Set 3 • Symbol Set 4 These correspond to the "base" datatypes defined in Appendix F.
Add	Button	Adds the *Value* and/or *Datatype* chosen by the user to the *Range Values* list.
Update	Button	Updates the value selected in the *Range Values* list with the values defined in the *Lower/Upper Boundary* and *Datatype* fields.
Delete	Button	Deletes the currently selected range.

F.4.6.3 The Specification Editor "Domain Knowledge" Tab

The Domain Knowledge tab of the Specification Editor allows users to record domain knowledge captured through GQASV. The GUI and functionality of this tab is explained below (Table 8-39).

Table 8-39: The Specification Editor 'Domain Knowledge' tab.

Specification Editor "Domain Knowledge" tab
Purpose: This screen is accessed from the Specification Viewer (Table 8-36). The purpose of this tab is to allow users to record domain knowledge against each specification field. This allows users to store domain knowledge they collect through Goal/Question/Answer/Specify (see Chapter 3, Section 3.10) that has not already been recorded on the Fields tab. When the user clicks on a field in the Fields tab, the Domain Knowledge tab is automatically populated with data relating to that field. The tab name includes the number of domain knowledge records defined for the current field (e.g. the tab name is "*Domain Knowledge (2)*" in the screen capture below).
User Interface


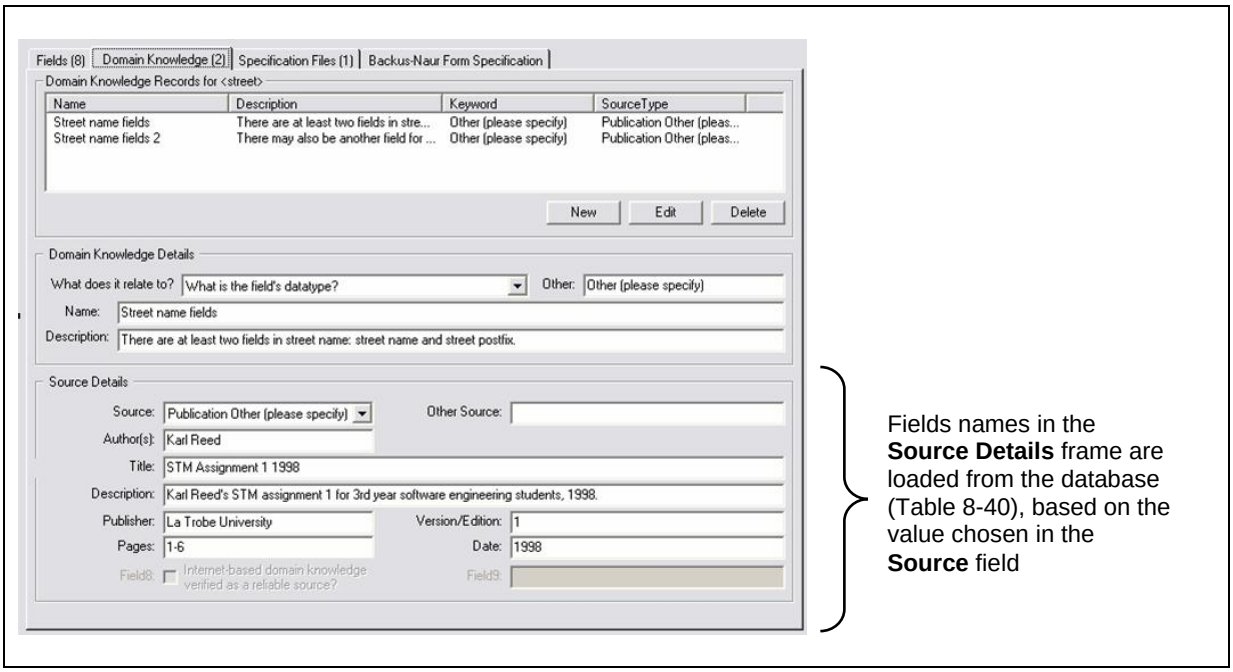
Field	Field Type	Functionality
Continued from previous page…		
Domain Knowledge Records	Record List	Lists all domain knowledge records defined for the currently selected field.
What does it relate to?	Combo Box	Allows the user to identify the GQASV question the domain knowledge relates to. Options are: • What is the field's datatype? • What is the field's set type? • For Set Type = Range: what are min and max values? • For Set Type = List: what are min and max valid data lengths? • Is the field mandatory? • Does the field repeat? • If field repeats, what are min and max repetitions? • Other (please specify)
Other	Text Box	If the user selects “Other (please specify)” from *What does it relate to?*, this field allowing them to describe their answer.
Name	Text Box	A name for the domain knowledge that is being stored.
Description	Text Box	A description of the domain knowledge that is being stored.
Source	Combo Box	The source of the domain knowledge. Options are: • Personal Knowledge • Personal Experience • Book • Textbook • Standard • Conference Paper • Journal Paper • White Paper • Magazine • Newspaper Article • Technical Report • Web Site • Domain Expert • Source Code • Publication Other (please specify) • Other (please specify) The field names in the Source Details frame automatically changes depending on which value is chosen from this field (see Table 8-40).

The names of fields on the Domain Knowledge tab are dynamic, as they depend on the value chosen in the Source field (see last row of Table 8-39). The sources currently supported in ARTT include resources such as personal knowledge, domain knowledge and personal experience, but also textbooks, standards and journal papers (see Table 8-40). The list of sources supported by ARTT may be expanded in future to include other sources like user documentation, help desk instructions, inspection reports, meeting notes or story cards.

Table 8-40: Field names for the Domain Knowledge tab of the Atomic Rules Testing Tool.

Source Type	Field 1	Field 2	Field 3	Field 4	Field 5	Field 6	Field 7	Field 8	Field 9
Personal Knowledge	Source Name:	Learnt Years:	Location:	Details:					
Personal Experience	Source Name:	Experience Years:	Location:	Details:					
Book	Author(s):	Title:	Series Title:	Publisher:	Edition:	Pages:	Date:		
Textbook	Author(s):	Title:	Series Title:	Publisher:	Edition:	Pages:	Date:		
Standard	Author(s):	Title:	Standard No.	Publisher:	Version No:	Pages:	Date:		
Conference Paper	Author(s):	Title:	Proceedings Name:	Publisher:	Volume:	Pages:	Date:		
Journal Paper	Author(s):	Title:	Journal Name:	Publisher:	Volume / Issue:	Pages:	Date:		
White Paper	Author(s):	Title:	White Paper No:	Publisher:	Version No:	Pages:	Date:		
Magazine	Author(s):	Title:	Magazine Name:	Publisher:	Volume / Issue	Pages:	Date:		
Newspaper Article	Author(s):	Title:	Newspaper Name:	Publisher:	Edition:	Pages:	Date:		
Technical Report	Author(s):	Title:	Report No:	Publisher:	Version No:	Pages:	Date:		
Web Site	Author(s):	Title:	URL:	Date Last Updated:	Date Last Accessed:			Verification Check:	Verification Details:
Domain Expert	Name:	Experience Years:	Title:	Discussion Date:	Email Address:	Contact Phone No:			
Source Code	Programmer Name:	Company Name:	Program Name:	Function Name:	Function Details:	Date Last Updated:	Date Last Accessed:		
Publication Other (please specify)	Author(s):	Title:	Description:	Publisher:	Version / Edition:	Pages:	Date:		
Other (please specify)	Desc. 1:	Desc. 2:	Desc.3:	Desc.4:	Desc.5:	Desc.6:	Desc.7:		

Figure 8-10: Example of the Source Details fields populating from the database.

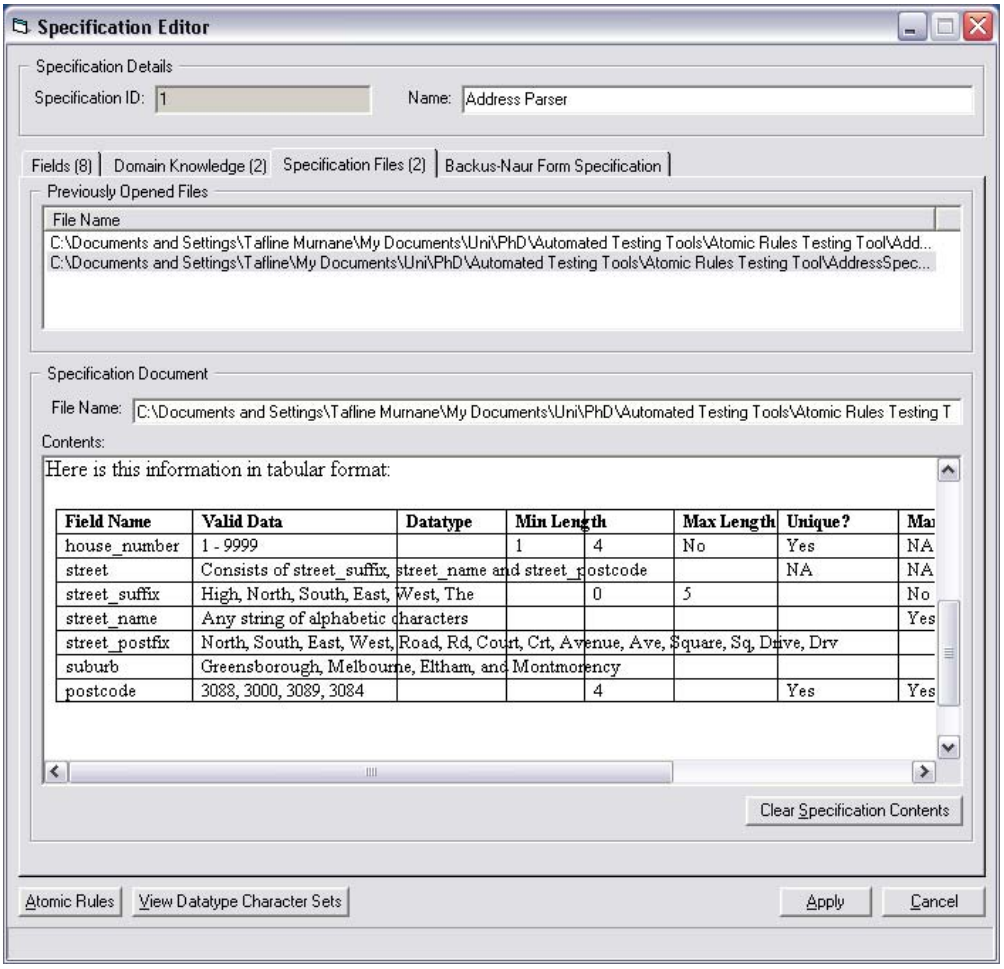


F.4.6.4 The Specification Editor "Specification Files" Tab

The Specification Files tab of the Specification Editor allows users to attach soft copies of specification documents to the currently open specification (Table 8-41).

Table 8-41: The Specification Editor 'Specification Files' tab.

Specification Editor "Specification Files" tab
Purpose: This screen is accessed from the Specification Viewer (Table 8-36). The purpose of this tab is to allow the user to link and view specifications in rich text format (RTF) to the specification that is currently open.
User Interface

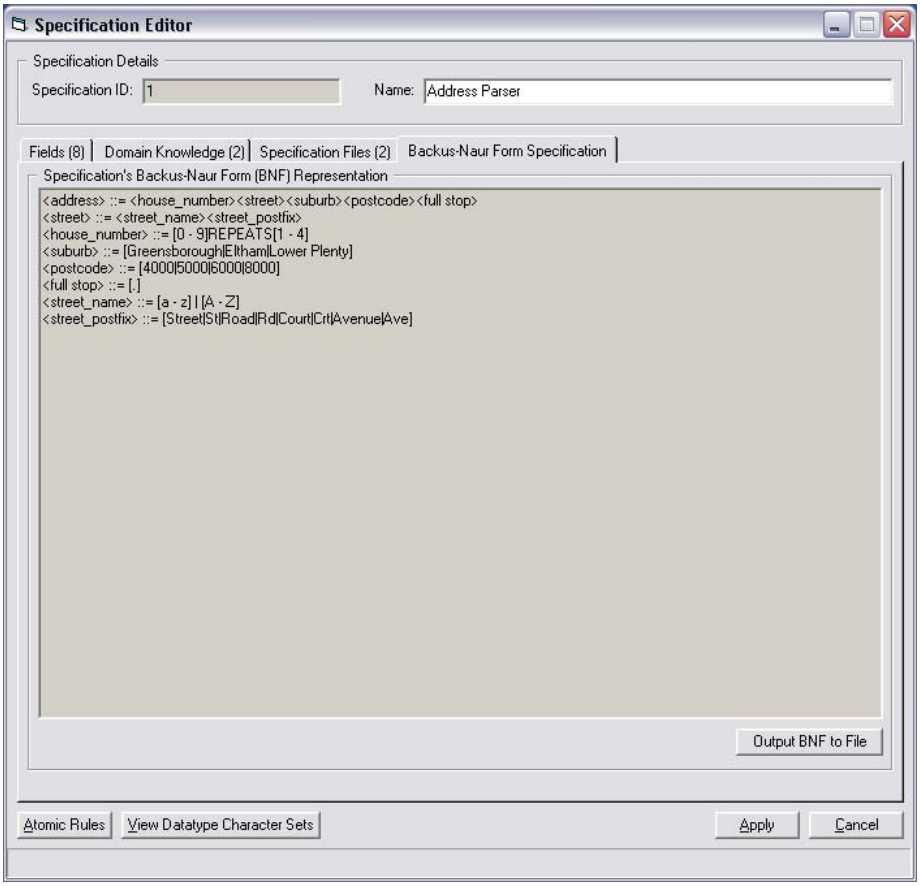
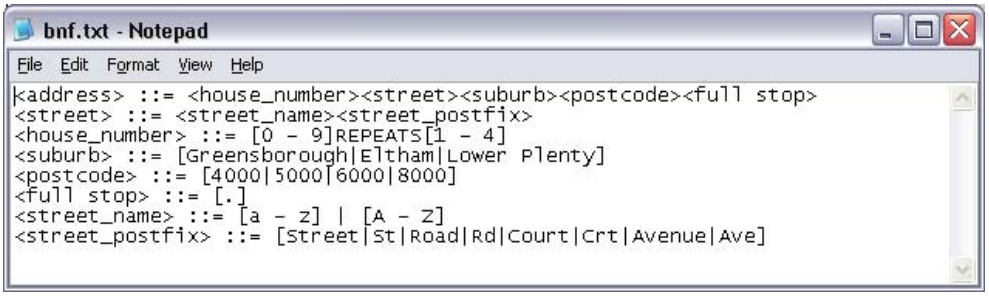


Field	Field Type	Functionality
Previously Opened Files	Record List	Lists all specification documents that have been linked to the current specification.
Contents	Rich Text Box	Displays a specification document.
Load Specification Contents / Clear Specification Contents	Button	When a specification is not loaded, this button is named "*Load Specification Contents*." Clicking it loads a specification into the *Contents* field. When a specification is open, this button is named "*Clear Specification Contents*." Clicking it closes the specification and clears the *Contents* field.

F.4.6.5 The Specification Editor "Backus-Naur Form" Tab

The Backus-Naur Form tab of the Specification Editor allows users to view an automatically produced BNF representation of the specification that is currently open (Table 8-42).

Table 8-42: The Specification Editor 'Backus-Naur Form Specification' tab.

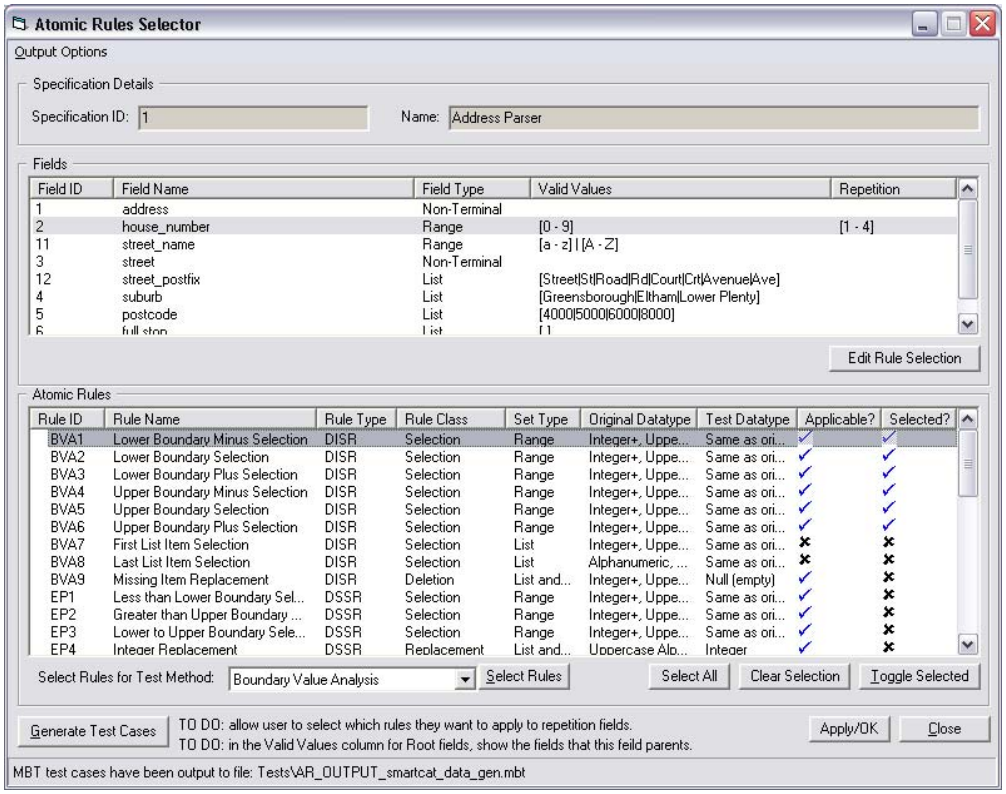
Specification Editor "Backus-Naur Form Specification" tab		
Purpose: This screen is accessed from the Specification Viewer (Table 8-36). The purpose of this tab is to display automatically generated BNF for the current specification and to allow users to output this to file.		
User Interface		
		
		
Field	**Field Type**	**Functionality**
Specification's Backus-Naur Form (BNF)	Text Box	Allows user to view the current specification in BNF.
Output BNF to File	Button	Allows the user to output the BNF to a text file.

F.4.7 The Atomic Rules Selector

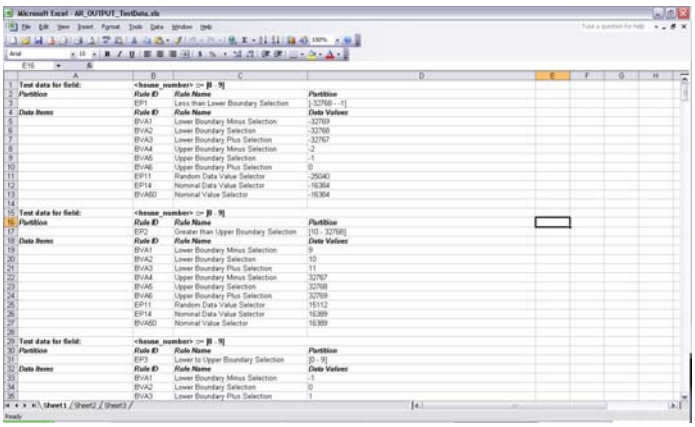
The Atomic Rules Selector is allows users to apply Atomic Rules from EP, BVA and ST to the currently open specification, in order to generate black-box test data (Table 8-43).

Table 8-43: The Atomic Rules Selector.

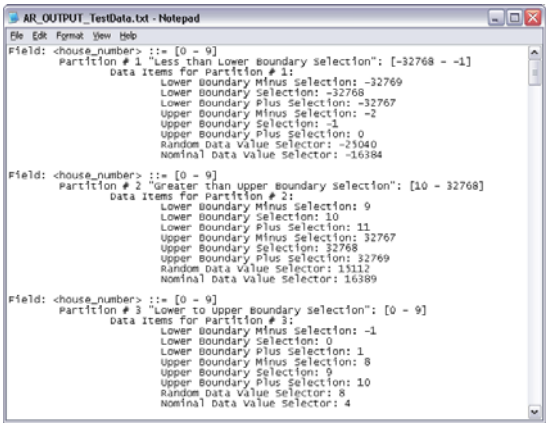
Atomic Rules Selector
Purpose: This screen can be accessed either from the Specification Viewer (Table 8-36) or the Specification Editor (Table 8-37). It allows users to apply Atomic Rules to a specification to automatically generate test data that can be output to an Excel Spreadsheet or a flat text file.
User Interface



Output Files



Continued from previous page…



Field	Field Type	Functionality
Output Options	Menu	This specifies the format in which the test data/cases are output. Options are: • Microsoft Excel Spreadsheet • Text file The default setting for this field is that both options are selected.
Specification ID	Text Box	The current specification's identifier.
Name	Text Box	The current specification's name.
Fields	Record List	Lists all fields defined for the current specification. When a field is selected, the Atomic Rules shows which Atomic Rules have been applied to that field.
Atomic Rules	Record List	Lists all Atomic Rules defined in the system and shows which can be applied (*Applicable?*) and have been applied (*Selected?*) to the current field.
Select Rules for Test Method	Combo Box	Allows the user to apply all Atomic Rules from a particular black-box method to the current field. Options are: • Boundary Value Analysis • Equivalence Partitioning • Syntax Testing
Select All	Button	Selects all Atomic Rules that can be applied to the current field.
Clear Selection	Button	Clears the selection of all Atomic Rules for the current field.
Toggle Selected	Button	Toggles the selection of the currently selected Atomic Rule for the currently field (i.e. changes a tick to a cross and visa versa).
Generate Test Cases	Button	Automatically generates a set of test data/cases for the current specification by applying all selected Atomic Rules to each field. Test cases are output to file automatically according to the file types chosen in the *Output Options* field.
Apply/OK	Button	Each time a user makes changes to the Atomic Rules applied to a particular field, they must click *Apply/OK* to save those changes to the database. They must also do this before generating test cases.
Close	Button	Closes the screen and returns the user to the calling screen (either the *Specification Editor* or the *Specification Viewer*).

F.5 Pseudo Code for Test Data Generation

The following figures contain pseudo code for the selection of list-based and range-based equivalence classes. This process is based on the ‘start position’ and ‘end position’ of the specific Data-Set Selection Rule that is being applied to the field under test (see Chapter 4, Section 4.3).

Figure 8-11: Pseudo code for selecting an equivalence class from a list-based field.

```
IF the field under test is a List THEN

    'Assign the start positions of the partition
    IF the chosen start position = first field value THEN
        Partition start position = 1
    ELSE IF the chosen start position = nominal value THEN
        Partition start position = number of values in the list / 2
    ELSE IF the chosen start position = random value THEN
        Partition start position = select a random number
    END IF

    'Assign the end positions of the partition -> must be greater than start position
    IF the chosen end position = last field value THEN
        Partition end position = count of values in list
    ELSE IF the chosen end position = nominal value THEN
        Partition end position = count of values in list / 2
    ELSE IF the chosen end position = random value THEN
        Partition end position = select a random value from the list
    END IF

    'Build the partition
    Count = 0
    FOR each value in the List
        IF Count is between the Start and End Positions THEN
            Add the value to the partition
        END IF
        Count = Count + 1
    NEXT value

END IF
```

Figure 8-12: Pseudo code for selecting a partition from a range-based field

```
IF the field under is a Range THEN

    'Assign start position of the partition
    IF the chosen start position = datatype lower boundary minus THEN
        Partition lower boundary = value just below lower boundary of field's datatype
    ELSE IF the chosen start position = datatype lower boundary THEN
        Partition lower boundary = value on lower boundary of field's datatype
    ELSE IF the chosen start position = datatype lower boundary plus THEN
        Partition lower boundary =  value just above lower boundary of field's datatype
    ELSE IF the chosen start position = datatype upper boundary minus THEN
        Partition lower boundary =  value just above lower boundary of field's datatype
    ELSE IF the chosen start position = datatype upper boundary THEN
        Partition lower boundary = value on upper boundary of field's datatype
    ELSE IF the chosen start position = datatype upper boundary plus THEN
        Partition lower boundary = value just above upper boundary of field's datatype
    ELSE IF the chosen start position = field lower boundary minus THEN
        Partition lower boundary = value just below lower boundary of field
    ELSE IF the chosen start position = field lower boundary OR first field value Then
        Partition lower boundary = value on lower boundary of field
    ELSE IF the chosen start position = field lower boundary plus THEN
        Partition lower boundary = value just above lower boundary of field
    ELSE IF the chosen start position = field upper boundary minus THEN
        Partition lower boundary = value just below upper boundary of field
    ELSE IF the chosen start position = field upper boundary OR last field value THEN
        Partition lower boundary = value on upper boundary of field
    ELSE IF the chosen start position = upper boundary plus THEN
        Partition lower boundary = value just above upper boundary of field
    ELSE IF the chosen start position = nominal value Then
        Partition lower boundary = upper boundary – lower boundary / 2
    ELSE IF the chosen start position = random value Then
        Partition lower boundary = random value between lower and upper boundaries
    END IF

    'Assign end position of the partition
    IF the chosen end position = datatype lower boundary minus THEN
        Partition lower boundary = value just below lower boundary of field's datatype
    ELSE IF the chosen end position = datatype lower boundary THEN
        Partition lower boundary = value on lower boundary of field's datatype
    ELSE IF the chosen end position = datatype lower boundary plus THEN
        Partition lower boundary =  value just above lower boundary of field's datatype
    ELSE IF the chosen end position = datatype upper boundary minus THEN
        Partition lower boundary =  value just above lower boundary of field's datatype
    ELSE IF the chosen end position = datatype upper boundary THEN
        Partition lower boundary = value on upper boundary of field's datatype
    ELSE IF the chosen end position = datatype upper boundary plus THEN
        Partition lower boundary = value just above upper boundary of field's datatype
    ELSE IF the chosen end position = field lower boundary minus THEN
        Partition lower boundary = value just below lower boundary of field
    ELSE IF the chosen end position = field lower boundary OR first field value Then
        Partition lower boundary = value on lower boundary of field
    ELSE IF the chosen end position = field lower boundary plus THEN
        Partition lower boundary = value just above lower boundary of field
    ELSE IF the chosen end position = field upper boundary minus THEN
        Partition lower boundary = value just below upper boundary of field
    ELSE IF the chosen end position = field upper boundary OR last field value THEN
        Partition lower boundary = value on upper boundary of field
    ELSE IF the chosen end position = upper boundary plus THEN
        Partition lower boundary = value just above upper boundary of field
    ELSE IF the chosen end position = nominal value THEN
        Partition lower boundary = upper boundary – lower boundary / 2
```

```
    'Continued from previous page…
    ELSE IF the chosen end position = random value THEN
        Partition lower boundary = random value between lower and upper boundaries
    END IF

    'Select the partition
    Partition = partition lower boundary to partition upper boundary

END IF
```

F.6 Datatypes Defined in the Atomic Rules Testing Tool

The following tables identify all datatypes and characters that are defined in the Atomic Rules Testing Tool (ARTT), which can be viewed through the Character Viewer screen (see Section F.4.4). These datatypes and characters are predominantly based on characters that appear in the ASCII table (see Appendix G).

Table 8-44: Datatypes defined in the Atomic Rules Testing Tool.

Datatype Name	Set Type	Base Datatype?	Composed Of
Boolean	List	Yes	{Boolean}
Numeric	List and Range	Yes	{Numeric}
Integer	List and Range	Yes	{Integer}
Real	List and Range	Yes	{Real}
Lowercase Alpha	List and Range	Yes	{Lowercase Alpha}
Uppercase Alpha	List and Range	Yes	{Uppercase Alpha}
Alpha	List	No	{Lowercase Alpha} U {Uppercase Alpha} -(Lowercase Alpha LB- & UB+& Uppercase Alpha LB- & UB+)
Alphanumeric	List	No	{Numeric} U {Alpha} - (Numeric LB- & UB+& Alpha LB- & AlphaUB+)
Control Character	List and Range	Yes	{Control Character}
Symbol (Set 1)	List and Range	Yes	{Symbol (Set 1)}
Symbol (Set 2)	List and Range	Yes	{Symbol (Set 2)}
Symbol (Set 3)	List and Range	Yes	{Symbol (Set 3)}
Symbol (Set 4)	List and Range	Yes	{Symbol (Set 4)}
Symbol	List	No	{Symbol (Set 1)} U {Symbol (Set 2)} U {Symbol (Set 3)} U {Symbol (Set 4)}
Null (empty)	List and Range	Yes	{Null (empty)}
Non-Alphanumeric	List	No	{Symbol (Set 1)} U {Symbol (Set 2)} U {Symbol (Set 3)} U {Symbol (Set 4) U {Control Character}} - (Symbol (Sets 1 to 4) LB- & UB+& Control Character LB- & UB+)

Table 8-45: Characters within each datatype that are defined in the Atomic Rules Testing Tool.

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Boolean	48	0	Zero	0	First Field Value
Boolean	49	1	One	1	Last Field Value
Numeric	47	/	Forward Slash	/	Datatype Lower Boundary -
Numeric	48	0	Zero	0	Datatype Lower Boundary
Numeric	49	1	One	1	Datatype Lower Boundary +
Numeric	50	2	Two	2	Non Descript Position
Numeric	51	3	Three	3	Non Descript Position
Numeric	52	4	Four	4	Nominal Value
Numeric	53	5	Five	5	Non Descript Position
Numeric	54	6	Six	6	Non Descript Position
Numeric	55	7	Seven	7	Non Descript Position
Numeric	56	8	Eight	8	Datatype Upper Boundary -
Numeric	57	9	Nine	9	Datatype Upper Boundary
Numeric	58	:	Colon	:	Datatype Upper Boundary +
Integer	1048574	Integer - Lower Boundary	Integer - Smallest	-32768	Datatype Lower Boundary
Integer	1048575	Integer - Upper Boundary	Integer - Largest	32768	Datatype Upper Boundary
Real	1048572	Real - Lower Boundary	Real - Smallest	-32767	Datatype Lower Boundary
Real	1048573	Real - Upper Boundary	Real - Largest	32767	Datatype Upper Boundary
Lowercase Alpha	96	`	Single Left Quote	`	Datatype Lower Boundary -
Lowercase Alpha	97	a	a	a	Datatype Lower Boundary
Lowercase Alpha	98	b	b	b	Datatype Lower Boundary +
Lowercase Alpha	99	c	c	c	Non Descript Position
Lowercase Alpha	100	d	d	d	Non Descript Position
Lowercase Alpha	101	e	e	e	Non Descript Position
Lowercase Alpha	102	f	f	f	Non Descript Position
Lowercase Alpha	103	g	g	g	Non Descript Position
Lowercase Alpha	104	h	h	h	Non Descript Position
Lowercase Alpha	105	i	i	i	Non Descript Position
Lowercase Alpha	106	j	j	j	Non Descript Position
Lowercase Alpha	107	k	k	k	Non Descript Position
Lowercase Alpha	108	l	l	l	Non Descript Position
Lowercase Alpha	109	m	m	m	Nominal Value
Lowercase Alpha	110	n	n	n	Non Descript Position
Lowercase Alpha	111	o	o	o	Non Descript Position
Lowercase Alpha	112	p	p	p	Non Descript Position
Lowercase Alpha	113	q	q	q	Non Descript Position
Lowercase Alpha	114	r	r	r	Non Descript Position
Lowercase Alpha	115	s	s	s	Non Descript Position
Lowercase Alpha	116	t	t	t	Non Descript Position
Lowercase Alpha	117	u	u	u	Non Descript Position
Lowercase Alpha	118	v	v	v	Non Descript Position
Lowercase Alpha	119	w	w	w	Non Descript Position
Lowercase Alpha	120	x	x	x	Non Descript Position
Lowercase Alpha	121	y	y	y	Datatype Upper Boundary -
Lowercase Alpha	122	z	z	z	Datatype Upper Boundary
Lowercase Alpha	123	{	Open Curly Brace	{	Datatype Upper Boundary +
Uppercase Alpha	64	@	At	@	Datatype Lower Boundary -
Uppercase Alpha	65	A	A	A	Datatype Lower Boundary

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Uppercase Alpha	66	B	B	B	Datatype Lower Boundary +
Uppercase Alpha	67	C	C	C	Non Descript Position
Uppercase Alpha	68	D	D	D	Non Descript Position
Uppercase Alpha	69	E	E	E	Non Descript Position
Uppercase Alpha	70	F	F	F	Non Descript Position
Uppercase Alpha	71	G	G	G	Non Descript Position
Uppercase Alpha	72	H	H	H	Non Descript Position
Uppercase Alpha	73	I	I	I	Non Descript Position
Uppercase Alpha	74	J	J	J	Non Descript Position
Uppercase Alpha	75	K	K	K	Non Descript Position
Uppercase Alpha	76	L	L	L	Non Descript Position
Uppercase Alpha	77	M	M	M	Nominal Value
Uppercase Alpha	78	N	N	N	Non Descript Position
Uppercase Alpha	79	O	O	O	Non Descript Position
Uppercase Alpha	80	P	P	P	Non Descript Position
Uppercase Alpha	81	Q	Q	Q	Non Descript Position
Uppercase Alpha	82	R	R	R	Non Descript Position
Uppercase Alpha	83	S	S	S	Non Descript Position
Uppercase Alpha	84	T	T	T	Non Descript Position
Uppercase Alpha	85	U	U	U	Non Descript Position
Uppercase Alpha	86	V	V	V	Non Descript Position
Uppercase Alpha	87	W	W	W	Non Descript Position
Uppercase Alpha	88	X	X	X	Non Descript Position
Uppercase Alpha	89	Y	Y	Y	Datatype Upper Boundary -
Uppercase Alpha	90	Z	Z	Z	Datatype Upper Boundary
Uppercase Alpha	91	[	Open Square Brace	[	Datatype Upper Boundary +
Alpha	65	A	A	A	First Field Value
Alpha	66	B	B	B	Non Descript Position
Alpha	67	C	C	C	Non Descript Position
Alpha	68	D	D	D	Non Descript Position
Alpha	69	E	E	E	Non Descript Position
Alpha	70	F	F	F	Non Descript Position
Alpha	71	G	G	G	Non Descript Position
Alpha	72	H	H	H	Non Descript Position
Alpha	73	I	I	I	Non Descript Position
Alpha	74	J	J	J	Non Descript Position
Alpha	75	K	K	K	Non Descript Position
Alpha	76	L	L	L	Non Descript Position
Alpha	77	M	M	M	Non Descript Position
Alpha	78	N	N	N	Non Descript Position
Alpha	79	O	O	O	Non Descript Position
Alpha	80	P	P	P	Non Descript Position
Alpha	81	Q	Q	Q	Non Descript Position
Alpha	82	R	R	R	Non Descript Position
Alpha	83	S	S	S	Non Descript Position
Alpha	84	T	T	T	Non Descript Position
Alpha	85	U	U	U	Non Descript Position
Alpha	86	V	V	V	Non Descript Position
Alpha	87	W	W	W	Non Descript Position
Alpha	88	X	X	X	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Alpha	89	Y	Y	Y	Non Descript Position
Alpha	90	Z	Z	Z	Non Descript Position
Alpha	97	a	a	a	Nominal Value
Alpha	98	b	b	b	Non Descript Position
Alpha	99	c	c	c	Non Descript Position
Alpha	100	d	d	d	Non Descript Position
Alpha	101	e	e	e	Non Descript Position
Alpha	102	f	f	f	Non Descript Position
Alpha	103	g	g	g	Non Descript Position
Alpha	104	h	h	h	Non Descript Position
Alpha	105	i	i	i	Non Descript Position
Alpha	106	j	j	j	Non Descript Position
Alpha	107	k	k	k	Non Descript Position
Alpha	108	l	l	l	Non Descript Position
Alpha	109	m	m	m	Non Descript Position
Alpha	110	n	n	n	Non Descript Position
Alpha	111	o	o	o	Non Descript Position
Alpha	112	p	p	p	Non Descript Position
Alpha	113	q	q	q	Non Descript Position
Alpha	114	r	r	r	Non Descript Position
Alpha	115	s	s	s	Non Descript Position
Alpha	116	t	t	t	Non Descript Position
Alpha	117	u	u	u	Non Descript Position
Alpha	118	v	v	v	Non Descript Position
Alpha	119	w	w	w	Non Descript Position
Alpha	120	x	x	x	Non Descript Position
Alpha	121	y	y	y	Non Descript Position
Alpha	122	z	z	z	Last Field Value
Alphanumeric	48	0	Zero	0	First Field Value
Alphanumeric	49	1	One	1	Non Descript Position
Alphanumeric	50	2	Two	2	Non Descript Position
Alphanumeric	51	3	Three	3	Non Descript Position
Alphanumeric	52	4	Four	4	Non Descript Position
Alphanumeric	53	5	Five	5	Non Descript Position
Alphanumeric	54	6	Six	6	Non Descript Position
Alphanumeric	55	7	Seven	7	Non Descript Position
Alphanumeric	56	8	Eight	8	Non Descript Position
Alphanumeric	57	9	Nine	9	Non Descript Position
Alphanumeric	65	A	A	A	Non Descript Position
Alphanumeric	66	B	B	B	Non Descript Position
Alphanumeric	67	C	C	C	Non Descript Position
Alphanumeric	68	D	D	D	Non Descript Position
Alphanumeric	69	E	E	E	Non Descript Position
Alphanumeric	70	F	F	F	Non Descript Position
Alphanumeric	71	G	G	G	Non Descript Position
Alphanumeric	72	H	H	H	Non Descript Position
Alphanumeric	73	I	I	I	Non Descript Position
Alphanumeric	74	J	J	J	Non Descript Position
Alphanumeric	75	K	K	K	Non Descript Position
Alphanumeric	76	L	L	L	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Alphanumeric	77	M	M	M	Non Descript Position
Alphanumeric	78	N	N	N	Non Descript Position
Alphanumeric	79	O	O	O	Non Descript Position
Alphanumeric	80	P	P	P	Non Descript Position
Alphanumeric	81	Q	Q	Q	Non Descript Position
Alphanumeric	82	R	R	R	Non Descript Position
Alphanumeric	83	S	S	S	Non Descript Position
Alphanumeric	84	T	T	T	Non Descript Position
Alphanumeric	85	U	U	U	Nominal Value
Alphanumeric	86	V	V	V	Non Descript Position
Alphanumeric	87	W	W	W	Non Descript Position
Alphanumeric	88	X	X	X	Non Descript Position
Alphanumeric	89	Y	Y	Y	Non Descript Position
Alphanumeric	90	Z	Z	Z	Non Descript Position
Alphanumeric	97	a	a	a	Non Descript Position
Alphanumeric	98	b	b	b	Non Descript Position
Alphanumeric	99	c	c	c	Non Descript Position
Alphanumeric	100	d	d	d	Non Descript Position
Alphanumeric	101	e	e	e	Non Descript Position
Alphanumeric	102	f	f	f	Non Descript Position
Alphanumeric	103	g	g	g	Non Descript Position
Alphanumeric	104	h	h	h	Non Descript Position
Alphanumeric	105	i	i	i	Non Descript Position
Alphanumeric	106	j	j	j	Non Descript Position
Alphanumeric	107	k	k	k	Non Descript Position
Alphanumeric	108	l	l	l	Non Descript Position
Alphanumeric	109	m	m	m	Non Descript Position
Alphanumeric	110	n	n	n	Non Descript Position
Alphanumeric	111	o	o	o	Non Descript Position
Alphanumeric	112	p	p	p	Non Descript Position
Alphanumeric	113	q	q	q	Non Descript Position
Alphanumeric	114	r	r	r	Non Descript Position
Alphanumeric	115	s	s	s	Non Descript Position
Alphanumeric	116	t	t	t	Non Descript Position
Alphanumeric	117	u	u	u	Non Descript Position
Alphanumeric	118	v	v	v	Non Descript Position
Alphanumeric	119	w	w	w	Non Descript Position
Alphanumeric	120	x	x	x	Non Descript Position
Alphanumeric	121	y	y	y	Non Descript Position
Alphanumeric	122	z	z	z	Last Field Value
Control Character	0	NUL	Null	ctrl-@	Datatype Lower Boundary
Control Character	1	SOH	Start of Heading	ctrl-A	Datatype Lower Boundary +
Control Character	2	STX	Start of Text	ctrl-B	Non Descript Position
Control Character	3	ETX	End of Text	ctrl-C	Non Descript Position
Control Character	4	EOT	End of Xmit	ctrl-D	Non Descript Position
Control Character	5	ENQ	Enquiry	ctrl-E	Non Descript Position
Control Character	6	ACK	Acknowledge	ctrl-F	Non Descript Position
Control Character	7	BEL	Bell	ctrl-G	Non Descript Position
Control Character	8	BS	Backspace	ctrl-H	Non Descript Position
Control Character	9	HT	Horizontal Tab	ctrl-I	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Control Character	10	LF	Line Feed	ctrl-J	Non Descript Position
Control Character	11	VT	Vertical Tab	ctrl-K	Non Descript Position
Control Character	12	FF	Form Feed	ctrl-L	Non Descript Position
Control Character	13	CR	Carriage Feed	ctrl-M	Non Descript Position
Control Character	14	SO	Shift Out	ctrl-N	Non Descript Position
Control Character	15	SI	Shift In	ctrl-O	Non Descript Position
Control Character	16	DLE	Data Line Escape	ctrl-P	Nominal Value
Control Character	17	DC1	Device Control 1	ctrl-Q	Non Descript Position
Control Character	18	DC2	Device Control 2	ctrl-R	Non Descript Position
Control Character	19	DC3	Device Control 3	ctrl-S	Non Descript Position
Control Character	20	DC4	Device Control 4	ctrl-T	Non Descript Position
Control Character	21	NAK	Neg Acknowledge	ctrl-U	Non Descript Position
Control Character	22	SYN	Synchronous Idel	ctrl-V	Non Descript Position
Control Character	23	ETB	End of Xmit Block	ctrl-W	Non Descript Position
Control Character	24	CAN	Cancel	ctrl-X	Non Descript Position
Control Character	25	EM	End of Medium	ctrl-Y	Non Descript Position
Control Character	26	SUB	Substitute	ctrl-Z	Non Descript Position
Control Character	27	ESC	Escape	ctrl-[	Non Descript Position
Control Character	28	FS	File Separator	ctrl-\	Non Descript Position
Control Character	29	GS	Group Separator	ctrl-]	Non Descript Position
Control Character	30	RS	Record Separator	ctrl-^	Datatype Upper Boundary -
Control Character	31	US	Unit Separator	ctrl-_	Datatype Upper Boundary
Control Character	32	Space	Space		Datatype Upper Boundary +
Symbol (Set 1)	31	US	Unit Separator	ctrl-_	Datatype Lower Boundary -
Symbol (Set 1)	32	Space	Space		Datatype Lower Boundary
Symbol (Set 1)	33	!	Exclamation Mark	!	Datatype Lower Boundary +
Symbol (Set 1)	34	"	Double Quote	"	Non Descript Position
Symbol (Set 1)	35	#	Hash	#	Non Descript Position
Symbol (Set 1)	36	$	Dollar	$	Non Descript Position
Symbol (Set 1)	37	%	Percent	%	Non Descript Position
Symbol (Set 1)	38	&	Ampersand	&	Non Descript Position
Symbol (Set 1)	39	'	Single Right Quote	'	Nominal Value
Symbol (Set 1)	40	(	Open Round Brace	(	Non Descript Position
Symbol (Set 1)	41	)	Close Round Brace	)	Non Descript Position
Symbol (Set 1)	42	*	Kleene Star	*	Non Descript Position
Symbol (Set 1)	43	+	Addition Sign	+	Non Descript Position
Symbol (Set 1)	44	,	Comma	,	Non Descript Position
Symbol (Set 1)	45	-	Hyphen	-	Non Descript Position
Symbol (Set 1)	46	.	Period	.	Datatype Upper Boundary -
Symbol (Set 1)	47	/	Forward Slash	/	Datatype Upper Boundary
Symbol (Set 1)	48	0	Zero	0	Datatype Upper Boundary +
Symbol (Set 2)	57	9	Nine	9	Datatype Lower Boundary -
Symbol (Set 2)	58	:	Colon	:	Datatype Lower Boundary
Symbol (Set 2)	59	;	Semi-Colon	;	Datatype Lower Boundary +
Symbol (Set 2)	60	<	Backwards Arrow	<	Non Descript Position
Symbol (Set 2)	61	=	Equal Sign	=	Nominal Value
Symbol (Set 2)	62	>	Forwards Arrow	>	Non Descript Position
Symbol (Set 2)	63	?	Question Mark	?	Datatype Upper Boundary -
Symbol (Set 2)	64	@	At	@	Datatype Upper Boundary
Symbol (Set 2)	65	A	A	A	Datatype Upper Boundary +

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Symbol (Set 3)	90	Z	Z	Z	Datatype Lower Boundary -
Symbol (Set 3)	91	[	Open Square Brace	[	Datatype Lower Boundary
Symbol (Set 3)	92	\	Backwards Slash	\	Datatype Lower Boundary +
Symbol (Set 3)	93	]	Close Square Brace	]	Nominal Value
Symbol (Set 3)	94	^	Hat	^	Non Descript Position
Symbol (Set 3)	95	_	Underscore	_	Datatype Upper Boundary -
Symbol (Set 3)	96	`	Single Left Quote	`	Datatype Upper Boundary
Symbol (Set 3)	97	a	a	a	Datatype Upper Boundary +
Symbol (Set 4)	122	z	z	z	Datatype Lower Boundary -
Symbol (Set 4)	123	{	Open Curly Brace	{	Datatype Lower Boundary
Symbol (Set 4)	124	|	Bar	|	Datatype Lower Boundary +
Symbol (Set 4)	125	}	Close Curly Brace	}	Nominal Value
Symbol (Set 4)	126	~	Tilde	~	Datatype Upper Boundary -
Symbol (Set 4)	127	DEL	Delete	DEL	Datatype Upper Boundary
Symbol	32	Space	Space		First Field Value
Symbol	33	!	Exclamation Mark	!	Non Descript Position
Symbol	34	"	Double Quote	"	Non Descript Position
Symbol	35	#	Hash	#	Non Descript Position
Symbol	36	$	Dollar	$	Non Descript Position
Symbol	37	%	Percent	%	Non Descript Position
Symbol	38	&	Ampersand	&	Non Descript Position
Symbol	39	'	Single Right Quote	'	Non Descript Position
Symbol	40	(	Open Round Brace	(	Non Descript Position
Symbol	41	)	Close Round Brace	)	Non Descript Position
Symbol	42	*	Kleene Star	*	Non Descript Position
Symbol	43	+	Addition Sign	+	Non Descript Position
Symbol	44	,	Comma	,	Non Descript Position
Symbol	45	-	Hyphen	-	Non Descript Position
Symbol	46	.	Period	.	Non Descript Position
Symbol	47	/	Forward Slash	/	Non Descript Position
Symbol	58	:	Colon	:	Nominal Value
Symbol	59	;	Semi-Colon	;	Non Descript Position
Symbol	60	<	Backwards Arrow	<	Non Descript Position
Symbol	61	=	Equal Sign	=	Non Descript Position
Symbol	62	>	Forwards Arrow	>	Non Descript Position
Symbol	63	?	Question Mark	?	Non Descript Position
Symbol	64	@	At	@	Non Descript Position
Symbol	91	[	Open Square Brace	[	Non Descript Position
Symbol	92	\	Backwards Slash	\	Non Descript Position
Symbol	93	]	Close Square Brace	]	Non Descript Position
Symbol	94	^	Hat	^	Non Descript Position
Symbol	95	_	Underscore	_	Non Descript Position
Symbol	96	`	Single Left Quote	`	Non Descript Position
Symbol	123	{	Open Curly Brace	{	Non Descript Position
Symbol	124	|	Bar	|	Non Descript Position
Symbol	125	}	Close Curly Brace	}	Non Descript Position
Symbol	126	~	Tilde	~	Non Descript Position
Symbol	127	DEL	Delete	DEL	Last Field Value
Null (empty)	100000	Null	Null - empty set		Non Descript Position
Non-Alphanumeric	0	NUL	Null	ctrl-@	First Field Value

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Non-Alphanumeric	1	SOH	Start of Heading	ctrl-A	Non Descript Position
Non-Alphanumeric	2	STX	Start of Text	ctrl-B	Non Descript Position
Non-Alphanumeric	3	ETX	End of Text	ctrl-C	Non Descript Position
Non-Alphanumeric	4	EOT	End of Xmit	ctrl-D	Non Descript Position
Non-Alphanumeric	5	ENQ	Enquiry	ctrl-E	Non Descript Position
Non-Alphanumeric	6	ACK	Acknowledge	ctrl-F	Non Descript Position
Non-Alphanumeric	7	BEL	Bell	ctrl-G	Non Descript Position
Non-Alphanumeric	8	BS	Backspace	ctrl-H	Non Descript Position
Non-Alphanumeric	9	HT	Horizontal Tab	ctrl-I	Non Descript Position
Non-Alphanumeric	10	LF	Line Feed	ctrl-J	Non Descript Position
Non-Alphanumeric	11	VT	Vertical Tab	ctrl-K	Non Descript Position
Non-Alphanumeric	12	FF	Form Feed	ctrl-L	Non Descript Position
Non-Alphanumeric	13	CR	Carriage Feed	ctrl-M	Non Descript Position
Non-Alphanumeric	14	SO	Shift Out	ctrl-N	Non Descript Position
Non-Alphanumeric	15	SI	Shift In	ctrl-O	Non Descript Position
Non-Alphanumeric	16	DLE	Data Line Escape	ctrl-P	Non Descript Position
Non-Alphanumeric	17	DC1	Device Control 1	ctrl-Q	Non Descript Position
Non-Alphanumeric	18	DC2	Device Control 2	ctrl-R	Non Descript Position
Non-Alphanumeric	19	DC3	Device Control 3	ctrl-S	Non Descript Position
Non-Alphanumeric	20	DC4	Device Control 4	ctrl-T	Non Descript Position
Non-Alphanumeric	21	NAK	Neg Acknowledge	ctrl-U	Non Descript Position
Non-Alphanumeric	22	SYN	Synchronous Idel	ctrl-V	Non Descript Position
Non-Alphanumeric	23	ETB	End of Xmit Block	ctrl-W	Non Descript Position
Non-Alphanumeric	24	CAN	Cancel	ctrl-X	Non Descript Position
Non-Alphanumeric	25	EM	End of Medium	ctrl-Y	Non Descript Position
Non-Alphanumeric	26	SUB	Substitute	ctrl-Z	Non Descript Position
Non-Alphanumeric	27	ESC	Escape	ctrl-[	Non Descript Position
Non-Alphanumeric	28	FS	File Separator	ctrl-\	Non Descript Position
Non-Alphanumeric	29	GS	Group Separator	ctrl-]	Non Descript Position
Non-Alphanumeric	30	RS	Record Separator	ctrl-^	Non Descript Position
Non-Alphanumeric	31	US	Unit Separator	ctrl-_	Non Descript Position
Non-Alphanumeric	32	Space	Space		Nominal Value
Non-Alphanumeric	33	!	Exclamation Mark	!	Non Descript Position
Non-Alphanumeric	34	"	Double Quote	"	Non Descript Position
Non-Alphanumeric	35	#	Hash	#	Non Descript Position
Non-Alphanumeric	36	$	Dollar	$	Non Descript Position
Non-Alphanumeric	37	%	Percent	%	Non Descript Position
Non-Alphanumeric	38	&	Ampersand	&	Non Descript Position
Non-Alphanumeric	39	'	Single Right Quote	'	Non Descript Position
Non-Alphanumeric	40	(	Open Round Brace	(	Non Descript Position
Non-Alphanumeric	41	)	Close Round Brace	)	Non Descript Position
Non-Alphanumeric	42	*	Kleene Star	*	Non Descript Position
Non-Alphanumeric	43	+	Addition Sign	+	Non Descript Position
Non-Alphanumeric	44	,	Comma	,	Non Descript Position
Non-Alphanumeric	45	-	Hyphen	-	Non Descript Position
Non-Alphanumeric	46	.	Period	.	Non Descript Position
Non-Alphanumeric	47	/	Forward Slash	/	Non Descript Position
Non-Alphanumeric	58	:	Colon	:	Non Descript Position
Non-Alphanumeric	59	;	Semi-Colon	;	Non Descript Position
Non-Alphanumeric	60	<	Backwards Arrow	<	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
Non-Alphanumeric	61	=	Equal Sign	=	Non Descript Position
Non-Alphanumeric	62	>	Forwards Arrow	>	Non Descript Position
Non-Alphanumeric	63	?	Question Mark	?	Non Descript Position
Non-Alphanumeric	64	@	At	@	Non Descript Position
Non-Alphanumeric	91	[	Open Square Brace	[	Non Descript Position
Non-Alphanumeric	92	\	Backwards Slash	\	Non Descript Position
Non-Alphanumeric	93	]	Close Square Brace	]	Non Descript Position
Non-Alphanumeric	94	^	Hat	^	Non Descript Position
Non-Alphanumeric	95	_	Underscore	_	Non Descript Position
Non-Alphanumeric	96	`	Single Left Quote	`	Non Descript Position
Non-Alphanumeric	123	{	Open Curly Brace	{	Non Descript Position
Non-Alphanumeric	124	|	Bar	|	Non Descript Position
Non-Alphanumeric	125	}	Close Curly Brace	}	Non Descript Position
Non-Alphanumeric	126	~	Tilde	~	Non Descript Position
Non-Alphanumeric	127	DEL	Delete	DEL	Last Field Value
ASCII	0	NUL	Null	ctrl-@	First Character
ASCII	1	SOH	Start of Heading	ctrl-A	First Character +
ASCII	2	STX	Start of Text	ctrl-B	First Character ++
ASCII	3	ETX	End of Text	ctrl-C	Non Descript Position
ASCII	4	EOT	End of Xmit	ctrl-D	Non Descript Position
ASCII	5	ENQ	Enquiry	ctrl-E	Non Descript Position
ASCII	6	ACK	Acknowledge	ctrl-F	Non Descript Position
ASCII	7	BEL	Bell	ctrl-G	Non Descript Position
ASCII	8	BS	Backspace	ctrl-H	Non Descript Position
ASCII	9	HT	Horizontal Tab	ctrl-I	Non Descript Position
ASCII	10	LF	Line Feed	ctrl-J	Non Descript Position
ASCII	11	VT	Vertical Tab	ctrl-K	Non Descript Position
ASCII	12	FF	Form Feed	ctrl-L	Non Descript Position
ASCII	13	CR	Carriage Feed	ctrl-M	Non Descript Position
ASCII	14	SO	Shift Out	ctrl-N	Non Descript Position
ASCII	15	SI	Shift In	ctrl-O	Non Descript Position
ASCII	16	DLE	Data Line Escape	ctrl-P	Non Descript Position
ASCII	17	DC1	Device Control 1	ctrl-Q	Non Descript Position
ASCII	18	DC2	Device Control 2	ctrl-R	Non Descript Position
ASCII	19	DC3	Device Control 3	ctrl-S	Non Descript Position
ASCII	20	DC4	Device Control 4	ctrl-T	Non Descript Position
ASCII	21	NAK	Neg Acknowledge	ctrl-U	Non Descript Position
ASCII	22	SYN	Synchronous Idel	ctrl-V	Non Descript Position
ASCII	23	ETB	End of Xmit Block	ctrl-W	Non Descript Position
ASCII	24	CAN	Cancel	ctrl-X	Non Descript Position
ASCII	25	EM	End of Medium	ctrl-Y	Non Descript Position
ASCII	26	SUB	Substitute	ctrl-Z	Non Descript Position
ASCII	27	ESC	Escape	ctrl-[	Non Descript Position
ASCII	28	FS	File Separator	ctrl-\	Non Descript Position
ASCII	29	GS	Group Separator	ctrl-]	Non Descript Position
ASCII	30	RS	Record Separator	ctrl-^	Non Descript Position
ASCII	31	US	Unit Separator	ctrl-_	Non Descript Position
ASCII	32	Space	Space		Non Descript Position
ASCII	33	!	Exclamation Mark	!	Non Descript Position
ASCII	34	"	Double Quote	"	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
ASCII	35	#	Hash	#	Non Descript Position
ASCII	36	$	Dollar	$	Non Descript Position
ASCII	37	%	Percent	%	Non Descript Position
ASCII	38	&	Ampersand	&	Non Descript Position
ASCII	39	'	Single Right Quote	'	Non Descript Position
ASCII	40	(	Open Round Brace	(	Non Descript Position
ASCII	41	)	Close Round Brace	)	Non Descript Position
ASCII	42	*	Kleene Star	*	Non Descript Position
ASCII	43	+	Addition Sign	+	Non Descript Position
ASCII	44	,	Comma	,	Non Descript Position
ASCII	45	-	Hyphen	-	Non Descript Position
ASCII	46	.	Period	.	Non Descript Position
ASCII	47	/	Forward Slash	/	Non Descript Position
ASCII	48	0	Zero	0	Non Descript Position
ASCII	49	1	One	1	Non Descript Position
ASCII	50	2	Two	2	Non Descript Position
ASCII	51	3	Three	3	Non Descript Position
ASCII	52	4	Four	4	Non Descript Position
ASCII	53	5	Five	5	Non Descript Position
ASCII	54	6	Six	6	Non Descript Position
ASCII	55	7	Seven	7	Non Descript Position
ASCII	56	8	Eight	8	Non Descript Position
ASCII	57	9	Nine	9	Non Descript Position
ASCII	58	:	Colon	:	Non Descript Position
ASCII	59	;	Semi-Colon	;	Nominal Value
ASCII	60	<	Backwards Arrow	<	Non Descript Position
ASCII	61	=	Equal Sign	=	Non Descript Position
ASCII	62	>	Forwards Arrow	>	Non Descript Position
ASCII	63	?	Question Mark	?	Non Descript Position
ASCII	64	@	At	@	Non Descript Position
ASCII	65	A	A	A	Non Descript Position
ASCII	66	B	B	B	Non Descript Position
ASCII	67	C	C	C	Non Descript Position
ASCII	68	D	D	D	Non Descript Position
ASCII	69	E	E	E	Non Descript Position
ASCII	70	F	F	F	Non Descript Position
ASCII	71	G	G	G	Non Descript Position
ASCII	72	H	H	H	Non Descript Position
ASCII	73	I	I	I	Non Descript Position
ASCII	74	J	J	J	Non Descript Position
ASCII	75	K	K	K	Non Descript Position
ASCII	76	L	L	L	Non Descript Position
ASCII	77	M	M	M	Non Descript Position
ASCII	78	N	N	N	Non Descript Position
ASCII	79	O	O	O	Non Descript Position
ASCII	80	P	P	P	Non Descript Position
ASCII	81	Q	Q	Q	Non Descript Position
ASCII	82	R	R	R	Non Descript Position
ASCII	83	S	S	S	Non Descript Position
ASCII	84	T	T	T	Non Descript Position

Datatype Name	Character ID	Name	Description	Character	Start/End Position
Continued from previous page…					
ASCII	85	U	U	U	Non Descript Position
ASCII	86	V	V	V	Non Descript Position
ASCII	87	W	W	W	Non Descript Position
ASCII	88	X	X	X	Non Descript Position
ASCII	89	Y	Y	Y	Non Descript Position
ASCII	90	Z	Z	Z	Non Descript Position
ASCII	91	[	Open Square Brace	[	Non Descript Position
ASCII	92	\	Backwards Slash	\	Non Descript Position
ASCII	93	]	Close Square Brace	]	Non Descript Position
ASCII	94	^	Hat	^	Non Descript Position
ASCII	95	_	Underscore	_	Non Descript Position
ASCII	96	`	Single Left Quote	`	Non Descript Position
ASCII	97	a	a	a	Non Descript Position
ASCII	98	b	b	b	Non Descript Position
ASCII	99	c	c	c	Non Descript Position
ASCII	100	d	d	d	Non Descript Position
ASCII	101	e	e	e	Non Descript Position
ASCII	102	f	f	f	Non Descript Position
ASCII	103	g	g	g	Non Descript Position
ASCII	104	h	h	h	Non Descript Position
ASCII	105	i	i	i	Non Descript Position
ASCII	106	j	j	j	Non Descript Position
ASCII	107	k	k	k	Non Descript Position
ASCII	108	l	l	l	Non Descript Position
ASCII	109	m	m	m	Non Descript Position
ASCII	110	n	n	n	Non Descript Position
ASCII	111	o	o	o	Non Descript Position
ASCII	112	p	p	p	Non Descript Position
ASCII	113	q	q	q	Non Descript Position
ASCII	114	r	r	r	Non Descript Position
ASCII	115	s	s	s	Non Descript Position
ASCII	116	t	t	t	Non Descript Position
ASCII	117	u	u	u	Non Descript Position
ASCII	118	v	v	v	Non Descript Position
ASCII	119	w	w	w	Non Descript Position
ASCII	120	x	x	x	Non Descript Position
ASCII	121	y	y	y	Non Descript Position
ASCII	122	z	z	z	Non Descript Position
ASCII	123	{	Open Curly Brace	{	Non Descript Position
ASCII	124	|	Bar	|	Non Descript Position
ASCII	125	}	Close Curly Brace	}	Last Character --
ASCII	126	~	Tilde	~	Last Character -
ASCII	127	DEL	Delete	DEL	Last Character

Appendix G. ASCII Table

The table below provides the character codes for each of the 128 characters of the ASCII table.

Table 8-46: The ASCII table.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	“	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	‘	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	´	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	11	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Decide control 1	49	31	1	81	51	Q	113	71	q
18	12	Decide control 2	50	32	2	82	52	R	114	72	r
19	13	Decide control 3	51	33	3	83	53	S	115	73	s
20	14	Decide control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	

Appendix H. Publications

The following papers were published at various national and international conferences, as part of the research and development that was carried out for this thesis. Although all papers have multiple authors, the content of those that contain definitions of the Atomic Rules approach, GQASV and SMT was solely produced by the author of this thesis.

On the Effectiveness of Mutation Analysis as a Black Box Testing Technique

Tafline Murnane
TATE Associates
Carlton Victoria
Australia
tmurnane@tate.com.au

Associate Professor Karl Reed
Department of Computer Science
and Computer Engineering
La Trobe University Australia
kreed@cs.latrobe.edu.au

Abstract

The technique of mutation testing, in which the effectiveness of tests is determined by creating variants of a program in which statements are mutated, is well known. Whilst of considerable theoretical interest the technique requires costly tools and is computationally expensive. Very large numbers of 'mutants' can be generated for even simple programs.

More recently it has been proposed that the concept be applied to specification based (black box) testing. The proposal is to generate test cases by systematically replacing data-items relevant to a particular part of a specification with a data-item relevant to another. If the specification is considered as generating a language that describes the set of valid inputs then the mutation process is intended to generate syntactically valid and invalid statements. Irrespective of their 'correctness' in terms of the specification, these can then be used to test a program in the usual (black box) manner.

For this approach to have practical value it must produce test cases that would not be generated by other popular black box test generation approaches. This paper reports a case study involving the application of mutation based black box testing to two programs of different types. Test cases were also generated using equivalence class testing and boundary value testing approaches. The test cases from each method were examined to judge the overlap and to assess the value of the additional cases generated. It was found that less than 20% of the mutation test cases for a data-vetting program were generated by the other two methods, as against 75% for a statistical analysis program. This paper analyses these results and suggests classes of specifications for which mutation based test-case generation may be effective.

1 Introduction

Testing software after it is completed remains an important aspect of software quality assurance despite the recent emphasis on the use of formal methods and 'defect-free' software development processes. As has been widely stated, testing does not prove the absence of errors. However, for some classes of programs it is possible in principle to define a 'safe' operational envelope based upon the set of test cases that it processes successfully [1]. Further, clients will frequently write contracts with acceptance testing clauses with the objective of verifying that the software does indeed perform as specified with the intention of taking legal action if it does not. Pre-delivery testing by developers can also provide critical data on the overall effectiveness of the development cycle by identifying residual fault rates.

Over time, a number of specification based (black box or prescriptive) test generation procedures have become popular and have been the subject of numerous studies as to their effectiveness. Broadly speaking, these provide a set of rules of varying detail and clarity that can be applied to a specification to generate test cases.

Traditional mutation analysis is a testing technique that was not originally intended for use with specification based testing. In traditional mutation analysis, a single fault is introduced into the program source code to create a new program version called a 'mutant.' Tests are created and are processed by the original and mutant programs with the goal of causing each mutant to fail (i.e. to produce output that differs from the non-mutant program). The effectiveness of the program test set is evaluated in terms of the number of mutants detected.

Budd and Gopal [2] found it was possible to apply the concept of mutation analysis to specification based testing. Rather than creating mutants from the program source code they are created by mutating the program specification.

In our proposal for mutation analysis, language elements (terminal elements) of the specification are used as mutation substitution elements. Each terminal element is systematically substituted for every other terminal element. A single element substitution produces one mutant specification. A mutation test set is then developed from the mutated specifications.

The goals of this research are:

1. to determine whether or not the mutant tests are able to detect errors in programs and if so, is there a class of specifications that would benefit from this type of testing and,

0-7695-1254-2/01 $10.00 © 2001 IEEE

2. whether this type of testing generates classes of tests that are not produced by other popular forms of black box testing and,
3. whether this type of testing produces small numbers of program-critical tests.

In the case study reported [3], the effectiveness of specification based mutation analysis was compared to boundary value analysis and equivalence class testing. In what follows, we summarise the case study and its results and make suggestions of the classes of programs for which this approach to testing would be effective.

2 Traditional Testing Techniques

2.1 Black Box Testing

The term 'black box' testing is used to describe tests that are derived primarily from a program's specification. In principle, the internal program source code is not considered. Test data derived from the specification is used to systematically test the input and output behaviour of the program. [4]. The goal is to generate a test set that fully exercises the program's functional requirements. Types of testing in this category include equivalence class testing, boundary value analysis, cause-effect graphing, error guessing, model checking and random testing.

2.2 Equivalence Class Testing

Equivalence class testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behavior. Test cases are designed to test the input or output domain partitions. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage [4]. The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed [1].

Jorgensen [5] identified one problem with equivalence partitioning. Often a specification does not define the output for an invalid equivalence class. Tucker [6] also noted that problems occur when the test data chosen for an equivalence class does not represent that partition in terms of the behaviour of the program function that is being tested.

Hamlet and Taylor [7] state that "Partition testing can be no better than the information that defines its sub-domains." If one input in an invalid equivalence class causes a failure in the program then all other inputs in that class must also cause a failure. If this is not the case then the equivalence class is not a good representative of that part of the program and thus the identification of additional partitions may be required. Due to the nature of this approach such problems may not be identified.

2.3 Boundary Value Analysis

Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the 'boundaries' of equivalence classes. Choices of boundary values include above, below and on the boundary of the class.

One disadvantage with boundary value analysis is that it is not as systematic as other prescriptive testing techniques. This is due to the fact that it requires the tester to identify the most extreme values inputs can take. Jorgensen noted that it is this type of abstract thinking that may allow a tester to improve the quality of the test sets used [5].

2.4 White Box Testing

White box testing involves the examination and testing of the program's internal composition. Test data is derived from examining the internal logic, branches and paths of the source code [4]. The goal is either to reach some coverage goal by testing and executing as many paths, branches and statements or other source characteristics as possible [8], or to ensure that certain expressions, decisions, branches, paths or source-attributes are exercised in particular a manner [9]. The number of source-attributes and coverage measures is language dependent and quite large (see for example Wu et al [10]).

White box and black box testing are complimentary and when used together can help to check whether a program conforms to its specification [6] (see for example Offutt and Liu [11][1]).

Rapps and Weyuker [12] noted that as the input domain of a program is generally very large, exhaustive testing is often impractical. Even for a small program containing a limited number of loops and branches, executing every statement is usually infeasible. They furthermore stated that ensuring all paths have been traversed does not guarantee that all errors in the code will have been detected, pointing for example to the problems in detecting 'def-use' errors. This view is supported by Weiser et al [9].

2.5 Traditional (Code-Based) Mutation Analysis

The main objective of traditional (code based) mutation analysis is to determine the effectiveness of a particular test suite. Faults are systematically introduced into the program's source code creating 'flawed clones' of the program called mutants. Each mutant has one language element in a single statement of the original program changed. The element substitution is based on a set of operators called 'mutation operators' [10].

A test case is designed for each mutant to try to detect the 'seeded' error. If the output from the mutated and non-mutated program under this test differs, then the test

[1] Offutt and Liu did state that functional testing had several advantages over structural testing.

has been successful in locating the mutant code and is assumed to be capable of locating similar errors. The mutant is 'killed' and is not executed again against other test cases [13]. Conversely, if the behaviors of the two programs are the same then the error was not detected and the test is discarded. New tests are then designed to try to detect the mutant code. In a complete mutation test all possible mutants of a particular program are produced and tested.

The mutation process may generate changes that leave the mutant functionally equivalent to the original program. This type of mutant should not be killed by any given test case which 'passes' testing the original program. The locating of these 'equivalent mutants' is usually done by hand.

The mutation score is the ratio of the number of killed mutants to the number of non-equivalent mutants and is the measure of the adequacy of the test set. Offutt and Lee stated that a test set is 'mutation-adequate' if the mutation score is 100% [14]. Generally, mutation scores of 90% are difficult to reach and scores over 95% are extremely difficult to achieve [11]. The ultimate goal of mutation analysis is to locate test cases which kill all non-equivalent mutants. Test sets which achieve this are referred to as "adequate relative to mutation" [13].

3 Specification Based Mutation Analysis

Specification based mutation analysis was first suggested by Budd and Gopal in 1984 [2]. Their approach involved mutating formal specifications whose language was defined using predicate calculus. Input test cases were generated by changing operators and predicates of the specification. More recent studies include the use of model checkers to automatically generate specification mutation test sets using several different types of mutation operators (see for example Black et al [16] [17] [18]).

In our case, we treat the specification as a language in which terminal sets can be mutated [3]. A specification can be characterised as a set of language elements which together describe the input and output behavior of a program, in much the same way as the syntax and semantics of the programming language determine valid forms of a program. Each data-item in the specification can be considered as a language or 'terminal' element. Collections of terminal elements are referred to as terminal sets. Production rules define how the terminal elements can be combined.

Substituting one terminal element for another creates one mutant specification. This process is repeated until every terminal element has been substituted for every other terminal element. Since each mutant contains one substituted element it can be referred to as a 'single-defect' mutant. 'Double-defect' mutants can be devised by substituting two terminal elements at a time. 'Production rule mutants' could also be created by mutating the production rules used to generate the input cases.

The 'mutation operator' substitutes one terminal element for another. A simple example is as follows. The terminal set $<terminal_1><terminal_2><termainal_3>$ could create the mutant $<terminal_2><terminal_2><termainal_3>$ by substituting the second terminal element for the first.

One test case is created from each mutant. Mutant test cases are classified as either a 'syntactically valid' or 'syntactically invalid' input. A syntactically valid input would make a program behave in a way that would be expected from a non-mutant input. In an input of this type, the terminal element that was substituted is 'syntactically equivalent' to the terminal element it replaced.

The syntactically invalid class of inputs can be decomposed into 'correct' and 'incorrect'. A syntactically invalid correct input is one that the program should and does recognise as containing a syntactic error. A syntactically invalid incorrect input is one that the program should recognise as containing a syntactic error but does not. This type of input may have located an inadequacy or fault in the program.

Creating a set of double-defect mutants could result in a more rigorous test set, as could production rule mutation. However the number test cases generated could be extremely large. Further, the consequences of the first mutation may directly interfere and complicate the implications of the second mutation, clouding the result of the test.

For some specifications, mutation analysis may produce a test set that appears to resemble a test set produced by random testing. The difference is that mutation analysis produces systematic test sets and is not dependent on randomisation by the tester.

One characteristic that is a requirement of this type of mutation analysis is that the specifications are written in a manner that facilitates the mutation process. It is apparent that some formal or semi-formal method is required where each terminal element is clearly defined. In the case study reported, the use of a semi-formal notation satisfied that requirement.

A shortcoming of mutation analysis is the cost involved in generating and executing test cases and examining the results. It is proposed that this testing technique would benefit greatly from automatic test case generation.

4 Previous Studies on Specification Based Mutation Analysis

Budd and Gopal's [2] approach to specification based mutation analysis involves producing specifications in predicate calculus based upon the predicate structure of the program under consideration. Their notation is chosen so that the input-output relationships are clear. In principle, the specification is mutated so that the new version contains an expression which if true, constitutes an illegal input. The expression should differ from its correct counterpart in that only one element is altered. Special steps are taken to deal with quantifiers, and relational operators may be mutated. An input test case is then produced which meets the mutated specification (i.e. makes it true).

Fabbri, Maldonado, Sugeta and Masiero [15] examined the use of mutation analysis to validate specifications presented as state charts, defining an appropriate mutation operator set to be taken as a fault model. A tool, Proteum/ST, was implemented to support the validation of finite state machine models. The goals of their research were to investigate ways of selecting useful test sets and how to ensure that a specification and its program had been thoroughly tested.

Black, Ammann and Majurski [16] experimented with using a (low-level language) model checker called 'Symbolic Model Version' or SMV to automatically generate complete specification based mutation test sets. "Complete" test sets include inputs and expected results. They used two types of mutation operators, creating both valid and invalid test sets. The model checker was used to produce counterexamples for each mutation operator, where each counterexample was a mutant of the original specification. They noted that their mutation operators were only useful for specifications that were described as finite models (within the context of a model checker). Branch coverage analysis was used to examine the usefulness of the test cases generated, finding that the tests were "quite good, but not perfect." The reported advantages of using a model checker for specification based mutation analysis was that the test case generation was completely automatic, as was the detection of equivalent mutants.

Ammann and Black [17] found that in order to make mutation analysis with a model checker possible they had to decompose specifications to lower language levels. They investigated a way of reducing larger state machines to sub-machines enabling these to be processed by model checkers. This reduction process was referred to as "finite focus." Since model checkers can handle finite state machines of no more than a few thousand states, the specification must allow decomposition. Thus the reduction of the specification's state machine allowed very large software systems to have test cases generated automatically. They proved that finite focus was a sound reduction technique, producing smaller state machines that were valid and creating a smaller mutation adequate test set.

Black, Okun and Yesha [18] examined a method involving the use of the SMV model checker to automatically generate complete mutation test sets from formal specifications using a predefined set of mutation operators. In order to perform the mutation testing, the specification had to be in a form that was readable by SMV. They focussed on redefining and comparing different types of operators and then reducing the number of mutation operators required for good test coverage. They presented classes of operators that provided different levels of coverage (up to 100%) and numbers of mutants created.

Black et al and Budd et al describe complex specification mutation schemes involving conditional logic which will inevitably be reflected in the processing programs. We consider that in many cases the practical advantages can be realised by merely permuting the input specification. Therefore our method of mutation analysis differs from these techniques in the following ways.

1. Only one mutation operator is required making the process far more simple and practical.
2. If the terminal elements are defined then the specification does not have to be changed to fit some predefined format.
3. The more complex the input specification the better the result of testing, for no increase in the complexity of the method.

5 The Case Studies and their Interpretation

The case study involved the comparison of boundary value analysis and equivalence partitioning to specification based mutation analysis [3]. The objective of this comparison was to examine the size and nature of the 'overlap' between the mutation analysis test set and the boundary value and equivalence class test sets. Two semi-formal specifications were used in this approach. Their syntax used a combination of COBOL or PL/I syntax and Backus-Naur Form notation. Both were programming assignments from Software Engineering subjects of La Trobe University (see [19] and [20]).

5.1 The Address Parser Specification

The first specification defines the input for an address parser (data-vetting) program. The input to this program is an address comprised of specific elements, shown in Figure 1. The aim of the program is to parse an address and if it is of a 'correct' format, write it to a file. If not the program is to report which elements of the address are incorrect. The symbols used in the specification are explained in Table 1, while the results of testing are outlined in Table 2. The complete specification included the requirement of directional indicators, for example the address 150 Main Road North Eltham 3095. In the interests of limiting the test set, this variant was not covered.

A standard address:
[{ UNIT }] ∧ddd∧ { , / } ∧ddd∧ <street>∧... <suburb>∧... <postcode>.
FLAT

A special flat/unit address:
[{ UNIT }] ∧ddd∧ <street>∧... <suburb>∧... <postcode>.
FLAT
RSD

A country or care-of address:
[{ C/- }] ∧... <street>∧... <suburb>∧... <postcode>.
C/o

Figure 1 Input elements of the address parser specification.

Table 1 Definition of specification notation. All other symbols are characters included as input.

Symbol	Actual Meaning
∧...	Represents one or more spaces.
d	Represents a digit.
<name>	Represents a character string.
{ }	Select one of the options contained within the braces.
[<name>]	<name> is optional.

Table 2 The results of testing the address parser program.

Number of Tests Created									%of Overlap with Mutation Analysis (MA)	
Mutation Analysis			Equivalence Class Testing (ECT)			Boundary Value Analysis (BVA)				
Total	Passed	Failed	Total	Passed	Failed	Total	Passed	Failed	ECT	BVA
290	10	280	29	10	19	89	31	58	14.5	17.9

A requirement of the address parser program was that if an invalid address was entered then the program has to be capable of recognising the 'incorrect' element(s) and output an appropriate message. This ability is illustrated by test cases one to three of Table 3, which shows sample data and the test methods capable of generating the test cases. The standard address is used as an example in this sample.

Conversely, the output generated by mutation test cases four and five highlight a program fault which was not found by boundary value or equivalence class testing. The fault is that the program produced an output message that did not correctly state which element of the address was incorrect. This illustrates that due to the extreme nature of some of the mutation tests generated, program faults were detected which were not found by conventional black box testing approaches.

Table 3 Sample test data and results of testing the address parser program.

#	Test Case	Program Output	Could be Generated by	Method of Generation
1	UNIT **3095** Main Road Eltham 3095.	Number has too many digits.	MA	Substitute postcode for unit number.
			ECT	Invalid class of unit number.
			BVA	Upper boundary of unit number.
2	UNIT **99** Main Road Eltham 3095.	Number has too few digits.	BVA	Lower boundary of unit number.
			ECT	Invalid class of unit number.
3	UNIT **Test** Main Road Eltham 3095.	Number has too few digits.	ECT	Invalid class of unit number.
4	UNIT **C/o** Main Road Eltham 3095.	Number has too few digits. Space required after suburb. Street not found. Invalid suburb. Full stop not found.	MA	"Care-of" address identifier substituted into the unit number.
5	UNIT 100 **C/o** Eltham 3095.	Space required after suburb. Street not found. Full stop not found. Invalid suburb.	MA	"Care-of" address identifier substituted into the street name.

5.2 The Statistical Analysis Specification

The second specification defines the input of a statistical analysis program which computes the standard deviation and average of values that are tagged by a one-letter identifier. The elements of this specification are listed in Figure 2. The overall results of testing this specification are shown in Table 4. Sample test data and results are shown in Table 5.

Batches of these letters and values are bracketed with the following records.
sbatch∧...<batchno><eor>
and
ebatch∧...<batchno><eor>

The last record in any collection of batches is:
lbatch∧...<eor>|lbatch...<eof>

The records in each batch are of the following form:
<record>::=<lpart><rpart><eor>
<lpart>::=<null>|∧...
<rpart>::=<letter>∧<value>|<rpart>∧<letter>∧<value>
<letter>::= any letter chosen from the set [B-L, S-W, Z]
<value>::= any valid, non-floating point decimal value in the range [-99, 99]

Figure 2 Input elements of the statistical analysis specification.

Table 4 The results of testing the statistical analysis program.

Number of Tests Created									% of Overlap with Mutation Analysis	
Mutation Analysis			Equivalence Class Testing (ECT)			Boundary Value Analysis (BVA)				
Total	Passed	Failed	Total	Passed	Failed	Total	Passed	Failed	ECT	BVA
104	68	36	25	10	15	30	12	18	76	76

Table 5 Sample test data and results of testing the statistical analysis program.

#	Test Case	Program Behaviour	Could be Generated by	Method of Generation
1	sbatch **20** G –**99** ebatch 20	Program accepts the input as valid.	MA	Substitute sbatch number for ebatch number.
			ECT	Valid class of rpart number.
			BVA	Upper boundary of rpart number.
3	sbatch **sbatch** G –99 ebatch 20	Program outputs error message stating that there was no sbatch number found.	MT	sbatch tag substituted into the sbatch number.
4	sbatch **ebatch** G –99 ebatch 20	Program outputs error message stating that there was no sbatch number and the rpart and ebatch tag was not found.	MT	ebatch tag substituted into sbatch number.

5.3 An Examination of the Results

In the specification for the address parser, few terminal elements were syntactically equivalent. Consequently, from the mutation test set produced the program found only a small number of addresses that were syntactically valid. For example the house number could be substituted for the unit/flat number without an error being raised, as both were three digits long. However if the house or unit/flat numbers were swapped with a text sentence such as the street name the program found a syntactic error in the input. The element that was the most interchangeable was the 'space.' One-space markers could be swapped for any one-or-more space markers without errors being detected in the input. The reverse was not equivalent, however the space marker could also be replaced for elements such as all of the optional address elements.

It was found that there was 17.93% equivalence between the mutation analysis and the boundary value analysis test sets, and 14.48% between the mutation analysis and the equivalence class test sets.

For the statistical program there was an extensive overlap between the mutation analysis test set and the boundary value and equivalence class test sets. For example, all three testing methods located errors in inputs involving a missing sbatch or ebatch tag and in inputs containing a letter or value outside of the specified range. Another type of test that produced equivalencies was the replacement of an element with the <null> element. When replacing with the <null> element, the three test sets produced equivalent results in most situations. Therefore there was a large overlap in the tests from the three methodologies.

A 75.96% equivalence was found between the mutation analysis test set and the boundary value and equivalence class test sets.

The testing process showed that although the programs were returning error messages when invalid inputs were entered, in many cases they were not correctly stating which section of the input contained the error. For the statistical program this inadequacy was located by all three testing methodologies. However, for the address parser program most mutation test cases were able to detect these types of errors, whereas the majority of the boundary value and equivalence class tests did not.

6 Mutation Testing Amenable Specifications

In the results reported in the previous section, the address parser specification produced a mutation test set in which there was a modest overlap with the boundary value analysis and equivalence class test sets (less than 20%), while the statistical program's specification produced a substantial overlap in the test sets (75%).

A closer examination of the two specifications suggests that some specifications will be more amenable to mutation based testing than others. While this issue is the subject of future work, we can make some informal comments that will be of practical guidance to practitioners. Consider a simple specification of the following form.

$\langle terminal_1\rangle\langle sep_1\rangle\langle terminal_2\rangle\langle sep_2\rangle\langle terminal_3\rangle$

where each of the $\langle sep_i\rangle = \{s_{i_1},\ldots,s_{i_u}\}$
and each of the $\langle terminal_j\rangle = \{t_{j_1},\ldots,t_{j_m}\}$

In general, the nature of the $s_{i_j} \in \langle sep_i\rangle$ and the $t_{j_k} \in \langle terminal_j\rangle$ will be such that it would be unlikely that substituting some arbitrary $t_{3_k} \in \langle terminal_s\rangle$ for $\langle sep_3\rangle$ would produce a test case that would have been generated by either equivalence class testing or boundary value analysis. However, we also need to consider the case of

substituting $t_{3_k} \varepsilon$ **<terminal$_s$>** for **<terminal$_1$>**, which would be a valid mutation operation.

Constructing an equivalence class test requires that there be some basis for dividing the terminal sets (or combinations of them) to construct equivalence partitions. We then choose one element from the partition as a test case. If for some reason the intersection of the terminal sets is non-null then we may have constructed a mutation test by default. However if the terminal sets are distinct then by definition, a valid equivalence class test cannot choose an element from another terminal set. Whether or not invalid equivalence class tests will generate cross-terminal set substitutions depends upon how the terminal set is extended to include illegal values.

In the case of boundary value tests, we point out that if the sets are discrete and finite then the concept of boundaries may have no practical meaning. If they are in some sense continuous or are in a sequence, then boundary values may be considered to exist. Alternatively the boundary values may be stated explicitly. A typical specification for such a terminal might be (without loss of generality) as follows.

<terminal$_i$> ::= $\{R \varepsilon N : ub_i \leq R \leq lb_i\}$

where ub and lb are upper and lower boundaries respectively.

In this case, if there are multiple terminal sets with this definition and their intersection is non-null, then both mutation analysis and boundary value and equivalence class testing can generate test cases that will be identical.

7 Conclusions and Future Work

While specification based mutation analysis can provide a tester with valuable information about the correctness of program behaviour, it is clear that it would not benefit all types of specifications. Future work will include an examination of the feasibility of identifying specifications that will benefit from mutation analysis and the development of mutation operators. Empirical experimentation will determine whether there is a statistical overlap between specification based mutation analysis and other popular forms of black box testing. An additional goal is to investigate whether specification based mutation analysis is effective at producing program-critical tests.

It is also clear that given an appropriate set of (formal) production rules that specify a program's input, a test case generator can be constructed using standard compiler writing techniques. It would then be possible, given appropriate mutation operators, to generate mutant test cases automatically. The simple substitution operator used in the test cases would be straightforward.

Finally, the authors recognise that the approach taken here has properties similar to random test case generation and might generally be regarded as a particular case of this approach.

Acknowledgements

We would like to acknowledge the contributions to this paper of Mr. John Murnane of the Department of Science and Mathematics Education, University of Melbourne, Australia.

We also acknowledge the support of the Department of Computer Science and Computer Engineering at La Trobe University, including the work of Mark Santos, whose use of this technique in a programming assignment lead to our formalisations and to the case study reported here.

Finally, we would like to acknowledge the support of TATE Associates.

References

[1] Karl Reed. *Software Reliability, Testing and Security Class Lecture Notes*. CSE31STM, subject of the Department of Computer Science and Computer Engineering, La Trobe University, Australia, 1998.

[2] Timothy A. Budd, Ajei S. Gopal. Program Testing by Specification Mutation. *Computer Language, vol. 10, no. 1*, Great Britain, 1985, pp. 63-73.

[3] Tafline Murnane. *The Application of Mutation Techniques to Specification Testing*. Honours Thesis, Department of Computer Science and Computer Engineering, La Trobe University, Australia, 1999.

[4] Glenford Myers. *The Art of Software Testing*. Wiley-Interscience Publication, 1979.

[5] Paul Jorgesen. Software Testing: A Craftsman's Approach. *Department of Computer Science and Information Systems, Grand State University Allendale, Michigan and Software Paradigms,* Rockford, Michigan, CRC Press 1995.

[6] Allen Tucker, Robert Cupper, W. Bradley, Richard Epstein, Charles Kelemen. *Fundamentals of Computing II. Abstractions, Data Structures, and Large Software Systems*. McGraw-Hill Inc, 1995.

[7] D. Hamlet, R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, vol. 16, no. 12, December 1990, pp. 1402 – 1411.

[8] Michael Dyer. *The Cleanroom Approach to Quality Software Development*. John Wiley & Sons Inc, Canada, 1992.

[9] M. D. Weiser, J. D. Gannon, and P. R. McMullin. Comparison of Structural Test Coverage Metrics *IEEE Software*, March 1985, Pages 80 – 85.

[10] Basili Wu and Karl Reed. A Structure Coverage Tool for ADA Software Systems. *Proceedings of the Joint Ada Conference*, Washington, D.C. (WADAS) March 1987.

[11] A. Offutt, S. Liu. *Generating Test Data from SOFL Specifications*. Preliminary draft yet to be published, written April 1997.

[12] Sandra Rapps, Elaine J Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, vol. SE-11 no. 4 April 1985.

[13] A. Offutt, J. Voas. Subsumption of Conditional Coverage Techniques by Mutation Testing. *Technical Report ISSE-TR-96-01,* January 1996.

[14] A. Offutt, Stephan Lee. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, vol. 20, no. 5, May 1994.

[15] S.C.P.F. Fabbri, J.C. Maldonado, J.C. Sugeta and P.C. Masiero. Mutation Testing Applied to Validate Specifications Based on Statecharts. *10th International Symposium on Software Reliability Engineering, Proceedings Los Alamitos: IEEE Computer Society*, Boca Raton, USA, pp. 210-219.

[16] Paul Ammann, Paul Black, William Majurski. Using Model Checking to Generate Tests from Specifications. *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods* Brisbane Australia, December 1998, pp 46-54.

[17] Paul Ammann, Paul Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. *Proceedings of the 18th Digital Avionics System Conference*, St. Louis Missouri, October 1999. IEEE vol. 2, section 10.4.6, pp 1-10.

[18] Paul Black, Vadim Okun, Yaacov Yesha. Mutation Operators for Specifications. *15th Annual Software Engineering Conference*, IEEE Computer Society, Grenoble, France, September 2000, pp. 81-88.

[19] Karl Reed. *CSE31STM Assignment Two.* CSE31STM, subject of Department of Computer Science and Computer Engineering, Latrobe University, Australia 1998.

[20] Karl Reed. *CSE32SRT Assignment Two.* CSE32SRT, subject of Department of Computer Science and Computer Engineering, Latrobe University, Australia 1998.

A Preliminary Survey on Software Testing Practices in Australia[1]

S.P. Ng[*], T. Murnane[†], K. Reed[†], D. Grant[*], T.Y. Chen[*]

[†]*School of Engineering & Mathematical Science*
La Trobe University
Kingsbury Drive, Bundoora 3086
Australia
Email: {t.murnane, k.reed}@latrobe.edu.au
Phone: (+613) 9479 1377
Fax: (+613) 9479 3060

[*]*School of Information Technology*
Swinburne University of Technology
John Street, Hawthorn 3122
Australia
Email: {sng, dgrant, tchen}@it.swin.edu.au
Phone: (+613) 9214 5505
Fax: (+613) 9819 0823

Abstract

This paper presents the findings of, to the best of our knowledge, the first survey on software testing practices carried out in Australian ICT industry. A total of 65 organizations from various major capital cities in Australia participated in the survey, which was conducted between 2002 and 2003.

The survey focused on five major aspects of software testing, namely testing methodologies and techniques, automated testing tools, software testing metrics, testing standards, and software testing training and education. Based on the survey results, current practices in software testing are reported, as well as some observations and recommendations for the future of software testing in Australia for industry and academia.

Keywords: *Software engineering, software testing, survey*

1. Introduction

Swinburne University of Technology, in conjunction with La Trobe University and sponsored by the Australian Computer Society conducted a survey on software testing in Australia between 2002 and 2003. Similar surveys are being run in several other Southeast Asian countries. Software development organizations from different industry sectors (government, pubic and private), domestic and foreign owned, in-house groups and software companies across various industries were invited to participate in the survey.

There were a number of reasons for conducting this survey:-

Firstly, anecdotal evidence from software developers suggests that testing is becoming an increasing percentage of the development budget.

Secondly, the authors' view is that software quality will become an increasingly important factor in software marketing. As this evolves, testing strategies will (in our view) become progressively more important. A carefully constructed survey has the potential of identifying the best practices, which can then be disseminated.

Thirdly, the survey may provide indications of future research directions.

Fourthly, the comparison with parallel surveys in the region will assist all national industries to both improve software quality and identify optimum testing strategies.

Finally, the results will provide guidance for those training software developers and software engineers.

The observations reported in this paper were based on 65 respondents successfully completing the questionnaire. Interestingly, the results from analyzing these 65 responses follow almost the same trends obtained from an earlier analysis performed three months ago using the first 41 responses. Despite the relatively small sample population in the survey, the consistency of the data obtained heightened our confidence to report the observations in this paper.

The remainder of this paper is structured as follows. Section 2 explains the methods that were used to plan and conduct the survey, including the method of selecting a research sample, the variables that the survey aimed to measure, the approaches used to invite subjects, and the methods of collecting data from respondents. Section 3 reports and discusses the results of the survey, including organization information of the respondents. Section 4 analyses and summarizes the survey findings, and discusses the implications of the survey on the software testing industry, as well as its implications on training and

[1] This Survey was funded by the Australian Computer Society under its research program.

education of software testing personnel, both in the workplace and at universities. Section 5 concludes the paper and suggests future work.

2. Survey Methodology

2.1 Survey Objectives

Two of the five objectives listed in the introduction were used as the design objectives for this survey, since the others are considered as outcomes that flow from these. The primary objective was to determine the types of testing techniques, tools, metrics and standards that organizations in Australia use when carrying out software testing activities (this of course embraces several of the listed objectives). The purpose of this was to provide a concise picture of the current industry best practices.

The second objective was to determine whether existing training courses in software testing taught in the workplace or in similar study at tertiary institutes adequately cover the types of testing methodologies and skills that industry requires. If these requirements were not met, the industry may benefit from the survey recommendations to address any deficiency observed and ultimately improve the existing training opportunities available to practitioners as well as novice testers.

Based on these two objectives, a number of hypotheses were employed to design the questionnaire and shape the direction of the survey.

2.2 Survey Description

The survey targeted senior employees involved with testing in software development organizations. Requests were addressed to software testing or project mangers as the personnel most likely to understand their testing environments and experiences within their organization.

Five major areas of software testing related activities were investigated by the survey. In addition, an introductory section was also included to assess the organization size and structure, and where relevant, history of the organization and its overall procedures with respect to software development and testing. Using the conjectures in our hypotheses as means of constructing specific questions, the questionnaire was arranged into the following six sections. The information sought can be summarized as follows.

Section A - Organization Information

This section captured the type and size of the organization, including specifics such as the current number of general employees and IT professionals, the number of applications developed and tested over the past three years, the allocated and actual budget for testing among the other various software development activities, as well as questions relating to whether the organization wrote specifications and whether changes to specifications were controlled and tracked.

Section B - Software Testing Methodologies and Techniques

The extent to which software testing methodologies and general testing techniques are used in the industry and the current practices of those organizations adopting structured methodologies and techniques in software testing were investigated in this section.

Section C - Automated Software Testing Tools

Questions relating to the extent to which automated testing tools are used in industry, including commercial and in-house developed tools, were placed in this section revealed. The level of satisfaction with such tools was assessed by querying the respondents' belief that the quality of developed software was being improved by the use of such tools.

Section D - Software Testing Metrics

This section explored the extent to which software testing metrics are used by industry, and if and how those metrics are improving the quality of software under development.

Section E - Software Testing Standards

The usage of standards for software testing in industry, including published standards such as ISO, CMM and their quality accreditation, as well as in-house developed standards was assessed in this section. Questions were posed to determine whether the use of standards was considered to improve the software development processes of the organization.

Section F - Software Testing Training and Education

This section determined the extent to which organizations provide training in software testing for their employees. Also examined was the organization's view on the factors that attract software testing staff to attend training courses as well as the benefits for testing staff that accrue. The usage of various sources of training courses (such as universities or TAFE colleges, external commercial training courses, in-house training and self-study) were also queried.

2.3 Survey Method

A questionnaire comprised of both closed and open-type questions was used. Survey interviews were conducted face to face, over the telephone, via facsimile or email attachment. To allow for more flexible arrangements, some respondents were invited to complete the online questionnaire at our survey web site[2]. In all cases, printed or verbal explanatory notes were provided to respondents to ensure consistent interpretation of the terminologies and questions in the questionnaire. In general, respondents took no longer than thirty minutes to complete the questionnaire. Confidentiality and privacy were assured to all individuals returning the questionnaire and the organization that they represented.

2.4 Sample Selection

Our survey targeted the population at the organizational level (or alternatively at departmental level if there was more than one department in an organization responsible for software development). A draft questionnaire of the survey was trialed against a small group of five organizations, and a number of adjustments were made based on the experiences and feedback we gathered from the pilot run. As a result, we aimed at targeting four different types of participants in this survey. The first preference was test managers, the second was a member of the test team, thirdly a software project manager, and finally a general organizational or departmental manager. This allowed us to deal with situations where there was no specific individual responsible for testing in the organization.

Five approaches to our target audiences were made over a twelve month period to identify a suitable sample for the survey. Resources used were:- an article which appeared in the May 2002 issue of Australian Computer Society (ACS) magazine, *Information Age*, reaching around 14,000 Australian IT professionals [10]; a one-page insert in the February/March 2003 issue of *Information Age*; a list of 350 companies constructed from Australia's national telephone directories; a list of software test-likely organizations from classified post advertisements appeared in a newspaper[3]; and a flyer to request for participation enclosed in the June 2003 issue of the *Software* magazine published by Software Engineering Australia (SEA) with circulation of over 6,000 copies distributed to its members nationally.

As a result, a total of 65 individuals or companies participated in the survey. This is a relatively low response rate, given the large number of organizations that were invited to participate in the survey, and the large estimated size of the population.

During the pilot study of the survey, a "focus groups" sampling method was used, in which we personally invited companies that survey members had connections with to participate.

This survey sample was then built in three stages. In the first round, non-probabilistic sampling called "convenience sampling" [4] was employed, where the participants were selected because they were easy to access or because we believed they had a good chance of representing the population. In the mail out stage, "cluster based sampling" [4] was adopted, in which the target population was filtered using an indicator that was deemed likely to classify them as not being a software test-likely organization. Companies which had shop fronts and software/hardware sales companies were considered unlikely to be software development organizations and hence were unlikely to be performing any software testing. Nevertheless, the response rates in all data-collection stages of the project were far below our expected target of 100 responses or more, although based on our conversations with other researchers, this reflects the experience of others in Australia attempting to gather similar information in different disciplines.

The relevance of the sample was, however, considered to be extremely important, in that over 70% of respondents had managerial or team leadership roles in their organization and we are satisfied that the results from the sample are likely to be "indicative", although may not be absolutely conclusive. In particular, the results show the attributes of those responding organizations, regardless of whether they perform software testing in an ad hoc or a systematic manner.

Discussions among university colleagues have suggested that the low response rate may indicate that a large number of software development groups do not use any vigorous testing methods. It is also possible that the Australian software developers, similar to their New Zealand counterparts [3], are "survey averse", and that the cost of attaining representative samples is beyond the scope of our current project budget. Nevertheless, we intend to investigate the reasons why practitioners were reluctant to participate in the survey as part of our follow-up activities of the project.

3. Survey Results

3.1 Organization Information

[2] URL of the software testing survey web site is http://acssesurvey.it.swin.edu.au

[3] This task is simplified by the fact the largest circulation newspapers run extensive IT supplements on Tuesdays of each week.

Of the 65 organizations responded to our survey, more than two-thirds (67.7%) belong to the local private commercial sector. In addition to these, 15.4% were from overseas-based private commercial organizations, 10.8% were from government, and 6.2% were public non-commercial organizations (Table I).

Table I - Respondents by sector

Sector Type	Response	%
Government	7	10.8
Public non-commercial organization	4	6.2
Local private commercial organization	44	67.7
Overseas-based private commercial organization	10	15.4
Joint venture between public and private sectors	0	0
Total	65	100.0

The majority industry type of the respondent organizations was software house and IT consultancies (49.2%). Other industries included finance and insurance, manufacturing and engineering, research and development, and telecommunications (Table II).

Table II - Respondents by industry

Industry Type	Response	%
Banking, finance & insurance	7	10.8
Education & training	1	1.5
Hotel, tourism, retail & trading	2	3.1
Manufacturing & engineering	4	6.2
Research & development	3	4.6
Software house & IT consultancy	32	49.2
Telecommunications	3	4.6
Other	13	20.0
Total	65	100.0

The 65 organizations ranged from large in size with over 500 employees (24.6% of organizations) to very small sizes of less than 20 (also 24.6%). Most of these had substantial experience in software development: 13 organizations claimed to have 6 to 10 years of relevant software testing experience, 19 organizations with 11 to 19 years, and 22 organizations have more than 20 years of experience. Again, although the survey sample size was not ideal, we believe that these 65 organizations of such diversities do provide us a valid set of sample data and allow us to reflect the current software testing practices in the country.

As our main interests are in software testing, our questions mainly focused on software testing issues, including budget allocation. We found that only 3 out of 65 organizations allocated 40% or more of the total development budget to software testing in the initial software development plan, while 49 organizations had allocated less than 40% of the budget to testing. Among these 49 organizations, most of them (16 each) allocated between 10 to 19% or between 20 to 29% of the initial budget to testing alone. Nine organizations allocated between 30 to 39%, and amazingly there were 8 organizations which allocated less than 10% of the total development budget to software testing during the planning phase. Despite these facts, only 11 organizations (16.9%) reported that they met their testing budget estimates. Twenty-seven organizations (41.5%) spent 1.5 times of the estimated cost in testing and 10 organizations (15.4%) even reported a ratio of actual to estimated testing cost of 2 or above. Even more surprisingly, there were 3 organizations (4.6%) which managed to complete testing activities using only half of their initial allocated testing budget.

3.2 External Consultants, Testing Responsibility and Organizational Issues

We were surprised to find that, in the past 3 years, 24 organizations (36.9%) hired external testers to assist the organization to implement software testing methods or tools. Of these, 50% outsourced less than 20% of the testing budget to external testers, and 29.2% outsourced between 20 to 39%. In terms of satisfaction level, 75% were either satisfied or highly satisfied with the service from external testers and another 16.7% were neutral. Only 1 organization (4.2%) was dissatisfied and one other was highly dissatisfied. These figures clearly indicate that current external software testing companies are providing a very high standard of services to their clients in Australia.

The majority of respondents (70.8%) were found to appoint a person who is solely responsible for managing software testing activities in their organization, showing that testing is becoming a more independent process in industry.

User acceptance testing and regression testing were extremely common for all software applications developed, the results being 31 (47.7%) and 45 organizations (69.2%) respectively. Of the 45 organizations performing regression testing, 24 of them (53.3%) repeated regression testing for every new version of the application whilst 13 organizations (28.9%) conducted regression testing again after every change in the application.

Another interesting finding was that 50 out of 65 organizations (76.9%) followed formal processes or procedures for approving changes in requirements and specifications during the software development lifecycle. There were also 50 out of 65 organizations that formally documented requirement and specification changes during system development. In other words, the remaining 15 organizations (23%) did not formally document these changes at all. A closer scrutiny of the raw survey data indicated that there was no significant correlation between the 50 organizations in which formal processes were followed to approve

requirement changes and those 50 organizations in which requirement changes were formally documented during system development. This observation reveals the existence of some degrees of inconsistencies and weaknesses within the software development practices in industry.

3.3 Software Testing Methodologies and Techniques

This section investigated the extent of adoption of software testing methodologies and techniques in organizations to improve the quality of their software products. Forty-two out of 65 organizations (64.6%) claimed the use of at least one structured software testing methodology in the past 3 years. While it is encouraging to see that almost two-thirds of the respondents employ some structured testing methodologies, the fact that slightly more than one-third of the organizations are still doing ad-hoc testing was quite remarkable. In fact, we imagine that the actual figures for ad-hoc testing may be even underestimated, as many such organizations may be reluctant or may not have been interested in responding to our survey.

The three most popular methodologies included test case selection, static analysis and dynamic analysis. In terms of selecting test cases, black-box testing (particularly boundary value analysis and random testing) were more common than white-box testing (29 responses for black-box versus 16 for white-box). Eighteen respondents adopted data flow analysis techniques. Only 3 organizations reported the use of mutation analysis and none reportedly use symbolic analysis. Although the unpopularity of such techniques may not be conclusive due to the small sample size in the survey, it is evident that these techniques are rarely used in the industry despite large volume of research work has recently been done in these areas [1, 2, 6, 7]. Comparing static analysis and dynamic analysis, we observed that document and code inspection attracted a slightly higher response rate than code walkthroughs (29 versus 22). In both cases, manual processes were still more commonly engaged than automated ones (. The use of automated tools in software testing will be further discussed in later sections.

Of the 42 organizations using some form of structured testing methodology in the past three years, 27 (64.3%) carried out structured testing for more than 80% of their projects, and 21 organizations (50%) have been adopting structured testing methodologies for over 5 years. While 10 respondents (23.8%) were unsure if the cost-effectiveness has been improved by the use of methodologies, 28 (66.7%) expressed their affirmative responses, in contrast to only 4 respondents (9.5%) who expressed their disappointment in adopting testing methodologies. It would be interesting to further investigate the reasons why there exists such a large percentage of respondents who were unsure about the effects of utilizing systematic testing approaches.

Major testing activities performed by respondents (in order of popularity) were designing test cases (55 organizations), documenting test results (54 organizations), re-using the same test cases after changes were made to the software (also 54 organizations), defining test objectives (48 organizations) and re-designing test cases based on the analysis of previous test results (41 organizations). We observed that although some organizations did not claim to use structured testing methodologies, they did perform basic testing activities such as designing test cases and documenting test results on a regular basis.

Among the 56 organizations (86.2%) that used standard test plan templates, 18 of them (32.1%) *always* updated the test plan whenever there was a change in requirements and specifications. While 22 (39.3%) and 12 (21.4%) organizations respectively *quite often* and *occasionally* updated their test plans, surprisingly, 4 organizations (7.1%) *never* update their test plans, even when requirements and specifications changes occur. This survey result suggests that some organizations still may not be practicing the proper procedures of continuously updating test plans even though this process is generally regarded as essential to guarantee the validity and efficiency of test plans.

There were 59 out of 65 organizations (90.8%) reporting that formal tests were performed to ensure the developed software meets its requirements and specifications, suggesting that user acceptance testing is widely used in industry. Twenty-five organizations (38.5%) reported that over 80% of their test cases generated in the past 3 years were derived from specifications, with 17 organizations (26.2%) reporting between 60 and 79%. Regarding the percentage of software faults detected in the past 3 years, 22 organizations (33.8%) found that between 40 to 59 % of such faults were related to specification defects, followed by 16 (24.6%) and 15 organizations (23.1%) falling within the range of 20 to 39% and 0 to 19% respectively.

As expected, “big bang” was the most popular integration testing approach (used by 33 organizations) probably due to its simplicity. This was followed by bottom-up and top-down approaches, which were used by 27 and 23 organizations respectively.

Pre-defined criteria were used by 48 respondents (73.8%) to stop testing of a software system. Face to face interviews revealed that several organizations still

adopt the common practice of ceasing testing once resources are exhausted, irrespective of possible number of faults that may remain in the software. Another common trend was to cease testing as soon as all "critical" or "show-stopper" faults had been detected and removed, despite the fact that those methods used to determine whether all such faults had been removed were, in most cases, neither formal nor methodological in nature.

Being software testing researchers, we were particularly interested in practitioner's views on the barriers to adopting testing methodologies in their workplace. The responses from the survey were summarized in Table III.

Table III - Barriers to adoption of testing methodology

Barrier	Response	Rank
Do not think there is any barrier	20	2
Lack of expertise	**28**	**1**
Lack of support tools	18	
Costly to use	14	
Difficult to use	3	
Time-consuming to use	20	2
Do not think it is useful or cost effective	5	
Do not know of any testing methodology	7	
Other	14	

As indicated in Table III, 43.1% reported that a lack of expertise as the dominant factor preventing or disadvantaging organizations from using software testing methodologies. About 30% of respondents did not believe there was any barrier to using methodologies in their organizations. On the other hand, the same number of respondents regarded testing methodologies as being time-consuming when used.

The largest problem reported with using testing methodologies was a lack of expertise, with almost half of the respondents encountered. In our opinion, there are two likely causes of this. Firstly, this could indicate that software testing professionals are not sufficiently trained in testing methodologies either at the university or industry level. The second cause may be that there is a genuine shortage of software testing professionals with such knowledge in industry. In either case, it is obvious that training opportunities of software testers are essential to improve the quality and reliability of the software products developed in the country.

3.4 Automated Software Testing Tools

There was substantial usage of automated software testing tools amongst the respondents. In the past 3 years, 44 organizations (67.7%) have automated some of their testing activities. Out of these 44 organizations, 30 (68.2%) acquired the tools by purchasing existing commercial products, while only 6 (13.6%) developed their own tools. Quite unexpectedly, we found that only 1 organization (2.3%) out-sourced development of their testing tools.

Among these 44 organizations, automated testing tools for test execution (35 organizations or 79.5%), regression testing (33 organizations or 75%), and test results analysis and reporting (27 organizations or 61%) ranked the top three positions for automated testing activities. Other activities such as generating test cases/scripts and test planning/management also attracted more than one-third of the respondents (20 and 17 organizations respectively). A large proportion of respondents (36 organizations or 81.8%) in fact employed multiple automated techniques in software testing.

Although it is widely believed that software quality will be improved by the use of automated testing, only 30 of the 44 respondents (68.2%) using testing tools agreed with this belief. Ten organizations (22.7%) were unsure, and 4 organizations (9.1%) gave a negative response to this question.

About half (32) of the 65 respondents reported that cost was a major barrier to using automated tools for software testing in their organizations. There were 26 and 16 respondents respectively regarding time and difficulties as factors which prevented them from using testing tools in their organizations. The actual response figures were presented in Table IV.

Table IV - Barriers to adoption of testing tools

Barrier	Response	Rank
Do not think there is any barrier	9	
Costly to use	**32**	**1**
Difficult to use	16	3
Time-consuming to use	26	2
Do not think it is useful	4	
Do not think it is cost-effective	9	
No information resource available	1	
Do not know of any software testing tool	4	
Other	28	

3.5 Software Testing Metrics

Out of the 65 survey respondents, only 38 (58.5%) used measurable test objectives. Not surprisingly, the most popular metric reported was defect count (used by 31 organizations), probably due to its simplicity.

It is encouraging to see that 19 (50%) of the 38 organizations using metrics applied them to more than 80% of the software applications developed in the past 3 years. However, only 21 organizations (55.3%) agreed that the quality of the developed software applications was improved by the use of the metrics. Thirteen organizations (34.2%) were unsure, and 4 organizations (10.5%) even disagreed about the

positive effect of metrics on software quality. This counter-intuitive result certainly deserves further investigation as follow-up activities of the project.

As shown in Table V, about 30% of the participants (20 organizations) reported no barrier or disadvantage in the use of metrics. On the other hand, there was about a quarter of respondents (17 organizations) who found the use of metrics to be too time-consuming.

Table V - Barriers to adoption of testing metrics

Barrier	Response	Ranking
Do not think there is any barrier	**20**	**1**
Costly to use	4	
Difficult to use	4	
Time-consuming to use	17	2
Do not think it is useful	5	
Do not think it is cost-effective	2	
No information resource available	6	
Do not know of any software testing metrics	3	
Other	22	

3.6 Software Testing Standards

Software testing standards were being adopted by 47 out of 65 respondents (72.3%). In-house developed standards were employed by 29 organizations, while 18 organizations used a combination of published and in-house standards for software testing. Nevertheless, there were only 3 organizations relying solely on published standards, indicating that those standards known to software developers were possibly quite deficient. On the whole, 39 (83%) of the 47 organizations agreed that such standards did improve the software development processes used in their organization, none disagreed, and 7 were unsure (14.9%)[4].

Of the 65 organizations responded to the survey, 22 (33.8%) were quality accredited for their software development processes. Interestingly, out of these 22 accredited organizations, only 15 (68.2%) believed that their software development processes were being improved by acquiring the accreditation, while 4 (18.2%) did not think so and another 3 (13.6%) were not sure.

Table VI summarizes the respondents' views on the barriers to adopting software testing standards in their organizations. The majority of them (28 organizations) thought that there was no barrier. There were also significant numbers of responses indicating that time (15 organizations) and cost (13 organizations) are the other two main deterrents to the use of testing standards.

[4] One survey participant had mistakenly left this response blank, thus the total percentage in this category does not add up to 100.

Table VI - Barriers to adoption of standards

Barrier	Response	Rank
Do not think there is any barrier	**28**	**1**
Costly to use	13	3
Difficult to use	4	
Time-consuming to use	15	2
Do not think it is useful	6	
Do not think it is cost-effective	5	
No information resource available	3	
Do not know of any software testing standards	4	
Other	18	

3.7 Software Testing Training and Education

It was very encouraging to see that 47 (72.3%) out of the 65 responding organizations provided some opportunities for their software testing staff to receive training in software testing. Commercial external training courses were the most popular (reported by 37 organizations), followed by internal courses (25 organizations) and self-study (22 organizations). In terms of frequency of training, 28 (59.6%) out of the 47 organizations provided training to staff only on a needs basis. It is to our disappointment to report that only 7 organizations (14.9%) offered regular training to their software testing employees.

Table VII - Barriers to provide training to software testing staff

Barrier	Response	Rank
Do not think there is any barrier	18	3
Cost	**31**	**1**
Time	22	2
Course	14	
Other	10	

In terms of barriers to providing training, cost is still considered to be the most significant factor (31 organizations), followed by availability of time (22 organizations). It is indeed disappointing to see that there are only 18 (27.7%) out of 65 organizations that did not believe there was any barrier to provide training to software testing staff (Table VII).

3.8 Test Organization - Teams, Independent Testers and Training

Out of the 65 organizations, 44 of them (67.7%) had an independent testing team. Among these 44, 26 organizations (59.1%) had over 80% of independent testers in the software testing team (i.e. testing personnel that do not participate in any software design or implementation activities). Furthermore, only 10 organizations (22.7%) had over 80% of their testing team members completing formal training in software testing, and 7 organizations (15.9%) had 60 to 79%. However, there were also 15 organizations (34.1%)

with less than 20% of their testers being formally trained. As mentioned earlier, this finding reveals the inadequacy of formal training of many testing staff, and suggests that there may be an urgent need to provide more opportunities for formal training in software testing.

From the collected data, 27 participants (61.4%) reported that less than 20% of their testing team members received training in software testing through university studies. There were 9 organizations (20.5%) reported to have over 80% of their testing team members trained by in-service training courses, whilst in 7 organizations (15.9%) this was between 60 to 79%. However, as many as 19 organizations (43.2%) had less than 20% of their testers receiving formal training by attending in-service training courses. This high percentage may indicate that there is a possible divergence between the courses provided by commercial providers and the actual needs of the industry. In addition, when asked for their required minimum qualification for software testers, more than one-third of organizations specifically required candidates with some previous testing knowledge and experiences, indicating that there is a very high demand to offer more education and training opportunities to the novice software testers.

4. Analysis and Summary of Survey Findings

As stated by Kitchenham and Pfleeger in [4], if a sample is not representative of the population then one cannot make definite generalizations of the population. Therefore, due to the smaller than expected survey sample we were unable to prove or disprove our hypotheses. Nevertheless, the survey provides some very valuable insights to the current software testing practices in Australia. This section gives a broader analysis of our survey findings.

4.1 Major Barriers and Disadvantages

The most evident barrier to using software testing methodologies and techniques was found to be a lack of expertise among the practitioners, with almost half of the respondents giving the same answer. This finding suggests that there could be a vast number of software testing staff who are not been appropriately trained in the use of formal testing methodologies or techniques. This may signify a deficiency in the training of software testing professionals to meet the actual demand of the industry, or deficiencies in the techniques themselves.

Cost was ranked first in the list of barriers to the use of automated testing tools (Table IV) and also in the list of barriers to provide training to software testing staff (Table VII) in organizations. In fact, cost was also ranked highly as a barrier to using testing metrics and standards in organizations. This could possibly be due to impact of the IT economy downturn in recent years, resulting in a much more competitive environment in the current IT industry.

Time is another critical impediment in the view of respondents. A high proportion regarded using automated tools (Table IV), metrics (Table V) and standards (Table VI) in their organizations as time consuming.

Difficulty of use was ranked (by about one quarter of respondents) as the third barrier to adopting automated testing tools (Table IV). There could be three reasons for this. Firstly, organizations may not be familiar enough with automated tools in general, so when they intend to purchase a tool they have no way of assessing the type of tool they require or how to judge the ease of use of the tools. Conversely, it could be that tool vendors do not provide sufficient on-the-job training when selling their tools to organizations. Thirdly, the tools themselves may be difficult to adopt. The same factor was ranked fifth in the metrics section (Table V).

4.2 Organization Sectors Adopting Structured Testing Methodology

It is our initial feeling that Government and public non-commercial organizations, being public-funded, are very likely to adopt structured testing methodology. To our surprise, we found that while there are about 70% of private organizations (both local and overseas) adopting some form of structured testing methodology, Government and public organizations reported a significantly lower percentage. Although this observation is only indicative due to the small sample size, it does reveal there is substantial room for improvement in software testing practices within government and public organizations.

4.3 Popularity of the Test Case Derivation Methods

Section 3.3 shows that in general, test case derivation is reasonably widely used amongst the respondents. Our conjecture would be that this is a manual process that connects to some extent with design practices, and which may support demonstrations to users more readily than automated test case generation. It is also possible that existing undergraduate computer science and software engineering programs embed this in their programming and/or testing subjects. The survey results also reveal

that deriving test cases from specifications (i.e. using black-box strategies) was likely to be more popular than deriving test cases from program codes (white-box strategies) in industry.

4.4 Testing Budgets

As reported in Section 3.1, about three-quarters of organizations allocated less than 40% of their development budget to software testing activities and only about one-fifth of the organizations could adhere to or spend less than their allocated testing budget. This could be a strong indication that software development organizations are not allocating realistic budgets to testing, or that their methods of estimating testing costs are non-realistic. We encourage organizations to establish databases of both estimated and actual testing costs in various kinds of software development projects, thus providing real life data for more accurate estimation of testing costs in future projects. It could be interesting to further study the principal strategies adopted by organizations in allocating budget to testing in the planning phase of the projects.

4.5 External Testers

As an overall analysis, there was a substantial level of satisfaction among organizations that hired external testers to assist them in software testing activities. We predicted that in future years, hiring external testers may become even more popular. This certainly indicates an increasing need of professional testers in Australia. At the same time, certification of software testers may become progressively more important, in order to guarantee the standard of service offered by external testers.

4.6 Stopping Rules and Metrics

One major point of concern with the survey responses was the methods of deciding when to stop testing (Section 3.3). While there is a number of practitioners still using such rules, stopping when resources run out is not regarded as a reasonable metric [9]. However, defining and using stopping rules is never simple or easy. Without the use of statistical models such as fault seeding or confidence bounds as discussed by Pfleeger [8] or reliability models derived by Musa and Ackerman [5], it could be potentially risky and even disastrous to the quality of the software by using non-statistical criteria.

As a matter of interest, 35 out of the 42 organizations (83.3%) using structured testing methodologies also used stop-testing criteria, and 33 out of the 42 respondents (78.6%) using structured testing methodologies also used testing metrics. This could indicate that the majority of organizations in industry that use structured methodologies also use metrics or stop-testing criteria. This phenomenon is further reinforced by the observation that 30 out of these 42 organizations (71.4%) employ both stop-testing criterions as well as metrics. It is also interesting to observe that out of the 22 organizations which do not use any structured testing methodology, 9 of them (40.9%) neither use software testing metrics nor stop-testing criterions at all. It seems fair to say there still exists a significant fraction of practitioners performing ad-hoc testing activities in Australia.

4.7 Automated Tools

As reported in Section 3.4, the most popular type of tools used is to support test execution (35 out of 44 organizations), followed by regression testing (33 organizations), with result analysis and reporting tools (27 organizations) being the third. This result is not surprising to us as these activities are very labour intensive and as such there are plenty of well-established tools in the market to handle these tasks.

Another interesting point to report is that out of the 42 organizations that use structured testing methodologies, 34 (80.9%) also used automated tools, while 10 out of 23 (43.5%) did not use any testing methodology but did use testing tools. These results show that there exists a large demand of automated tools in the software testing industry. Provided that these tools are of high quality and the tool vendors provide sufficient training to the users, organizations are eager to adopt automated tools to facilitate their testing activities.

4.8 Standards

As reported in Section 3.6, very few organizations reportedly used published standards. Most organizations that use standards either develop their own from scratch or modify published standards to suit their needs. This insinuates that there may be a deficiency in the existing published software testing standards to suit the environment of Australian's organizations, and suggests that relevant professional bodies in Australia, such as ACS or SEA, should consider forming a special interest group to establish a set of software testing guidelines specifically for practitioners in Australia, and then transform these guidelines into standards when they are further improved and generally accepted by the majority of practitioners in Australia.

4.9 Training and Education

Our results indicate that training courses offered by universities or TAFE colleges contribute only 10.7% of the total training opportunities for organizations to train their testing staff. This may be due to the lack of practical skills delivered to tertiary students in traditional software testing courses, or the lack of short courses in software testing at university. We anticipate that in the future, more practical research in software testing will be carried out in universities. Perhaps these research results could be incorporated into university courses to provide more modern and useful skills to students and meet the rising demands of industry.

5. Conclusions and Future Work

In this paper, we presented and analyzed the findings of our preliminary software testing survey conducted in several major capital cities in Australia between 2002 and 2003. Although the sample size was smaller than ideal, we are confident that our findings reveal some trends of the current industry practices in software testing.

As a second stage of the survey, we plan to increase the sample population to facilitate a more vigorous statistical analysis of the obtained data. We would also like to compare the data from the Australian industry to that obtained from other Southeast Asian countries in order to assess the competitiveness of Australia among its neighbours in the Asia-Pacific region.

Since the reliability of the survey sample has not been firmly established, all organizations involved in the mail out that did not respond to our call for participation will be contacted again to ascertain their reason for not participating. This may give a better indication as to the reliability of the survey sample, as well as whether or not our generalizations are valid.

As we remarked earlier, there is anecdotal evidence in Australia at least, that substantial resources are being committed to testing by some developers. At the same time, users continue to grapple with faulty software (at a time when extremely high quality infrastructure systems, e.g. EFTPOS, exist).

The need for surveys of this type is clear. Establishing the optimum relationship between testing and software quality; that is ensuring that testing strategies are in place which yield the highest quality software, is increasingly important as software begins to intrude more and more into people's daily lives. We are convinced that this survey, despite its limitations, will assist in this process.

Acknowledgements

The work described in this paper was supported by a grant from the Australian Computer Society. We would also like to acknowledge Australian Computer Society and Software Engineering Australia for including our promotion flyers in the magazines for circulation to their members. Dr. Marcelle Schwartz of La Trobe University provided invaluable statistical advice, and Thilky Perera of Bond University assisted in developing the survey sample. Last but not the least, we are grateful to all respondents of the survey. Without their efforts and enthusiasms, this survey could never be successful. Finally, any errors or omissions are the fault of the authors.

References

[1] P. Ammann, "System Testing via Mutation Analysis of Model Checking Specifications", *ACM SIGSOFT Software Engineering Notes*, Volume 25, No. 1, January 2000, pp 30.

[2] Murial Daran, Pascale Thévenod-Fosse, "Software Error Analysis: a Real Case Study involving Real Faults and Mutations", *ACM SIGSOFT Software Engineering Notes*, Volume 21, No. 3, May 1996, pp 168-171.

[3] L. Groves, R. Nickson, G. Reeves, S. Reeves and M. Utting "A Survey of Software Practices in the New Zealand Software Industry", *Proceedings of the.2000 Australian Software Engineering Conference,* Gold Coast, Queensland, Australia, April 28-30, 2000 pp 189-101.

[4] B. Kitchenham and S. L. Pfleeger, "Principles of Survey Research, Part 5: Populations and Samples", *ACM SIGSOFT, Software Engineering Notes*, Volume 27, No. 5, September 2000, pp 17-20.

[5] J. D. Musa and A. F. Ackerman, "Quantifying Software Validation: When to Stop Testing?", *IEEE Software*, May 1989, pp 19–27.

[6] N. Mateev, V. Menon, K. Pingali, "Fractal Symbolic Analysis", *Proceedings of the 15th international conference on Supercomputing*, June 2001, pp 38-49.

[7] T. Murnane and K. Reed. "On the Effectiveness of Mutation Analysis as a Black-box Testing Technique", *Proceedings of the 2001 Australian Software Engineering Conference*, Canberra, Australia, 2001, pp 12-20.

[8] S. L. Pfleeger, *Software Engineering: Theory and Practice*, Prentice-Hall, USA, 2001.

[9] R. S. Pressman, *Software Engineering: A Practitioners Approach*, McGraw-Hill, International Edition, 1992.

[10] K. Reed, "Testing, testing, testing", *Australian Computer Society Information Age*, April/May 2002, pp 56–58.

Towards Describing Black-Box Testing Methods as Atomic Rules

Tafline Murnane, Richard Hall and Karl Reed
Department of Computer Science and Computer Engineering
La Trobe University, Bundoora, Australia
{t.murnane, richard.hall, k.reed}@latrobe.edu.au

Abstract

Ideally, all black-box testing methods should be interpreted in the same way by different testers. In reality however, inconsistencies and ambiguities in original method descriptions may lead to differing interpretations and varying test set quality. In this paper, we decompose these methods into Atomic Rules for selecting test data and constructing test cases. We validate the rules via a worked example and discuss a pilot experiment to determine whether Atomic Rules are simpler to learn and use. Our approach also enables method tailoring and may simplify method comparison.

1. Introduction

Some practitioners may argue that skilled black-box testers should be able to derive high-yield test sets purely from knowledge and experience, and that test selection methods like Equivalence Partitioning (EP) should only be used to supplement heuristic knowledge [21]. Our view is that prescriptive black-box methods are essential in software engineering, and that an artificial division exists between novices and experts due to at least three consistency problems in these methods: definition by exclusion, multiple versions and notational differences.

Some methods do not clearly define how to select invalid test data. For example, an invalid EP equivalence class "has something else" [17] is often defined, containing all inputs other than those that are specified as valid. Ideally, a member from every class of data would be selected, but a novice may only be aware of a subset of these classes. Thus, *definition by exclusion* assumes familiarity with the 'universe of discourse' with respect to program inputs. As a result, different testers may produce vastly dissimilar test sets and, consequently, statements relating to program correctness and the nature of faults detected may be meaningless.

In addition, *multiple versions* of each method exist. For example, one Boundary Value Analysis (BVA) approach selects test data on, inside and outside field boundaries [4], while others do not include inside [5, 9, 10, 14, 17] and outside [8, 14] boundaries. Presently, no textbook, standard or paper describes every method or version. Thus, testers may not know how their chosen approach compares to others, even within an approach.

Method learning is complicated by *notational differences*, as a new notation must be understood for each new method learnt. For example, EP and BVA are often described as partitioning approaches in which the input domain is subdivided [17]. Syntax Testing[1] (ST) is not described in this way, despite the fact that partitions are implicitly created on each input parameter. Also, boundary values are selected by at least two versions of ST [2, 12], and both EP and BVA select values that lie outside the boundaries of numerical fields [17]. Although it may seem as if ST is highly dissimilar to EP and BVA, these overlaps suggest that these methods could be described in the same way. In fact, every method consists of rules for selecting test cases and can be described by the same three-step process.

1. Select valid and invalid data sets called partitions for each input and/or output parameter[2].
2. Select at least one individual data value from each partition chosen in (1).
3. Select various combinations of the data values chosen in (2) to construct test cases.

We believe that the consistency problems discussed here can be solved by decomposing these methods into *Atomic Rules* that are able to produce consistent test sets regardless of individual knowledge or experience. The methods we have investigated include those that can be applied to specifications that give the data definitions of program input/output parameters. A characterisation schema for our Atomic Rules is discussed in Section 2. As proof of concept, in Section 3 we decompose EP and BVA and show that the resulting Atomic Rules can be combined to describe the original methods. In Section 4 we use a worked example[3] to demonstrate that test cases derived from a representative specification by the original EP and BVA approaches can also be produced by Atomic Rules. Section 5 discusses the preliminary results of a pilot experiment which analysed method learnability and usability. Section 6 presents a tailored approach to black-box testing.

---

[1] Syntax testing is also known as 'input validation testing' and 'grammar-based testing.'

[2] In essence, our work suggests that all black-box test case selection methods are in fact based on the equivalence partitioning concept.

[3] Worked examples were used as proof of concept by Rugg, McGeorge and Maiden [20].

Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)
0730-3157/05 $20.00 © 2005 **IEEE**

2. The Atomic Rule Schema

A characterisation schema for Atomic Rules was created (see Table 1). Such schemas have been used to standardize other software engineering "technologies" to facilitate the selection of appropriate techniques with respect to specific problem domains (see [3, 11, 18, 19, 20, 22]). We applied this approach to black-box methods described in thirteen different places [1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 17] to determine their common characteristics. Four attributes (Table 1: column 1) are not self-explanatory and thus require discussion: rule type, original datatype**,** test datatype and set type.

The *rule type* attribute differentiates between *where* rules are used in the three-step test case selection process, as follows.

1. Select valid and invalid partitions for each input and/or output parameter by applying a *Data-Set Selection Rule* (DSSR) to each parameter.
2. Select at least one individual data value from each partition chosen in (1) by applying a *Data-Item Selection Rule* (DISR) to each partition.
3. Select various combinations of the data values chosen in (2) by applying a *Test Case Construction Rule* (TCCR) to create test cases.

Original datatype describes the data domain to which a rule can be applied, while *test datatype* describes the data domain that may be selected as test data. For example, some ST rules only apply to delimiter fields, while others select data of a different datatype from a field's original. Both attributes make use of six datatypes necessary for EP and BVA rules (see Appendix 2 for our definitions of integer, real, alpha, alphanumeric, non-alphanumeric and null). Rules derived from other methods may use other datatypes (such as those defined in [13] and [23]).

Finally, *set type* differentiates between data defined as lists and ranges. For example, some EP rules only apply to ranges of numerical data, while others only apply to data stored in lists [17]. *Lists* can be expressed as $L ::= [v_1 \mid v_2 \mid \ldots \mid v_n]$ where n is the number of v values in the list, and *ranges* as $\{R : lb \le R \le ub\}$ or $R ::= [lb - ub]$ which denotes a range of values from lower boundary *lb* to upper boundary *ub*. These terms were adapted from similar concepts discussed in [8:p79, 10, 14, 17].

3. Decomposition and Recomposition

Twenty-two Atomic Rules (see Appendix 2) were derived from EP and BVA by analysing various versions of these methods with respect to the Atomic Rules schema. To avoid definition by exclusion, all invalid equivalence classes were selected from within the six datatypes defined in the previous section (for example, see EP4…EP8). EP rules are positioned within the three-step test case selection process as follows.

Table 1: The Atomic Rules schema.

Attribute	Type	Definition
Test Method	enum	Method from which the rule was derived.
Number	string	A unique identifier given to each rule.
Name	string	A name given to each rule.
Description	string	Describes what the rule does.
Source	enum	Reference(s) from which rule was derived. NA denotes rules defined in this paper.
Rule Type	enum	Possible values: DSSR, DISR, TCCR (see discussion in Section 2).
Set Type	enum	Specifies the set type to which each rule applies. Options are List and Range.
Valid or Invalid	enum	Defines whether the rule selects Valid or Invalid test data.
Original Datatype	*data type*	Defines the datatypes to which each rule can be applied: integer, real, alpha, alphanumeric, non-alphanumeric, null, or "All" if all datatypes apply.
Test Datatype	*data type*	Defines the datatype of selected test data: integer, real, alpha, alphanumeric, non-alphanumeric, null, "All" if all datatypes apply, or "Same as original" if it is the same as Original Datatype.
Test Data Length	integer	Specifies the maximum length of selected test data. If original datatype and test datatype are the same, then "Same as original" will appear.
# Fields Populated	string	Number of input/output parameters for which the rule selects test data.
# Tests Derived	string	Counts the number of test cases derived. DSSRs and DISRs do not select tests so they are 0. TCCRs hold an equation to calculate this, based on the number of parameters in the test case.

Equivalence Partitioning

1. Select equivalence classes:
   a) if *set type* ≡ *range* then apply rules EP1…EP9,
   b) if *set type* ≡ *list* then apply rules EP4…EP10.
2. Select one data value from each equivalence class selected in (1) by applying EP11 to each class.
3. Select test cases that are:
   a) *valid*: by applying EP12 to valid data values chosen in (2),
   b) *invalid*: by applying EP13 to invalid data values chosen in (2).

We use several EP rules within the equivalent BVA definition because there is an overlap between EP and BVA. Consequently, EP rules which select equivalence classes need to be applied before BVA rules (e.g. see [17]). EP rules which construct test cases are also used.

Boundary Value Analysis

1. Select valid equivalence classes:
   a) if *set type* ≡ *range* then apply rule EP3,
   b) if *set type* ≡ *list* then apply rule EP10.
2. Select boundary values for:
   a) each class chosen in (1a) by applying BVA1…BVA6 and BVA9 to each class,
   b) each class chosen in (1b) by applying BVA7…BVA9 to each class.
3. Select test cases that are:
   a) *valid*: by applying EP12 to valid data values chosen in (2),
   b) *invalid*: by applying EP13 to invalid data values chosen in (2).

4. Validation

To validate our approach we need to show correspondence between the proposed Atomic Rules and the relevant black-box methods (specifically Myers' EP and BVA [17]). For a complete demonstration, it would be necessary to generate test cases for specifications covering every possible type of input. However, it is unnecessary to include all datatypes because data is either defined as ranges or lists, thus it is adequate to include at least one example of both. The following specification, written in standard BNF (see Figure 1) fulfils this requirement, having both data lists (street, suburb and postcode) and a data range (house_number).

```
<address> ::= <house_number><street><suburb><postcode>
<house_number> ::= [1 – 9999]
<street> ::= [Annensen Court | Aaran Close | …| Zoo Court]
<suburb> ::= [Abbotsford | Aberfeldie | … |Yooralla |Yuroke]
<postcode> ::= [0800 | 0801 | 0810 | … |7325 | 7330 | 7331
                | 7466 | 7467 | 7468 | 7469 | 7470]
```

Figure 1: A simple address validation specification [16].

We first generate equivalence classes for this specification, as our BVA process requires that a number of EP Atomic Rules already be applied. To show correspondence, we compare classes derived by Myers' approach to those selected by EP Atomic Rules (see Table 2). The third column contains three sub-columns: the first describes the corresponding Myers' equivalence class number (by which the column is ordered), the second is the identification number of the Atomic Rule that derived the equivalence class, and the third is the actual equivalence class chosen by the Atomic Rule.

By visual inspection, every equivalence class selected by Myers' approach is matched by at least one class derived by a corresponding Atomic Rule.

Table 2: Equivalence classes selected by Myers' approach and the corresponding Atomic Rules.

Parameter	Myers' Equivalence Classes	Corresponding Atomic Rule Equivalence Classes
house_ number	1) number 1 to 9999	1 & 4) EP3: 1 to 9999
	2) number < 1	2 & 4) EP1: < 1
	3) number > 9999	3 & 4) EP2: > 9999
	4) has digits	5) EP5: real
	5) has something else	5) EP6: alpha
		5) EP7: alphanumeric
		5) EP8: non-alphanumeric
		5) EP9: missing value
street	6) pick street from list	6 & 7) EP10: pick from list
	7) has letters	8) EP4: integer
	8) has something else	8) EP5: real
		8) EP7: alphanumeric
		8) EP8: non-alphanumeric
		8) EP9: missing value
suburb	9) pick suburb from list	9 & 10) EP10: pick from list
	10) has letters	11) EP4: integer
	11) has something else	11) EP5: real
		11) EP7: alphanumeric
		11) EP8: non-alphanumeric
		11) EP9: missing value
postcode	12) pick postcode from list	12 & 13) EP10: pick from list
	13) has digits	14) EP5: real
	14) has something else	14) EP5: alpha
		14) EP7: alphanumeric
		14) EP8: non-alphanumeric
		14) EP9: missing value

Next, we compare the boundaries derived by Myers' BVA to those selected by Atomic Rules, both using valid equivalence classes (see Table 3). The third column contains three sub-columns: the identification number of the matching Myers' boundary, the number of the Atomic Rule that derived the corresponding boundary, and the actual boundary chosen by the Atomic Rule.

Table 3: Boundaries selected by Myers' approach and by the corresponding Atomic Rules.

Parameter (Equivalence Class)	Myers' Boundaries	Corresponding Atomic Rule Boundaries
house_ number (1 to 9999)	1) below lower boundary 2) on lower boundary 3) on upper boundary 4) above upper boundary	1) BVA1: lower boundary - 1[4] 2) BVA2:on lower boundary 3) BVA5: upper boundary 4) BVA6: upper boundary + 1
street (pick street from list)	5) select first street 6) select last street	5) BVA7: select first list item 6) BVA8: select last list item
suburb (pick suburb from list)	7) select fist suburb 8) select last suburb	7) BVA7: select first list item 8) BVA8: select last list item
postcode (pick postcode from list)	9) select first postcode 10) select last postcode	9) BVA7: select first list item in list 10) BVA8: select last item in list

Additional boundaries could have been selected by Atomic Rules. For example, the *BVA9: Missing Item* rule also applies to the *street* field. However, these are not included in Table 3 because complete coverage of Myers' boundaries has been achieved.

In both cases, our rules have achieved complete coverage of the equivalence classes and boundary values selected by Myers' approach. Having presented a basic validation of the Atomic Rules approach, the following section discusses the results of a preliminary evaluation.

5. Preliminary Evaluation

To determine whether novice testers find Atomic Rules easier to learn and use than original approaches, a pilot experiment was conducted. Thirty-three university students enrolled in a third-year software testing course were exposed to two different representations of EP and BVA: Myers' approach and the corresponding Atomic Rules. The students were divided into two groups, based on which approach they learnt first; one group learnt Myers' representation first while the second group learnt the corresponding Atomic Rules first[5]. A questionnaire surveyed students on their experience of the approaches. We now discuss a summary of our preliminary findings[6].

Students rated their initial and final understanding of EP and BVA (Table 4: columns 2-5) using a Likert scale of: 1 = Very Poor, 2 = Poor, 3 = Average, 4 = Good, 5 = Very Good, 6 = Excellent. This table shows that all students increased their understanding of EP and BVA.

[4] For rules that increment or decrement upper or lower boundaries by 1, this unity is equivalent to the minimum positive value that can be described by whichever number representation scheme is appropriate.

[5] The two approaches were taught by different lecturers, thus we plan to re-run the experiment this year (2005) with the lecturers swapped.

[6] See [15] for more information on the experiment design.

Students also rated their understanding of Myers' approach and Atomic Rules (Table 4: cols 6-7) using the same scale. Overall, 57% rated their understanding of Myers' approach as below-average, whereas 100% rated their understanding of Atomic Rules as Good or above. A t-test showed a significant increase in their self-rated understanding of Atomic Rules compared to Myers' approach; $t(30) = -7.65$, $p < .01$. Thus, students were able to gain a better understanding of Atomic Rules.

Table 4: Student survey results.

	Understanding of Black-Box Testing Methods				Understanding of Approaches	
	Initial		Final			
	EP	BVA	EP	BVA	Myers	Atomic Rules
Rating	**Percentages**					
Very Poor	3	9	0	0	6	0
Poor	15	18	0	0	15	0
Average	36	45	0	3	36	0
Good	15	18	12	21	15	12
Very Good	24	9	58	55	18	70
Excellent	6	0	30	21	3	18
Frequencies	**Values**					
Mean	3.61	3.00	5.18	4.94	3.35	5.03
St. Deviation	1.27	1.06	0.64	0.75	1.25	0.56

Students also indicated which method they learnt first and which they would choose to use in future (Table 5). Due to variation in enrolments and timetables, more students learnt Myers' approach first. A t-test examined group differences in student's self-rated understanding of Myers' representation (Table 4: col 6) across the order in which the approach was learnt (Table 5: col 2), showing a significantly greater understanding if Myers's approach was learnt before Atomic Rules; $t(29) = 2.67$, $p = .01$. Conversely, a t-test that examined group differences in understanding of Atomic Rules (Table 4: col 7) across approach learnt first (Table 5: col 2) indicated that the order in which students learnt Atomic Rules did not have a significant effect on their understanding of the approach; $t(31) = -0.77$, $p = .45$. This suggests that Atomic Rules are simpler for novices to learn.

Table 5: Approach students leant first versus the approach they indicated they would use in future.

Approach	Leant First (%)	Use in Future (%)
Myers	61	9
Atomic Rules	39	91

A chi-square test showed that a significantly higher number of students indicated that they prefer to use Atomic Rules in future; $\chi^2(1, N=33) = 22.09$, $p < .001$.

6. Atomic Rules Tailoring

The decomposition of black-box methods into Atomic Rules enables custom combinations of these rules to be selected for a specific purpose or within particular constraints (i.e. tailoring [20]). For a simple example, consider a specification in which all data is of the set type list (see Figure 2).

```
<country> ::= [Afghanistan | … |Yemen | … | Zimbabwe]
<telephone_ code> ::= [1 | 7 | 20 | 30 | 31 | … | 88213 | 88216]
```

Figure 2: A specification for international telephone codes.

With this constraint, only those rules with attribute *set type* ≡ *list* are applicable. By inspection (see Appendix 2) these include EP4…EP13 and BVA7…BVA9.

7. Conclusion

In this paper we developed a representation for black-box testing methods which addressed three consistency problems: notational differences, definition by exclusion, and multiple versions. First, by decomposing the original methods using a characterisation schema, we created a uniform notation called Atomic Rules. Second, by using explicit datatypes within the schema, we avoid definition by exclusion. Finally, it was unnecessary to populate the schema with all versions of every method to demonstrate correspondence with our Atomic Rules. Instead, through a representative worked example using Myers' original definition, we showed that the process of decomposing several versions of equivalence partitioning and boundary value analysis was invertible.

We did not attempt to solve the multiple versions problem here. However, we are presently engaged in four things: decomposing all black-box methods and versions into Atomic Rules, determining the optimal order of rule application, exploring the feasibility of unifying all methods into a super-method, and comparing the fault-detection ability of the super-method to that of the original methods. In so doing, we expect all redundant overlaps between methods and versions to be eliminated. For example, EP and BVA both include rules that select test data from inside and outside field boundaries. Such redundancy is evident in *BVA1: Lower Boundary - 1 Selection* and *EP1: < Lower Boundary Selection*. If both EP and BVA were used in testing, it may be unnecessary to use both rules. Also, while the results of the pilot experiment that assessed the learnability and usability of Atomic Rules by novice testers were encouraging, we plan to repeat this experiment to eliminate irrelevant environmental features, such as student preference for lecturer.

We believe that a complete representation of black-box methods as Atomic Rule is essential for facilitating test process tailoring. We have demonstrated that the rules described in this paper can be used for at least one tailoring example. Ultimately, we would like testers to be able to select the rules that apply to their specific problem domain in a consistent and repeatable way.

Acknowledgements

We thank John Murnane of the University of Melbourne, and the COMPSAC reviewers, for their invaluable comments. We thank the Victorian State Government and La Trobe University's Department of Computer Science and Computer Engineering for the research scholarships that support Miss Murnane's PhD candidature. Finally, we thank the students of CSE32STR 2004 for participating in our experiment.

Appendix 1 – Datatypes/Datatype Schema

This appendix contains the datatype schema (Table 6) and six datatypes (Table 7) defined for use with EP and BVA (note that the structure of the two tables only differs to conserve space). Future research may extend these datatypes for use in other testing methods.

Table 6: The datatype schema.

Attribute	Type	Definition
Name	string	A unique name for each data type.
Set Type	enum	Describes the set type of the datatype. Options are List and Range.
Size	string	Max length of datatype in bytes. Length can depend on implementation using the datatype [13], for which "Max buffer length" will appear.
Example	string	A simple example.

Table 7: Datatypes defined using the datatype schema.

Name	Set Type	Size	Example
Integer	List & Range	Max buffer length	List: [-30, 4, 16, -1, 25] Range: [-16 – 33]
Real	List & Range	Max buffer length	List: [10.4, -100.5, 3.2] Range: [-12.1 – 54.23]
Alpha	List & Range	1 byte	List: [e, a, n, B, c, H, l] Range: [e - g]
Alphanumeric	List	Max buffer length	List: [4z3A, A83, b44]
Non-Alphanumeric	List & Range	1 byte	List: [", (, %, *, ", +, &] Range: [" - +]
Null (empty)	List	0 bytes	List: []

Appendix 2 – Atomic Rules for Equivalence Partitioning and Boundary Value Analysis

Table 8 contains the Atomic Rules that were defined for EP and BVA using the schema in Table 1.

Table 8: Atomic Rules for Equivalence Partitioning (EP) and Boundary Value Analysis (BVA).

Attribute	Values	Values	Values	Values	Values
Test Method	EP	EP	EP	EP	EP
Number	EP1	EP2	EP3	EP4	EP5
Name	< Lower Boundary Selection	> Upper Boundary Selection	Lower to Upper Boundary Selection	Integer Replacement	Real Replacement
Description	Select an equivalence class containing values below lower boundary	Select an equivalence class containing values above upper boundary	Select an equivalence class containing values between boundaries	Select an equivalence class containing every integer value	Select an equivalence class containing every real value
Source	[17] & [4]	[17] & [4]	[17] & [4]	NA	[4]
Rule Type	DSSR	DSSR	DSSR	DSSR	DSSR
Set Type	Range	Range	Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Valid	Invalid	Invalid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Real, Alpha, Alphanumeric, Non-Alphanumeric	Integer, Alpha, Alphanumeric, Non-Alphanumeric
Test Datatype	Same as original	Same as original	Same as original	Integer	Real
Test Data Length	Same as original	Same as original	Same as original	Max length of datatype	Max length of datatype
# Fields Populated	1	1	1	1	1
# Tests Derived	0	0	0	0	0

Attribute	Values	Values	Values	Values	Values
Test Method	EP	EP	EP	EP	EP
Number	EP6	EP7	EP8	EP9	EP10
Name	Alpha Replacement	Alphanumeric Replacement	Non-Alphanumeric Replacement	Missing Value Replacement	Valid List Selection
Description	Select an equivalence class containing every alpha value	Select an equivalence class containing every alphanumeric value	Select an equivalence class containing non-alphanumeric values	Select an equivalence class containing a NULL value	Select an equivalence class containing all values in specified list
Source	[4]	NA	NA	NA	[17]
Rule Type	DSSR	DSSR	DSSR	DSSR	DSSR
Set Type	List or Range	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Invalid	Valid
Original Datatype	Integer, Real, Alphanumeric, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Alphanumeric	All	All
Test Datatype	Alpha	Alphanumeric	Non-Alphanumeric	Null	Same as original
Test Data Length	1	Max length of datatype	1	0	Max length of datatype
# Fields Populated	1	1	1	1	1
# Tests Derived	0	0	0	0	0

Attribute	Values	Values	Values	Values	Values
Test Method	EP	EP	EP	BVA	BVA
Number	EP11	EP12	EP13	BVA1	BVA2
Name	Data Value Selector	Valid Test Case Constructor	Invalid Test Case Constructor	Lower Boundary - 1 Selection	Lower Boundary Selection
Description	Randomly selects one data value from an equivalence class	Construct minimum # of tests required to cover all valid classes	Construct one test per invalid class (one field invalid, all others valid)	Select value at lower boundary - 1	Select value at lower boundary
Source	NA	[17]	[17]	[4]	[4]
Rule Type	DISR	TCCR	TCCR	DISR	DISR
Set Type	List or Range	List or Range	List or Range	Range	Range
Valid or Invalid	Depends on the class	Valid	Invalid	Invalid	Valid
Original Datatype	All	All	All	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric
Test Datatype	Same as original	Same as original	Same as original	Same as original	Same as original
Test Data Length	Max length of datatype	Max length of datatype	Max length of datatype	Same as original	Same as original
# Fields Populated	1	*n*, the number of fields	*n*, the number of fields	1	1
# Tests Derived	0	*1 to m*, where *m* = # valid classes selected	*m*, where *m* = # invalid classes selected	0	0

Attribute	Values	Values	Values	Values	Values
Test Method	BVA	BVA	BVA	BVA	BVA
Number	BVA3	BVA4	BVA5	BVA6	BVA7
Name	Lower Boundary + 1 Selection	Upper Boundary - 1 Selection	Upper Boundary Selection	Upper Boundary + 1 Selection	First List Item Selection
Description	Select value at lower boundary + 1	Select value at upper boundary - 1	Select value at upper boundary	Select value at upper boundary + 1	Select first item of a list
Source	[4]	[4]	[4]	[4]	[17]
Rule Type	DISR	DISR	DISR	DISR	DISR
Set Type	Range	Range	Range	Range	List
Valid or Invalid	Valid	Valid	Valid	Invalid	Valid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	Integer, Real, Alpha, Non-Alphanumeric	All
Test Datatype	Same as original	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original	Same as original	Same as original
# Fields Populated	1	1	1	1	1
# Tests Derived	0	0	0	0	0

Attribute	Values	Values
Test Method	BVA	BVA
Number	BVA8	BVA9
Name	Last List Item Selection	Missing Item Replacement
Description	Select last item of a list	Replace field with null
Source	[17]	[17]
Rule Type	DISR	DISR
Set Type	List	List or Range
Valid or Invalid	Valid	Invalid
Original Datatype	All	All
Test Datatype	Same as original	Null
Test Data Length	Same as original	0
# Fields Populated	1	1
# Tests Derived	0	0

References

[1] Beizer, B. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, New York, USA, 1984.

[2] Beizer, B. *Software Testing Techniques*. Von Nostrand Reinhold, New York, USA, 1990.

[3] Birk, A. Modelling the Application Domains of Software Engineering Technologies. *IESE Report No. 014.97/E*, Fraunhofer IESE, Germany, 1997.

[4] British Standards Institute. *Software Testing: Software Component Testing Standard*. British Computer Society, SIGIST, 1998.

[5] Craig, R.D., and Jaskiel, S.P. *Systematic Software Testing*. Artech House Publishers, USA, 2002.

[6] Grindal, M., Lindström, B., Offutt, A. J., and Andler, S. F. An Evaluation of Combination Strategies for Test Case Selection. *Technical Report HS-IDA-TR-03-001*, Department of Computer Science, University of Skövde, October 2004, pp 1-74.

[7] Hetzel, W. *The Complete Guide to Software Testing*. QED Information Sciences Inc, Wellesley, Massachusetts, USA, 1988.

[8] Jorgensen, P. *Software Testing: A Craftsman's Approach*. Department of Computer Science and Information Systems, Grand State University Allendale, Michigan and Software Paradigms, Rockford, Michigan, CRC Press, 1995.

[9] Kaner, C. *Testing Computer Software*. TAB Books Inc, USA, 1988.

[10] Lewis, W.E. *Software Testing and Continuous Quality Improvement*. CRC Press, Florida, USA, 2000.

[11] Maiden, N.A.M., and Rugg, G. ACRE: Selecting Methods for Requirements Acquisition. *Software Engineering Journal*, 11(3):183-192, May 1996.

[12] Marick, B. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*. Prentice Hall PTR, Englewood Cliffs, New Jersey, USA, 1995.

[13] Meek, B. A Taxonomy of Datatypes. *ACM SIGPLAN Notices*, 24(9):159-167, September 1994.

[14] Mosley, D.J. *The Handbook of MIS Application Software Testing. Methods, Techniques, and Tools for Assuring Quality Through Software Testing*. Prentice-Hall, USA, 1993.

[15] Murnane, T. Design of an Experiment to Analyse Black-Box Method Learnability and Usability. Technical Report 2/05, Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Australia, 2004.

[16] Murnane, T., and Reed, K. On the Effectiveness of Mutation Analysis as a Black-box Testing Technique. *Proceedings of the 2001 Australian Software Engineering Conference* (ASWEC '01), Canberra, Australia, IEEE, 2001, pp 12-20.

[17] Myers, G. *The Art of Software Testing*. John Wiley & Sons Inc, USA, 1979.

[18] Prieto-Díaz, R. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):89-97, 1991.

[19] Prieto-Díaz, R., and Freeman, P. Classifying Software for Reusability. *IEEE Software*, 4(1):6-16, 1987.

[20] Rugg, G., McGeorge, P., and Maiden, N. Method Fragments. *Expert Systems*, 17(5):248-257, November 2000.

[21] Sommerville, I. *Software Engineering*. 6th edition, Pearson Education Limited, England, UK, 2001.

[22] Vegas, S., Juristo, N., and Basili, V. *Implementing Relevant Information for Testing Technique Selection. An Instantiated Characterization Schema*. Kluwer Academic Publishers, USA, 2003.

[23] Wiederhold, G., and Qian, X. Database Engineering. In *The Encyclopaedia of Software Engineering*, vol 1:269-282, Wiley-Interscience, USA, 1994.

Tailoring of Black-Box Testing Methods

Tafline Murnane, Karl Reed and Richard Hall
Department of Computer Science and Computer Engineering
La Trobe University, Bundoora, Australia
{t.murnane, k.reed, richard.hall}@latrobe.edu.au

Abstract

Currently, black-box testing methods are effective yet incomplete. Consequently, test engineers may find it necessary to perform ad hoc customisation for each application under test. In this paper, we present procedures for customising black-box methods that model such "error guessing" in a reproducible and reusable way. As a preliminary evaluation, we customise a generalised representation of black-box methods and compare the effectiveness of the resulting test cases with those derived by two existing methods. Our procedures facilitate the development of both domain-specific and novel experimental black-box methods.

1. Introduction

The proper usage of black-box testing methods assumes two conditions hold: input and output fields are completely specified, and the methods themselves are complete. In reality, it appears that neither condition holds the majority of the time.

With respect to *field completeness*, a 2003/2004 survey of software testing practices amongst Australian software development organisations found that of sixty-five organisations interviewed, over half reported that 20-59% of their detected program defects were related to specification defects [27]. For a specific example of field incompleteness, consider the following real-life scenario. An engineer working on a requirements specification for financial software discovered that the program under development needed to validate credit card numbers. The engineer assumed that all members of the development team were familiar with valid credit card number formats, and thus omitted specifying their input data format explicitly. The result is a tester being unable to derive an effective black-box test set and unable to verify that the software meets the client's requirements.

Field definitions can be difficult to extract from specifications, partly because their data and behaviour can be specified in multiple places. For example, often different parts of a specification discuss processing, data inputs and error checking [1]. Consequently, there exists a need for a requirement elicitation procedure that ensures that program input/output data is completely specified, enabling effective black-box testing. Thus, the first aim of this paper is to provide an initial version of such a procedure.

On the other hand, the existence of ad hoc test methods such as Error Guessing[1] question *method completeness* [23]. As Jorgensen states, "special values testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform… There are no guidelines, other than to use 'best engineering judgement'. As a result, special value testing is very dependant on the abilities of the tester… Even though special value testing is highly subjective, it often results in a set of test cases which is more effective in revealing faults than the test sets generated by the other methods… testimony to the craft of software testing" [16]. However, while it may be more effective, its application cannot be guaranteed, since it is currently "ill-defined" [1], though some believe these approaches are systematic and amenable to abstraction [11]. For example, Test Catalogues collect special values in repositories [17, 22].

There are two ways black-box methods could be made more complete: by creating a generalised method and by creating procedures whereby any method could be systematically customised for a particular application domain. We have already created a generalised method [24], a process which was difficult as "testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades via scores of writers" [16]. Thus, its adoption is uncertain as it requires learning more terminology, and as no generalised method could make the claim of universality. An example of varying terminology can be seen in the various names given to data sets created by dividing the input and output domain into classes of homogenous data whose mapping involves executing identical deterministic processes [24]. In Category Partition Testing (CPT) these sets are called "choices" [28]; in Equivalence Partitioning (EP) they are called "equivalence classes" [26]; in Syntax Testing (ST) [10] they are implicitly created.

Thus, the second aim of this paper is to provide an initial set of procedures for Systematic Method Tailoring (SMT). They are generalisable because they are uncoupled from, and thus can be used with, any black-box testing method.

The organisation of this paper is as follows. In Section 2 we present our requirement elicitation

[1] Error Guessing is also referred to as 'Special Values Testing' [16].

Proceedings of the 2006 Australian Software Engineering Conference (ASWEC'06)
1530-0803/06 $20.00 © 2006 IEEE

procedure that ensures that input and output data is completely specified. In Section 3 we discuss our generalised representation called 'Atomic Rules' and show how it can be used with GQAS to generate test data. In Section 4 we present three procedures for tailoring black-box methods which, in theory, model Error Guessing. In Section 5 we conduct a preliminary evaluation of our requirement elicitation procedure and of the effectiveness of test cases derived by the Atomic Rules representation of two black-box methods to those derived by a tailored method. In Section 6 we conclude and discuss future research.

2. Goal/Question/Answer/Specify

As black-box test sets are derived by applying black-box methods to program specifications, they require detailed specifications that describe the application domain of each program under test in terms of its input and output fields. We propose a simple yet readily applied requirement elicitation procedure called Goal/Question/Answer/Specify (GQAS) which identifies information that is required to be captured in order to enable effective black-box testing. This procedure can be used during requirements elicitation or by testers prior to testing. Together, GQAS and our three tailoring approaches (see Section 4) are based on the Goal/Question/Metric (GQM) paradigm [3, 4, 5, 6].

GQAS collects what we consider to be the minimum information required to conduct black-box testing. That is, the datatype, set type and size in terms of minimum and maximum lengths of input/output data, whether the field is mandatory or repeats, and (ideally) the valid data set the program should accept and the invalid data set it should reject. While the last two items are essential for deriving valid and invalid test cases, generic tests can be constructed using only the first three. Also, if the data set is defined, then the datatype, set type and size can be deduced and can act as an error checking mechanism.

Each application of this technique results in one GQAS instance (i.e. one for each field being specified).

GQAS consists of the following four steps:

1. As a *goal*, state that a particular field is going to be specified for the purpose of conducting black-box testing.
2. Consider the following *questions*:
   a. What is the field's datatype? [Integer | Real | Alpha | Alphanumeric | Non-Alphanumeric]+[2]
   b. What is the field's set type? [Range | List]
   c. For Ranges, what are the minimum and maximum values; for Lists, what is the minimum and maximum length of valid data?
   d. What valid data set should the program accept and what invalid data should it reject?
   e. Is the field mandatory? [Yes | No]
   f. Does the field repeat? [Yes | No]; if Yes, what are the minimum and maximum number of repetitions?
3. Seek and record *answers* to these questions by searching for domain knowledge in textbooks, standards, papers, or websites[3], or by speaking to domain experts (e.g. clients or experienced testers). Each answer may state how it was obtained, as this may be useful for testing, development or maintenance of the program being specified and for future developments in the same domain.
4. *Specify* the field using a formal notation (e.g. Backus-Naur Form), including valid and invalid data sets, if available.

Subsequently, a set of test data selection rules that suit the newly defined field can be selected using a tailoring approach (Section 4). This step is analogous to the metrics selection step in GQM. However, one of the main differences between GQM and GQAS is that question definition is part of the GQM process, whereas GQAS has fixed questions as the same information is required when specifying any input or output field.

Although GQM and a number of other goal-oriented requirement engineering approaches have been used for requirement elicitation (e.g. see [9, 12, 19, 20, 32]), to the best of our knowledge, this is the first recorded use of GQM in the analysis of specification completeness and in the collection of domain knowledge, specifically in support of black-box testing activities.

3. The Atomic Rules Approach

The Atomic Rules approach [24] decomposes black-box testing methods into autonomous fragments for:- partitioning the program input and output domains (Data-Set Selection Rules); selecting test data from each partition (Data-Item Selection Rules); and constructing test cases (Test Case Construction Rules). A characterisation schema describes each rule's attributes, giving them a standardised representation (see Appendix 1). The following three-step process encapsulates black-box test case derivation procedures:

1. Select valid and invalid partitions for each input and/or output field by applying a *Data-Set Selection Rule* (DSSR) to each field.
2. Select at least one individual data value from each partition chosen in (1) by applying a *Data-Item Selection Rule* (DISR) to each partition.
3. Select various combinations of the data values chosen in (2) by applying a *Test Case Construction Rule* (TCCR) to create test cases.

When used in conjunction with a set of Atomic Rules of a specific method, these procedures can be used to construct black-box test cases in the usual way. For example, Myers' original definitions of EP and Boundary Value Analysis (BVA) have been reconstructed with Atomic Rules and have been shown to derive test cases that are equivalent to those derivable by the original methods [24].

Our generalised representation of black-box methods was produced by conducting a thorough study of the fundamental fragments of EP, BVA, ST and several combinatorial methods, including Orthogonal Array Testing [21], Specification-Based Mutation Testing [25] and those discussed by Grindal *et al.* [14] (see [24] for EP and BVA rules and Appendix 1 for ST rules).

[2] These datatypes were defined to enable EP and BVA to be described using Atomic Rules [24]. This can be extended by adding or combining datatypes; e.g. Alpha could divide into Single and Multiple Alpha.

[3] The correctness of domain knowledge obtained from web sites should be verified by domain experts before it is relied upon as being accurate.

GQAS can easily be applied to this generalised representation as several attributes of the Atomic Rules schema match questions asked in GQAS. Once a GQAS instance has been defined, an Atomic Rule set can be selected through the use of one of our three tailoring procedures, which are discussed in the following section.

4. Systematic Method Tailoring

In this section we present our three Systematic Method Tailoring (SMT) procedures:

1. Selection-based tailoring
2. Creation-based tailoring
3. Creation-based tailoring via selection, using:
    a. All combinations
    b. Paired combinations
    c. Selective combinations

Although each is discussed separately, in practice, a combination of all three approaches may be used.

4.1 Selection-Based Tailoring

In *selection-based tailoring*, new black-box methods are defined by matching the set type (i.e. range or list) of rules in the generalised representation against the set type of valid data for each field under test. It is a bottom-up approach that is based on existing black-box methods. For example, Myers provides different guidelines for deriving test data for each set type [26].

Thus, a new method m_n could be defined by selecting *i to j* rules from *k* to *l* methods $\{m_n = select(r_i, ..., r_j) : r_i, ..., r_{i+p} \in m_k, ..., r_{j-q}, ..., r_j \in m_l, m_k \neq m_l\}$, where each method contains a subset of rules that can be applied to a specification using the three-step process in Section 3.

For example, rules that could be selected to test *<age> ::= [0 – 150]* include *BVA1: lower boundary – 1 selection* and *BVA6: upper boundary + 1 selection*. However, these could not be applied to *<colour> ::= {brown | blue | green}*, as it is impossible to predict what comes before or after the lower and upper boundaries.

4.2 Creation-Based Tailoring

In *creation-based tailoring*, new Atomic Rules that have not been defined in existing methods are generated. This is useful when testers suspect that a specific input may be effective for testing a particular field, and is similar to Error Guessing [26]. However, as each rule is defined using the Atomic Rules schema, it is available for future reuse.

Thus, a new rule r_{i+1} that is not in the set of existing rules *R* could be defined, $\{r_{i+1} : r_{i+1} \notin R\}$. For example:

a. Variations of ST rules [22], e.g. r_{i+1}: *first character selection*, which selects the first character of an input value (see Table 1).
b. Rules that select specific input values, e.g. r_{i+2}: *select 0*, to test for divide by zero errors.
c. Rules to select sets of input values, e.g. r_{i+3}: *select all ASCII symbols*.
d. Rules that select Unicode characters [2], e.g. r_{i+4}: *Unicode U+00FC (ü) replacement*, which could be effective in testing programs with international character support.
e. Rules for testing programs with Graphical User Interfaces, e.g. r_{i+5}: *maximum character selection*, which could add characters to a text field until no more characters will fit. This could be useful for testing for buffer overflows.

Each rule is defined by creating a new instance of the Atomic Rules schema (see Table 1).

4.3 Creation-Based Tailoring via Selection

In *creation-based tailoring via selection*, existing rules are combined to create new rules. There are three types of tailoring within this class: all combinations, paired combinations and selective combinations. In each of these procedures, new rules are defined by creating new instances of the Atomic Rules schema.

In *all combinations*, a set of all rules $\{r_1, ..., r_n\}$ are combined, $\{r^n = <r_1, ..., r_1>, ..., <r_n, ..., r_n> : \forall r \in R\}$. However, this causes a state-space explosion of n^n combinations, where *n* is the number of rules in the generalised representation. Thus, this may be only useful for experimentally locating combinations not found through other tailoring procedures.

In *paired combinations*, each rule is paired with every other rule, where each pairing creates a new rule $\{r_{n+1} = r_i \cup r_j, \forall r \in R\}$. Some examples are:

a. r_{n+1}: *uppercase first item = BVA7: first list item selection* $\cup$ *ST7: uppercase a lowercase letter*.
b. r_{n+2}: *smallest integer replacement = EP4: integer replacement* $\cup$ *BVA2: lower boundary selection*.
c. r_{n+3}: *alphabetic letter Z or z replacement = EP6: alpha replacement* $\cup$ *BVA5: upper boundary selection*.

In *selective combinations*, rule amalgamation is based on the "gut feel" of the tester that certain combinations may cause program failure. Again, this is similar to Error Guessing. For example, if a tester suspects that a program does not place an upper limit on the number of digits that can be input into a numerical field, a new rule r_{n+4}: *largest integer/real replacement = EP4: integer replacement* $\cup$ *EP5: real replacement* $\cup$ r_{i+5}: *maximum character selection* could be defined.

Some combinations create rules that already exist, for example, *EP7: alphanumeric replacement = EP4: integer replacement* $\cup$ *EP6: alpha replacement*. Also, some rules are contradictory, for example, *EP6: alpha replacement* cannot be combined with *EP10: valid list item selection* as the first rule selects invalid data while the second selects valid data. This is similar to the identification of contradictory test frames in CPT [28]. While a complete listing of contradictory combinations is outside the scope of this paper, a list is currently under development.

Table 1: Definition of a new Atomic Rule r_{i+1}.

Attribute	Values	Explanation
Test Method	Syntax Testing	Definition of a new ST rule
Number	r_{i+1}	A unique identifier
Name	First Character Selection	Name reflects rule functionality
Description	Select the first character of an input string	Brief description of what the rule does
Source	NA	New rule, so no existing reference
Rule Type	DISR	Selects one test data value
Set Type	List or Range	Can be applied to data defined in a list or a range
Valid or Invalid	Invalid	Selects data that the program should recognise as faulty
Original Datatype	Multiple: Integer, Real, Alpha, Alphanumeric, Non Alphanumeric	For the resulting test data to be considered invalid, this rule can only be applied to datatypes of greater than one character
Test Datatype	Same as original	Does not change the datatype of the original field
Test Data Length	1	Selects one character per application, thus length is 1
# Fields Populated	1	Selects test data for one field per application
# Tests Derived	0	This is a DISR, thus it does not construct test cases

5. Preliminary Evaluation

As a preliminary proof of concept, we apply GQAS and SMT to an internet-based Foreign Exchange Calculator [34] (Figure 1). To assess their effectiveness, we compare the results of applying a set of EP and BVA Atomic Rules to those selected by a new method derived by SMT. However, as we do not have access to the program specification, we can apply GQAS to obtain possible definitions. To limit the scope of the example, only the Foreign Currency field will be tested. Settings for fields "I wish to," "Select the foreign currency" and "Select the currency type" are shown in Figure 1.

1. *Goal*: to specify the Foreign Currency field of the Foreign Exchange Calculator in order to enable black-box testing.
2. *Questions*:
   a. What is the field's datatype?
   b. What is the field's set type?
   c. Ranges: minimum and maximum values; Lists: what is the minimum and maximum length of valid data?
   d. What valid data set should the program accept, and what invalid data should it reject?
   e. Is the field mandatory?
   f. Does the field repeat? Minimum/maximum repetitions?
3. *Answers*:
   a. *Datatype*: based on experience with international money transfers, acceptable datatypes are Integer and Real (i.e. non floating-point numbers).
   b. *Set type*: based on experience with banking systems, it is reasonable to assume that the interval of allowable values is continuous, thus set type is Range.
   c. *Minimum/maximum range values*: various searches were used to discover this and we discuss them here to give the reader an understanding of the process followed. First, a search of the St George website with "international transfer" located "Foreign Exchange Services" [35], which included a telephone number. When called, the operator reported that there were no minimum or maximum limits placed on exchanges. However, through programming domain knowledge, we know unlimited input lengths can cause buffer overflow and conversion exceptions. Thus, the next search determined the financial worth of the richest person on Earth (i.e. Bill Gates), which may be a sensible value to use. According the Forbes this is US$46.5 billion [18]. However, if the top twenty-five billionaires saved their money with the same bank, their total financial worth could be more sensible. According to Forbes, this is US$496.8 billion [18]. Taking this even further, we may consider the GDP of the largest economy in the world, the USA, US$10.8 trillion [33]. These answers provide application domain data that is potentially sensible for defining this field. However, the maximum variable size of the programming language used, and combined implementation and runtime domain issues, could be considered. The application was written in JavaScript (discovered by viewing the source code), which is capable of representing numbers in the range[4] $\pm 1.7976931348623157 \times 10^{308}$ [13]. For this application, this limit would be the maximum output value when converting to a particular currency or when an input is represented internally. Thus, this figure needs to be divided by the largest possible exchange rate, which are available real-time on the Reserve Bank of Australia website [31]. Plausible values are 0.1 to 1000. Thus, sensible minimum and maximum range values could be $\pm 1.7976931348623157 \times 10^{305}$.
   d. *Valid data set*: as described in step c.
   e. *Is the field mandatory?* Yes.
   f. *Does the field repeat*? No.
4. *Specify*:<foreign_currency> ::= $[-1.7976931348623157 \times 10^{305} – 1.7976931348623157 \times 10^{305}]$

Atomic Rules from EP and BVA can now be applied to generate test data (Table 2), followed by the definition of test cases by a tailored method (Table 3). Although the input field permits more numbers to be added, for test case 15 (Table 3), an arbitrarily large number chosen to represent the maximum possible digits is 120,000.

Although floating point representations are used throughout this discussion, when providing input to the program, an integer or fixed point decimal value containing 309 digits to the left of the decimal point was used. Thus, the inputs specified by test cases 8 to 13 contain 309 digits, the first section of which is the 17 digit mantissa of the resultant value. For example, in the case of test case 9, the number input to the program is 17976931348623157 followed by 297 "9"'s.

In fact, there are two sets of values that could have been used, depending upon whether we were testing implementation domain/run-time issues (i.e. variable storage limits) or application domain issues (i.e. sensible values for maximum amounts) [30]. While it could be more sensible to derive test cases based on the latter, for the purposes of this example, we have focused on implementation domain issues.

As Table 2 and Table 3 show, the tailored method detects a suspected fault with test case 14 (see Figure 5) that is not detected by EP or BVA. Further examination revealed that inputting the string **"<>** followed by any other symbols and clicking Calculate causes the symbols to be printed to the right of the input field.

Both EP and the tailored method detect that the program does not limit input data lengths (Table 2, tests 1 and 2; Table 3, test 15), causing a suspected buffer overflow (Figure 3). Note that the resulting screen does not specify what was wrong with the input. BVA did not detect this as the exchange rate used was overestimated.

[4] For the purposes of this discussion, we only consider exponents > 0.

Table 2: Equivalence Partitioning and Boundary Value Analysis test cases for the Foreign Currency field.

#	Rule	Test Data	Result
1	EP1: < lower boundary selection	-1.797693134 8623157x10^{350}	Suspected buffer overflow (Figure 3)
2	EP2: > upper boundary selection	+1.797693134 8623157x10^{350}	Suspected buffer overflow (Figure 3)
3	EP3: lower to upper boundary selection	50000	Correct result output (Figure 2)
4	EP6: alpha replacement	g	Input rejected, validation message shown (Figure 4)
5	EP7: alphanumeric replacement	g55f	Input rejected, validation message shown (Figure 4)
6	EP8: non-alphanumeric replacement	*	Input rejected, validation message shown (Figure 4)
7	EP9/BVA9: missing value replacement		Input rejected, validation message shown (Figure 4)
8	BVA1: lower boundary – 1 selection	-1.79769313486 23158x10^{305} - 1	Correct result output (Figure 2)
9	BVA2: lower boundary selection	-1.79769313486 23158x10^{305}	Correct result output (Figure 2)
10	BVA3: lower boundary + 1 selection	-1.79769313486 23158x10^{305} + 1	Correct result output (Figure 2)
11	BVA4: upper boundary – 1 selection	1.797693134862 3158x10^{305} - 1	Correct result output (Figure 2)
12	BVA5: upper boundary selection	1.797693134862 3158x10^{305}	Correct result output (Figure 2)
13	BVA6: upper boundary + 1 selection	1.797693134862 3158x10^{305} + 1	Correct result output (Figure 2)

Table 3: Test cases of a tailored black-box method derived using SMT for the Foreign Currency field (each of these rules were defined in Section 4).

#	Rule	Test Data	Result
14	r_{i+3}: select all ASCII symbols	!@#$%^&* ()_+{}|:"<> ?[]\;',./~`	Input rejected, validation message shown (Figure 4). Symbols output to the right of the Foreign Currency Field (Figure 5)
15	r_{n+4} largest integer/real replacement	120000 9's	Suspected buffer overflow (Figure 3)
16	r_{i+4}: Unicode U+00FC (ü) replacement	ü	Input rejected, validation message shown (Figure 4)

Figure 1: Screen capture of the St George Bank's online Foreign Exchange Calculator [34].

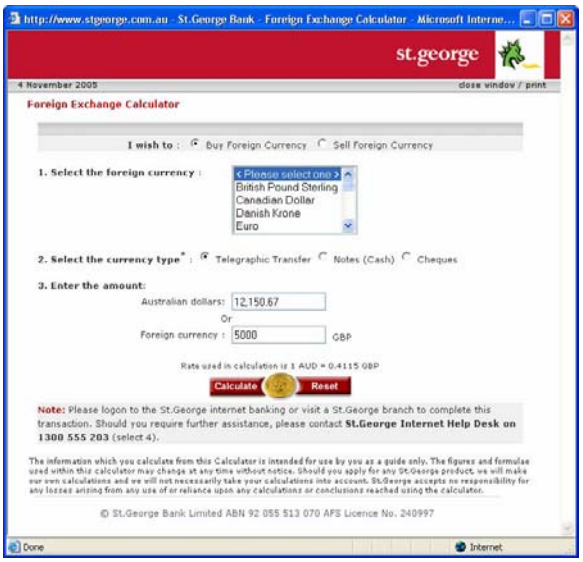


Figure 2: Result of executing the program with valid values.

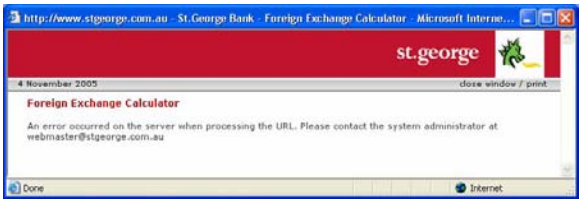


Figure 3: Result of executing the program with very large input, causing a suspected buffer overflow fault.

Figure 4: Validation message displayed when the program is executed against an invalid datatype.

Figure 5: Symbols output to the right of the Foreign Currency field when test case 14 of Table 3 is applied.

1530-0803/06 $20.00 © 2006 **IEEE**

6. Conclusions and Future Research

In this paper we presented two procedures: the first (Goal/Question/Answer/Specify, GQAS) allows testers to ensure fields are completely specified; the second (Systematic Method Tailoring, SMT) facilitates customisation of black-box methods when they are incomplete. GQAS is a necessary precondition for successful application of SMT, as without complete field specification, it is impossible to generate effective test sets for verifying program correctness. SMT has three components: selection of appropriate black-box testing method fragments, creation of new fragments when existing fragments are incomplete; and creation of new fragments as a result of synthesising existing fragments.

We evaluated these procedures by applying them to an internet-based application. First, we articulated a complete field specification using GQAS. Then, we applied SMT, EP and BVA to one field, and compared the results in terms of their error detection effectiveness. We showed that SMT identified a suspected fault that EP and BVA could not detect, improving the effectiveness of the black-box testing conducted.

We developed these procedures because black-box testing (and its teaching) should improve if it is more systematic, its results are more reproducible, and if it is less dependent on tester knowledge and experience (i.e. on Error Guessing). These procedures also facilitate the capturing of ad hoc test data selection rules used by experienced testers; these rules could be shared in the software engineering community and incorporated into software engineering education curricula.

Thus, we believe these procedures can be applied in both industry and academia. In *industry*, our procedures can be used to develop domain-specific customised test suites that better satisfy the testing requirements of each application under test. Our GQAS procedure could facilitate the integration of test engineers into the requirements elicitation process, which could assist iterative software development. These procedures also present *academia* with an opportunity to explore combinations of existing test data selection rules and derivations of new rules and new black-box methods, using a common framework and terminology.

The development of our procedures potentially raises the profile of formal black-box methods, and allows the 'craft' of software testing to be articulated. Software testers are provided with a language for describing the ways in which they generate ad hoc tests, assisting in communication within testing teams and potentially assisting in the transfer of knowledge and experience from senior to junior testers.

In future research, we will formally evaluate (via the use of industry surveys and experiments) whether the procedures presented here capture the implicit processes followed by testers when conducting Error Guessing. In addition, we will determine whether there are patterns to the types of Atomic Rules that cannot be combined, and will investigate whether there is a mathematical limit to the number of rules that can be created, such that a universal black-box method, while perhaps not practical, is theoretically possible. Finally, in larger-scale industry-based experiments, we will compare the effectiveness, usability and drawbacks of tailored black-box methods to traditional methods and will determine whether improvements can be seen in using GQAS as a requirements elicitation technique.

7. Acknowledgements

We thank the Department of Computer Science and Computer Engineering of La Trobe University for their continuing financial support of Ms. Murnane's PhD candidature. We also thank SPRIG (the Staff and PostgRaduate Information technology research Group) for their support and encouragement of research students within the Department.

8. Appendix 1

Tables 4 to 7 contain Atomic Rules for Syntax Testing (ST). A number of these overlap with Equivalence Partitioning and Boundary Value Analysis rules [24], as the methods are similar. For example, *EP4–EP8* cover rules defined in [10, 7], *BVA1–BVA6* cover definitions in [8, 22] and *EP9* and *BVA9* in [7, 10, 15, 22]. Thus, only rules unique to ST are included. Note that *Source = NA* denotes new rules not covered by existing methods.

Table 4: Atomic Rules for Syntax Testing.

Attribute	Values	Values	Values	Values	Values
Test Method	ST	ST	ST	ST	ST
Number	ST1	ST2	ST3	ST4	ST5
Name	Remove last character	Replace last character	Add extra character to end of field	Remove first character	Replace first character
Description	Remove the last character of an input string	Replace the last character of a string with an invalid value	Add an extra character to the end of a string	Remove the first character of a string	Replace the first character of a string with an invalid value
Source	[8, 22]	[22]	[7, 22]	NA	NA
Rule Type	DISR	DISR	DISR	DISR	DISR
Set Type	List or Range	List or Range	List or Range	List or Range	List or Range
Valid or Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Original Datatype	All	All	All	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original	Same as original
Test Data Length	m - 1, where m is the original field length	Same as original	m + 1, where m is the original field length	m - 1, where m is the original field length	Same as original
# Fields Populated	1	1	1	1	1
# Tests Derived	0	0	0	0	0

Table 5: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values	Values	Values
Test Method	ST	ST	ST	ST	ST
Number	ST6	ST7	ST8	ST9	ST10
Name	Add extra character to start of field	Uppercase a lowercase letter	Lowercase an uppercase letter	Null all input	Duplicate Field
Description	Add an extra character to the start of a string	Change the case of a uppercase letter to lowercase	Change the case of a lowercase letter to uppercase	Construct a test case that is empty	Construct a test case that has one field duplicated (all other fields are assigned their nominal value)
Source	NA	[22]	[22]	[8]	[10, 7]
Rule Type	DISR	DISR	DISR	TCCR	TCCR
Set Type	List or Range	List or Range	List or Range	List or Range	All
Valid or Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
Original Datatype	All	Alpha	Alpha	All	All
Test Datatype	Same as original	Same as original	Same as original	Null	Same as original
Test Data Length	m + 1, where m is the original field length	Same as original	Same as original	0	Same as original
# Fields Populated	1	1	1	n, where n is the number of fields in the specification	n, where n is the number of fields in the specification
# Tests Derived	0	0	0	1	1

Table 6: Atomic Rules for Syntax Testing (continued).

Attribute	Values	Values	Values	Values	Values
Test Method	ST	ST	ST	ST	ST
Number	ST11	ST12	ST13	ST14	ST15
Name	Add Additional Field	Select Each List Alternative	Select All List Alternatives	Select All List Alternatives in Reverse Order	Reference Replacement
Description	Construct a test case that contains a new field (contents of new field must be defined, possibly using GQAS)	For a specification that includes a list, create a set of test cases in which each alternative in the list is selected once (all other fields are assigned their nominal value)	Select every alternative from a list in the one test	Select every alternative from a list in the reverse order in the one test	For a non-terminal fields that references another terminal, create a test case in which the non-terminal references itself
Source	[10, 7]	[22]	[22]	[22]	[22]
Rule Type	TCCR	TCCR	DISR	DISR	TCCR
Set Type	All	All	All	All	All
Valid or Invalid	Invalid	Valid	Invalid	Invalid	Invalid
Original Datatype	All	All	All	All	All
Test Datatype	Same as original	Same as original	Same as original	Same as original	Same as original
Test Data Length	Same as original	Same as original	Same as original	Same as original	Same as original
# Fields Populated	n, where n is the number of fields in the specification	n, where n is the number of fields in the specification	1	1	n, where n is the number of fields in the specification
# Tests Derived	1	p, where p is the number of alternatives	0	0	q, where q is the number of references to other non-terminals in the specification

Table 7: Atomic Rules for Syntax Testing (continued).

Attribute	Values
Test Method	ST
Number	ST16
Name	Syntax Cover
Description	Construct a set of test cases which link-cover the syntax graph of the specification under test
Source	[7, 15]
Rule Type	TCCR
Set Type	All
Valid or Invalid	Valid
Original Datatype	All
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	n, where n is the number of specification fields
# Tests Derived	r, where r is the number of basis paths [29]

References

[1] Abbott, J. *Software Testing Techniques*. NCC Publications, Manchester, England, UK, 1986.

[2] Aliprand, J., Allen, J., Becker, J., Davis, M., Everson, M., Freytag, A., Jenkins, J., Ksar, M., McGowan, R., Muller, E., Moore, L., Suignard, M., and Whistler, K. (editors). *The Unicode Standard Version 4.0*. Addison-Wesley, USA, 2003.

[3] Basili, V. R. Software Modeling and Measurement: The Goal/Question/Metric Paradigm. *Technical Report UMIACS-TR-92-96*, University of Maryland, USA, pp. 1-24, Sept 1992.

[4] Basili, V. R., Gainluigi, C., Rombach, H. D. The Goal Question Metric Approach. *The Encyclopaedia of Software Engineering*, 1:528-532, Wiley & Sons Inc., 1994.

[5] Basili, V. R., and Selby, R. W. Data Collection and Analysis in Software Research and Management. *Proceedings of the American Statistical Association of Biomeasure Society Joint Statistical Meetings*, Philadelphia, USA, August 1984.

[6] Basili, V., and Weiss, D. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10(6):728-738, November 1984.

[7] Beizer, B. *Black Box Testing. Techniques for Functional Testing of Software and Systems*. John Wiley & Sons Inc., USA, 1995.

[8] Beizer, B. *Software Testing Techniques*. Von Nostrand Reinhold, New York, USA, 1990.

[9] Bonifati, A., Cattaneo, F., Ceri, S., Fuggetta, A., and Paraboschi, S. Designing Data Marts for Data Warehouses. *ACM Transactions on Software Engineering and Methodology*, 10(4), pp. 452-483, October 2001.

[10] British Standards Institute. *Software Testing: Software Component Testing Standard*. British Computer Society, 1998.

[11] Craig, R.D., and Jaskiel, S.P. *Systematic Software Testing*. Artech House Publishers, USA, 2002.

[12] Dubois, E., Yu, E., and Petit, M. From Early to Late Formal Requirements: a Process-Control Case Study. *Proceedings of the 9th International Workshop on Software Specification and Design*, pp. 34-42, IEEE, April 1998.

[13] Flanagan, D. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., USA, 4th edition, 2002.

[14] Grindal, M., Lindström, B., Offutt, A. J., and Andler, S. F. An Evaluation of Combination Strategies for Test Case Selection. *Technical Report HS-IDA-TR-03-001*, Department of Computer Science, University of Skövde, pp. 1-74, 2004.

[15] Hetzel, W. *The Complete Guide to Software Testing*. QED Information Sciences Inc, USA, 1988.

[16] Jorgensen, P. *Software Testing: A Craftsman's Approach*. Department of Computer Science and Information Systems, Grand State University, Allendale, CRC Press, 1995.

[17] Kaner, C., Bach, J., and Pettichord, B. *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons, Inc., New York, USA, 2002.

[18] Kroll, L., and Gildman, L. (editors). The World's Billionaires, http://www.forbes.com/billionaires/, date last updated: 3/10/05, date last accessed: 4/11/05.

[19] Lamsweerde, A. van, and Willemet, L. Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089-1114, December 1998.

[20] Letier, E., and Lamsweerde, A. van. Reasoning about Partial Goal Satisfaction for Requirements and Design Engineering. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press, USA, pp. 53-62, 2004.

[21] Mandl, R. Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing. *Communications of the ACM*, 28(10):1054-1058, Oct 1985.

[22] Marick, B. *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*. Prentice Hall PTR, USA, 1995.

[23] Mosley, Daniel J. *The Handbook of MIS Application Software Testing. Methods, Techniques, and Tools for Assuring Quality Through Testing*. Prentice-Hall, Inc., USA, 1993.

[24] Murnane, T., Hall, R., and Reed, K. Towards Describing Black-Box Testing Methods as Atomic Rules. *Proceedings of the 29h Annual International Computer Software and Applications Conference,* Scotland, pp. 437-442, July 2005.

[25] Murnane, T., and Reed, K. On the Effectiveness of Mutation Analysis as a Black-box Testing Technique. *Proceedings of the 2001 Australian Software Engineering Conference*, Australia, IEEE, pp. 12-20, 2001.

[26] Myers, G. *The Art of Software Testing*. John Wiley & Sons Inc, USA, 1979.

[27] Ng, S., Murnane, T., Reed, K., Grant, D., and Chen, T. Y. A Preliminary Survey of Software Testing in Australia. *Proceedings of the 2004 Australian Software Engineering Conference*, Australia, IEEE, pp. 116-125, April 2004.

[28] Ostrand, T. J., and Balcer, M. J. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, 31(6):676-686, June 1988.

[29] Pressman, R. S. *Software Engineering: A Practitioners Approach*. McGraw-Hill, Singapore, 1992.

[30] Reed, K. An *Outline of a Knowledge Based Approach to Software Project Planning*. Position paper, Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Australia, 1990.

[31] Reserve Bank of Australia website, Exchange Rates. http://www.rba.gov.au/Statistics/exchange_rates.html, date last updated: 4/11/05, date last accessed: 4/11/05.

[32] Sommerville, I., Sawyer, P., and Viller, S. Viewpoints for Requirements Elicitation: a Practical Approach. *Proceedings of the 3rd International Conference on Requirements Engineering* (ICRE '98), Colorado Springs, USA, IEEE, pp. 74-81, 1998.

[33] Special Broadcasting Service. *World Guide: The Complete Fact File on Every Country*. Hardie Grant Books, 2003.

[34] St George Bank. St George Bank Calculators, http://www.stgeorge.com.au/calculators/default.asp?orc=business, date last updated: unknown, date last accessed: 4/11/05.

[35] St George Bank. Foreign Exchange Services, http://www.stgeorge.com.au/smallbus/intern_soln/foreign_xservice/default.asp?orc=business, date last updated: unknown, date last accessed: 4/11/05.

On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis

Tafline Murnane[1], Karl Reed and Richard Hall
Department of Computer Science and Computer Engineering
La Trobe University, Bundoora, Australia
tpmurnane@students.latrobe.edu.au, k.reed@latrobe.edu.au, richard.hall@latrobe.edu.au

Abstract

Currently, Equivalence Partitioning and Boundary Value Analysis are taught at La Trobe University using Myers' original representation of these black-box testing methods. We previously proposed an alternative representation called Atomic Rules. In this paper we present the statistical results of two similar experiments that examine which of these approaches enable students to write more complete and correct black-box test sets and which approach students prefer to use. We compare the results of these experiments and discuss how the results could change the teaching of black-box testing methods at La Trobe University and in industry.

1. Introduction

In this paper we present the results of two similar experiments in which two groups of novice software testers were exposed to two different representations of Equivalence Partitioning (EP) and Boundary Value Analysis (BVA): Myers' original definition [19] and the corresponding Atomic Rules representation [17]. The aim was to determine which representation enabled the testers to write more complete and correct black-box test sets. The testers who participated in the experiment were third and fourth year students enrolled in a software testing subject at La Trobe University. Thirty-two students participated in 2004 and forty in 2005.

Each group was given a two-hour lecture on one of the representations (Figure 1). During a subsequent tutorial, students were tested on their comprehension of EP and BVA by deriving black-box test cases from a fictional specification. To ensure every student had equal opportunity to learn the two representations, the groups were subsequently swapped and the process repeated. An Initial Questionnaire was used in the first lecture to collect data on student's current understanding of black-box testing methods and on their previous programming and testing experience. A Reflect and Review Questionnaire was used in the final lecture to collect data on student's initial and final understanding of EP and BVA and on their preferred method representation.

[1] Ms. Murnane is also currently working as a Software Test Consultant for K. J. Ross & Associates in Melbourne, Australia.

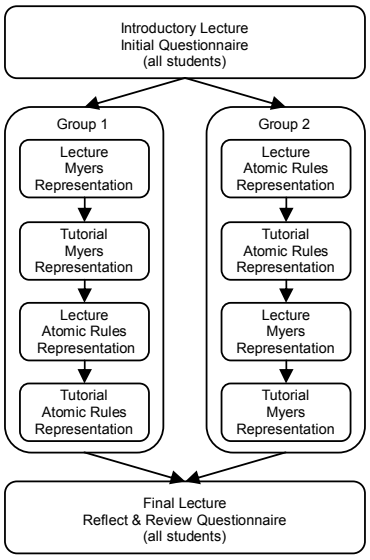


Figure 1: The experiment process.

While the aims of the two experiments and materials presented during lectures were the same for both years, three changes were made in 2005 that were significant enough for it to be considered to be a different experiment. As each group's lecture took place at the same time in different locations, two lecturers were required. In 2004, the first author of this paper taught Atomic Rules while the second taught Myers. Then, to eliminate the extraneous variable of student preference for lecturer, the lecturers were swapped in 2005. Also, the 2004 results suggested that students could handle more challenging specifications during tutorials. Thus, longer and more complex specifications were used in 2005. Lastly, to ensure students had enough time to complete their work, tutorials were increased from one hour in 2004 to two hours in 2005. To avoid conflict with other university classes or commitments (a factor which could impact student performance [6]), all work for this part of the experiment was completed in class.

These experiments are part of a larger project which explores the learnability, usability and effectiveness of black-box testing methods. In this paper we investigate representation learnability, which we define to be the ease at which a novice can gain knowledge of a

Proceedings of the 2007 Australian Software Engineering Conference (ASWEC'07)
0-7695-2778-7/07 $20.00 © 2007 IEEE

particular concept, and usability, which we define to be the level of satisfaction a tester feels when using a particular representation. The larger project will involve an industry-based case study in which we will compare the effectiveness of Atomic Rules and Systematic Method Tailoring (which enables new Atomic Rules and new black-box methods to be defined [18]) to the black-box methods used by professional testers. We will also determine whether industry testers use ad hoc or exploratory test case selection rules that are not covered by existing black-box methods, and if so, will determine whether they can be described as Atomic Rules.

The remainder of this paper is structured as follows. We present a brief overview of Atomic Rules in Section 2. We discuss our experiment design in Section 3, including hypotheses, group allocation, threats to validity and the specifications used during tutorials. A preliminary analysis of the 2004 Reflect and Review questionnaire was published in [17]; in Section 4 we present the full experiment results, including data collected on other questionnaires and during tutorials. These results are discussed in Section 5. Finally, we present out conclusions and future work in Section 6.

2. Overview of the Atomic Rules Approach

The Atomic Rules approach decomposes black-box testing methods into individual elements for partitioning a program's input and output domains, selecting test data from each partition, optionally mutating the selected data values, and constructing test cases [17]. The aim of developing this generalised representation was to make black-box methods easier to learn and use by describing them more precisely. It resolves a number of consistency problems inherent in the original methods, including ambiguity issues that could cause testers using the same methods to produce vastly dissimilar test sets, and the problem of numerous versions of each method existing in the literature [17]. A characterisation schema was developed to give the methods a standard representation (Table 1) and the following four-step procedure[2] was produced to standardise and describe the black-box test case selection process:

1. Select valid and invalid partitions for each input and/or output field by applying a *Data-Set Selection Rule* (DSSR) to each field.
2. Select at least one individual data value from each partition chosen step in 1 by applying a *Data-Item Selection Rule* (DISR) to each partition.
3. Mutate the data values selected in step 2 by applying a *Data-Item Manipulation Rule* (DIMR) to each data value.
4. Select various combinations of the data values chosen in steps 2 and 3 by applying a *Test Case Construction Rule* (TCCR) to create test cases.

When applied with the Atomic Rules from a particular black-box method or a variety of methods, this four-step test case selection procedure can be used to construct black-box test cases in the usual way [17].

[2] Although step 3 was defined after the initial publication of the Atomic Rules approach [17] and after the experiment, it is included here for completeness.

For example, consider the field *<age> ::= [0 – 150]*. An EP DSSR that selects all values between the lower and upper boundaries of range-related fields (Table 1) could be applied, selecting the valid equivalence class *[0 – 150]*. A BVA DISR that selects the upper boundary value of an equivalence class could be applied to this class, selecting the valid data value *150*. A Syntax Testing DIMR that adds an extra character (e.g. the letter *a*) to the start of a data value could be applied, selecting the invalid data value *a150*. If this was included in an input string with another field such as *Surname ::= [A-Z, a-z, -]$^{1\text{-}100}$*, an EP TCCR that selects a test case containing one invalid value per test could be applied, which could result in the invalid test case *Smith a150*.

Table 1: Example of an Atomic Rule [15].

Attribute	Values
Test Method	Equivalence Partitioning
Number	EP3
Name	Lower to Upper Boundary Selection
Description	Select an equivalence class containing all values that lie between the lower and upper boundaries of a field
Source	[19]
Rule Type	DSSR
Set Type	Range
Valid or Invalid	Valid
Original Datatype	Integer, Real, Alpha, Non-Alphanumeric
Test Datatype	Same as original
Test Data Length	Same as original
# Fields Populated	1
# Tests Derived	0

3. Experiment Design

The primary independent variable used in these experiments was the first black-box testing method representation learnt. The approach to manipulating the independent variable was the type technique [13] whereby the type of variable presented is varied over two separate treatments. In the following, we describe the experiment hypotheses, group allocation and threats to validity that relate to this experiment.

3.1 Hypotheses

To compare the learnability and usability of the two representations, we measured the completeness (H_{11}) and correctness (H_{13}) of the black-box test sets derived, the efficiency of derivation (H_{12}), the questions asked during derivation (H_{14}) and the level of satisfaction the students experienced while deriving test cases (H_{15}). Thus, the following hypotheses were defined.

Completeness:

H_{01}: The completeness of the black-box test set derived by novice testers is independent of the approach used.

H_{11}: Novice testers using Atomic Rules derive a more complete test set compared to those using Myers' approach.

Efficiency:

H_{02}: The efficiency of black-box test case derivation by novice testers is independent of the approach used.

H_{12}: Novice testers using Atomic Rules derive test cases more efficiently compared to those using Myers' approach.

Errors made (Correctness):

H_{03}: The number of errors made by novice testers during black-box test case derivation is independent of the approach used.

H_{13}: Novice testers using Atomic Rules make fewer errors during test case derivation compared to those using Myers' approach.

Questions asked:

H_{04}: The number of questions asked by novice testers during black-box test case derivation is independent of the approach used.

H_{14}: Novice testers using Atomic Rules ask fewer questions compared to those using Myers' approach.

User satisfaction:

H_{05}: There is no difference between the usability of Atomic Rules or Myers' approach when used by novice testers in the derivation of black-box test cases.

H_{15}: Novice testers find Atomic Rules simpler to use than Myers' approach when deriving black-box test cases.

These hypotheses are similar to several that were used in an experiment that compared how effective novice testers were in selecting appropriate software testing methods when using descriptions of the methods from a characterisation schema versus novices who used textbook descriptions of the methods [23].

3.2 Group Allocation

The participants in our experiments were divided into two comparison groups [13]. To provide repetition, each group was divided into two subgroups, with each deriving test cases from a different specification (Table 2). A discussion of how group allocation affects validity is given in Section 3.4.

Table 2: Number of participants per group and subgroup.

Year	Group	Myers	Atomic Rules
2004	Subgroup 1	13	8
	Subgroup 2	5	6
	Total	**18**	**14**
2005	Subgroup 1	10	8
	Subgroup 2	10	12
	Total	**20**	**20**

3.3 Input Data Specifications

The main requirement that was placed on the specifications that were used during tutorials was that they had to include at least one numerical range, one list of values and a number of different datatypes; e.g. alphas, numbers and symbols. This allowed students to derive tests for a 'base' set of set types and datatypes.

One of the 2004 specifications was for a fictional Personal Details Recording System (Figure 2), while one of the 2005 specifications was for a Patient Record System (Figure 3). Both were written in a semi-formal notation and contained input fields defined using a combination of Backus-Naur Form and natural language.

There were two primary differences between these specifications: length and complexity. In 2004, the top level non-terminal node contained five fields (including two spaces) whereas the corresponding node in 2005 contained fourteen fields. Thus, the 2005 specifications were substantially longer. Also, the 2005 specifications contained a recursive field definition, which made test case derivation more challenging (e.g. <level_digits> field). To compensate for this change, the tutorials were extended from one hour in 2004 to two hours in 2005.

Specification:

<personal_details> ::= <id_number> <s> <surname> <s> <gender>

<id_number> ::= [100 – 999]

<surname> ::= 1 to 100 characters from the sets alpha and non-alphanumeric (i.e. letters and/or symbols)

<gender> ::= {Male | Female}

<s> ::= one to seven single spaces

Example Record:

555 Smith Male

Figure 2: Specification 1 – Personal Details System.

Specification:

<patient_record> ::=
<name>^…"<ailment>"^…<floor_no>,^…<building>,^…<patient_no>

<name> ::= 1 to 100 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)

<ailment> ::= 1 to 150 characters from sets alpha and non-alphanumeric (i.e. letters and symbols)

<floor_no> ::= <level_no> ^ Floor

<level_no> := <level_digits><level_postfix>

<building> ::= {Fredrick Building | John-Scott Memorial Ward | Mary House | Norman Building | … | Zane Square Building | Zoo Ward}

<patient_no> ::= <d><d><d><d>

<d> ::= [0 – 9]

<level_digits> ::= <d> | <level_digits> <d>

<level_postfix> ::= {nd | rd | st | th}

^ ::= one space

^… ::= one or more spaces

Example Record:

Joe Hamish Bloggs "Viral pneumonia, ear infection, and lower abdomen pain" 5th Floor, John-Scott Memorial Ward, 1234

Figure 3: Specification 2 – Patient Record System.

3.4 Threats to Validity

In this section we explore threats to internal and external validity [13] that apply to our experiments.

Internal Threats to Validity

History. Experiment outcomes can be biased by time lapses between the application of the treatment variable and measurement of the dependent variable or between pre-test and post-test measurements [8] or if discussions take place between groups during that time [9]. This posed a minimal threat as treatment took place during a lecture that was between two hours and three days prior to measurement. To combat this threat, participants were asked not to discuss the experiment with each other until after the final lecture. As there was no assignment or exam during the experiment, we did not expect students to have a great need to hold discussions during that time.

Maturation. Changes or differences in a participant's internal condition (e.g. age, hunger, fatigue, boredom) [8], knowledge level [9], lecturer preference, or

enthusiasm [23] can bias results. For example, students who are excited about being involved in an experiment on a new technique may work harder on that technique. To ensure this did not bias results, Myers' representation was referred to as *Model 1* and Atomic Rules as *Model 2* and students were not told which was new until after the final lecture. Also, students were informed that there would be a gift for every member of the class at the end of the experiment whether they chose to participate or not, which we hoped would compensate them for any disruption they may have experienced during the experiment[3]. To combat boredom, students were reminded that the work they completed during tutorials would prepare them for their assignments and exam and also for future work in industry. As tutorial attendance was not compulsory, bored students could choose not to attend class but were informed that the subject was an important part of their courses. As the experiments were run over six separate classes, it was hoped that fatigue and hunger did not affect results. Selecting students from the same year levels should have negated the knowledge threat [9]. To mitigate the potential lecturer preference bias threat, the lecturers were swapped in 2005.

Instrumentation. This threat relates to research observers becoming accustomed to experiment materials or increasing their experience in measuring data [9]. To ensure the same standards were followed throughout analysis, standard measurement scales and analysis processes were followed. Also, to standardise analysis, one person was responsible for all data analysis.

Selection. Random group allocation can be used to mitigate the threat that the groups will be biased; e.g. if one group has a higher mean intelligence level than the other [9]. Conversely, if participants allocate themselves to groups, then the sample within each group is voluntary, not random, allocation [4]. While random allocation was achieved in 2005 by drawing participant names out of a hat, it was not achieved in 2004 due to a timetabling problem, which resulted in students allocating themselves to groups according to their chosen tutorial day/time. However, an analysis of the average grade in each group for that subject revealed that there was no significant difference between the two groups (see Section 4). Thus, this threat should not have biased experiment results.

Testing. Bias can occur if participants are given the same test more than once and they become familiar with the types of responses required [9]. Although our participants derived test cases for the two representations over two weeks, we have not included the results of the second week in our analysis as it would not measure their understanding of the representation learnt. Rather, it would test how well they adjusted to learning a second representation. We may analyse this statistic in future, as it may be useful as a preliminary assessment of whether industry testers would adjust to using Atomic Rules after having used different approaches as part of their jobs.

[3] A gift of chocolate, which was allowed by the university's ethics committee, was given to all students as our way of thanking them.

Reliability. This relates to the consistency of results being obtained from the same person with the same or equivalent tests on different occasions, allowing an error of measurement to be calculated [1]. The simplest approach is to repeat the experiment on two separate occasions, where the error of measurement is a reliability coefficient which is the correlation between the two scores for each individual [1]. Since our experiments took place during university semesters, there was not enough time to repeat the same test twice. However, within each group, the same test was repeated across two subgroups. This allowed us to test experiment reliability.

Population and sample. Validity can be affected if the sample is not representative of the entire population [10]. Convenience sampling was the method of recruitment in our experiments, where participants were selected as they were easily accessible [10]. We recognize that our samples are not representative of all novice testers, specifications or black-box methods; e.g. approaches such as Syntax Testing were not covered. Thus, our results are considered to be indicative, not conclusive.

Threats to internal validity which were not applicable to our experiments included diffusion of treatments, compensatory equalization, compensatory rivalry and resentful demoralization, as these only applicable when using control groups [9]. We could not use control groups in our experiments, as students in those groups would have been disadvantaged in their assignment and exam as a result of not learning the two representations.

Also, in experiments involving students, participants sometimes work on tasks at home; thus copying is a threat [23]. However, in our case, all tasks for the main body of the experiment were completed in class and every second student was given a different specification during tutorials so they could not copy from each other.

In addition, a bias can exist if participants do not follow the processes and procedures of the techniques prescribed [23]. We asked students to show all workings while deriving test cases as we were interested in discovering when they did not follow the method procedures, as this may identify method ambiguity.

External Threats to Validity

Language. Participants may be disadvantaged if experiment materials are not written in their native language [23]. Some international students were involved in our experiments and all materials were written in English. However, since the students were enrolled at an English-speaking university, it was expected that they would be able to understand the language used, and if not, that they would ask a question.

Interaction of setting and treatment. This relates to the ability to generalise experiment findings across other environmental settings [9], which in our case is determining whether our results are applicable to industry professionals. This threat is not applicable in our experiment as industry testers can be considered to be expert software testers, while our experiment was aimed at novices.

Interaction of history and treatment. This threat relates to the ability to generalise research outcomes to the past and future; e.g. if a classroom experiment runs during the main semester, the outcomes may be different than if it were conducted over the summer break, due to different types of students being enrolled at that time [9]. One way of resolving this threat is to replicate the experiment at a different time of year. Our experiments were run during the same semester over two years and we do not anticipate having the resources to repeat the experiments over summer as very few third and fourth year subjects run during that time at La Trobe University and there have never been any official student requests to do so in this software testing subject.

4. Results

In this section we present data on the demographic of students involved in the experiments, followed by results for each of the five hypotheses. These results are then discussed in Section 5. Note that one-tailed tests were used in all significance tests.

Demographic

In the initial questionnaire, students were asked about their prior software testing and industry experience. Twenty-six out of thirty-two students completed this questionnaire in 2004 (81.25%), while thirty-seven out of forty completed it in 2005 (92.5%).

Many students reported having prior experience with software testing during university lectures and/or university assignments (Table 3). However, while 19.2% in 2004 and 13.5% in 2005 reported having no prior experience with software testing methods, 73.1% in 2004 and only 48.6% in 2005 stated that they received such experience through university lectures.

Table 3: Prior software testing experience.

Software Testing Experience	2004 (n = 26)	2005 (n = 37)
None	19.2%	13.5%
University Lectures	73.1%	48.6%
University Assignments	61.5%	62.2%
As a Tutor	0%	0%
Other	11.5%	16.2%

We also found that 84.6% of the 2004 group had prior experience with black-box testing methods, compared to only 54.1% in 2005 (Table 4). These statistics suggests that there may have been a decrease in the amount of software testing training that was given to students in the 2005 group in the earlier years of their degrees.

Table 4: Prior experience with black-box testing methods.

Ever used any black-box testing methods?	2004 (n = 26)	2005 (n = 37)
Yes	84.6%	54.1%
No	15.4%	45.9%

Participants rated their level of experience with the following black-box methods: Boundary Value Analysis (BVA), Cause-Effect Graphing (CEG), Decision Tables (DT), Equivalence Partitioning (EP), Orthogonal Array Testing (OAT), Random Testing (RT), Specification-Based Mutation Testing (SBMT), State-Transition Diagram Testing (STT), Syntax Testing (ST) and Worst Case Testing (WCT). Although statistics were only required for EP and BVA, we enquired about nine other methods to obtain an overall picture of the class's current black-box testing experience. Students rated their understanding using a Likert scale of: 1 = none, 2 = basic, 3 = intermediate, 4 = advanced and 5 = expert (Tables 5 and 6). With the exception of BVA and RT in 2004, the majority of students reported having limited amounts of experience with black-box testing methods.

Table 5: Participants initial understanding of black-box testing methods in 2004 (n = 26).

	Black-Box Testing Methods										
	BVA	CEG	DT	EP	EG	OAT	RT	SBMT	STT	ST	WCT
Rating	Percentages (%)										
None	19	96	54	65	73	100	46	96	73	54	65
Basic	15	0	8	4	8	0	15	0	8	8	15
Intermediate	31	4	27	8	12	0	31	4	12	23	12
Advanced	23	0	8	23	4	0	4	0	8	15	8
Expert	12	0	4	0	4	0	4	0	0	0	0

Table 6: Participants initial understanding of black-box testing methods in 2005 (n = 37).

	Black-Box Testing Methods										
	BVA	CEG	DT	EP	EG	OAT	RT	SBMT	STT	ST	WCT
Rating	Percentages (%)										
None	65	95	76	84	81	97	73	97	87	78	90
Basic	16	0	5	5	8	0	19	0	8	14	5
Intermediate	16	5	16	11	11	3	8	3	5	8	5
Advanced	3	0	3	0	0	0	0	0	0	0	0
Expert	0	0	0	0	0	0	0	0	0	0	0

Interestingly, very few students reported having any prior experience working in industry (Table 8).

Table 7: Prior industry experience.

Position in Industry	2004 (n = 26)	2005 (n = 37)
Project Manager	3.8%	0%
Technical Team Leader	3.8%	0%
Business Analyst	0%	0%
Programmer	3.8%	2.7%
Analyst	3.8%	2.7%
Test Team Leader	3.8%	0%
Test Team Member	0%	2.7%
Other	0%	5.4%

A comparison of the mean overall grade of each group in the subject showed that there was no significant difference between the groups in 2004 or 2005 (Table 7).

Table 8: Comparison of overall grades for each group.

Year	N	Approach	Mean	Std Dev	t-test
2004	18	Myers	66.17	20.88	$t(38) = .428$, $p = .336$
	14	Atomic Rules	73.29	14.56	
2005	20	Myers	68.1	19.49	$t(30) = -1.08$, $p = .143$
	20	Atomic Rules	65.8	14.1	

Completeness (H_{01}/H_{11})

To assess completeness we compared the percentage of EP equivalence classes, BVA boundary values and EP and BVA test cases derived correctly by each group.

In 2004, a t-test showed a significant difference between the groups for EP equivalence class and test case derivation, where the mean was higher for the Atomic Rules group (Tables 9, 10). According to Cohen's Effect Size [16], these relationships were strong. The 2004 BVA results were inconclusive (Tables 11, 12). Conversely, the mean EP class (Tables 9, 10) and BVA boundary value (Table 11) coverage was higher for Myers' group in 2005 and Cohen's Effect Size showed moderate to strong relationships. The results for BVA test cases in 2005 were inconclusive (Table 12).

Interestingly, the mean EP and BVA coverage achieved by Myers' group in 2004 and 2005 was relatively the same in both years (Tables 9 to 12).

Table 9: Percentage of coverage of EP equivalence classes.

Year	N	Approach	Mean	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	49.76	16.95	$t(30) = -3.82$, $p = .0003$	1.35 strong
	14	Atomic Rules	78.86	26.10		
2005	20	Myers	48.61	16.13	$t(38) = 4.815$, $p < .001$	1.53 strong
	20	Atomic Rules	26.54	12.65		

Table 10: Percentage of coverage of EP test cases.

Year	N	Approach	Mean	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	36.23	23.29	$t(30) = -4.87$, $p = .0002$	1.73 strong
	14	Atomic Rules	78.86	26.10		
2005	20	Myers	38.94	20.92	$t(38) = 2.649$, $p = .006$	0.85 moderate
	20	Atomic Rules	23.65	15.11		

Table 11: Percentage of coverage of BVA boundary values.

Year	N	Approach	Mean	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	18.88	19.98	$t(30) = .58$, $p = .28$	NA
	14	Atomic Rules	23.81	26.82		
2005	20	Myers	26.00	19.51	$t(38) = 2.776$, $p = .004$	.90 moderate
	20	Atomic Rules	11.52	12.80		

Table 12: Percentage of coverage of BVA test cases.

Year	N	Approach	Mean	Std Dev	t-test	Cohen's Effect Size
2004	18	Myers	14.88	18.83	$t(40) = -.39$, $p = .35$	NA
	14	Atomic Rules	18.81	26.22		
2005	20	Myers	10.22	16.92	$t(38) = .141$, $p = .445$	NA
	20	Atomic Rules	9.56	12.63		

In addition to data collected during tutorials, students in 2005 were asked to derive black-box test cases in their class assignment using one of the representations (Table 13). We found that significantly more students chose to use the Atomic Rules approach in that year.

Table 13: Representation used in the assignment (n = 38).

Year	Approach	Used in Assignment	Chi-Square
2005	Myers	27.5%	$\chi^2(1, N = 38) = 6.737$, $p = .009$
	Atomic Rules	67.5%	

In addition, the average assignment mark in 2005 for students who used the Atomic Rules representation in their assignment was significantly higher (Table 14).

Table 14: Average mark achieved in the assignment (n = 38).

Year	Approach	Mean Mark	t-test
2005	Myers	67.91	$t(36) = -1.93$, $p = .03$
	Atomic Rules	85.52	

Efficiency (H_{02}/H_{12})

We calculated the speed at which students completed test case derivation by counting the number of students who ran out of time before finishing test case derivation during tutorials. We found significantly more students in the Atomic Rules group ran out of time before completing their work in both 2004 and 2005 (Table 15).

Table 15: Number of participants who ran out of time.

Year	Approach	N	Out of Time (Count)	Out of Time (Percent)	Test of Two Proportions
2004	Myers	18	5	27.77%	$\delta = -.4366$, $z = -2.45$, $p = .007$
	Atomic Rules	14	10	71.43%	
2005	Myers	20	7	35%	$\delta = -.6$, $z = -3.98$, $p < .001$
	Atomic Rules	20	19	95%	

Correctness (H_{03}/H_{13})

To assess correctness, we counted the number of errors that students made during tutorials. We found that significantly fewer errors were made by students in the Atomic Rules group during EP equivalence class derivation in 2004 (Table 16). A similar result was seen in BVA boundary value definition in 2004, although the result was just outside the 95% confidence interval (Table 18). However, no significant difference was found between the groups during BVA or EP test case derivation in 2004 (Tables 17, 19) or during EP and BVA derivation in 2005 (Tables 16 to 19).

Table 16: Errors made in EP equivalence class derivation.

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	20.81	374.50	$U = 48.5$, $p = .001$
	14	Atomic Rules	10.96	153.50	
2005	20	Myers	21.65	433	$U = 177$, $p = .274$
	20	Atomic Rules	19.35	387	

Table 17: Errors made in EP test case derivation.

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	15.56	280	$U = 109$, $p = .245$
	14	Atomic Rules	17.71	248	
2005	20	Myers	21.55	431	$U = 179$, $p = .292$
	20	Atomic Rules	19.45	389	

Table 18: Errors made in BVA boundary value derivation.

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	18.5	333	U = 90, p = .0675
	14	Atomic Rules	13.93	195	
2005	20	Myers	22.15	443	U = 167, p = .192
	20	Atomic Rules	18.85	377	

Table 19: Errors made in BVA test case derivation.

Year	N	Approach	Mean Rank	Sum of Ranks	Mann-Whitney U
2004	18	Myers	17	306	U = 117, p = .349
	14	Atomic Rules	15.86	222	
2005	20	Myers	20.70	414	U = 196, p = .463
	20	Atomic Rules	20.30	406	

Questions Asked (H_{04}/H_{14})

Participants were asked to document the questions they asked during tutorials. However, we found that only three students in 2004 and no students in 2005 recorded questions. Possible reasons could be that students:

1. were reluctant to ask questions,
2. did not have enough time to record questions, or
3. had a sound understanding of the methods taught.

Although we hope the third option was the case, we do not have enough data to clarify this. In future experiments we could ask students on a questionnaire whether they recorded any questions, and if not, why.

User Satisfaction (H_{05}/H_{15})

To assess user satisfaction, students completed a Reflect and Review Questionnaire. Thirty-two students completed this questionnaire in 2004 (100% of the class) and twenty-eight in 2005 (70% of the class).

In 2004, students were asked which model they would prefer to use in future and this was compared to the model they learn first (Table 20) [17]. A chi-square test indicated that significantly more students would prefer to use the Atomic Rules approach in future.

Table 20: Approach students leant first versus approach they indicated they would use in future (n = 32).

Year	Approach	Leant First	Use in Future	Chi-Square
2004	Myers	61%	9%	$\chi^2(1, N = 32) = 21.16$, $p < .001$
	Atomic Rules	39%	91%	

In 2005 we posed a slightly different question. Students were asked to rate the likelihood that they would use the models in future (Table 21) using a Likert scale of: 1 = very unlikely, 2 = somewhat unlikely, 3 = neither likely nor unlikely, 4 = somewhat likely, 5 = very likely. However, the mean response was relatively even for both groups and no significant difference was found.

Table 21: Likelihood of using approaches in future (n = 28).

Year	Approach	Model Learn t First	Model use in future (mean)		t-test
			Myers	Atomic Rules	
2005	Myers	46.43%	3.46	3.47	$t(26) = -.01$, $p = .307$
	Atomic Rules	53.57%	3.23	3.73	$t(26) = -1.12$, $p = .445$

Students also rated their understanding of the two representations using a Likert scale of: 1 = very poor, 2 = poor, 3 = average, 4 = good, 5 = very good, 6 = excellent. In both years, students reported that their understanding of EP and BVA had improved by the end of the experiment (Tables 22 and 23, columns 2-5).

In 2004, 57% of students rated their understanding of Myers' representation as below-average, whereas 100% rated their understanding of Atomic Rules as good or above (Table 22, cols 2-5). A significant difference was found in their self-rated understanding of the Atomic Rules approach as compared to Myers' approach, where the mean was higher for Atomic Rules; $t(30) = -7.65$, $p < .01$. Thus, students reported that they were able to gain a better understanding of Atomic Rules [17].

Table 22: Self-rated understanding in 2004 (n = 32) [17].

	Understanding of Black-Box Testing Methods				Understanding of Approaches	
	Initial		Final			
	EP	BVA	EP	BVA	Myers	Atomic Rules
Rating	**Percentages (%)**					
Very Poor	3	9	0	0	6	0
Poor	15	18	0	0	15	0
Average	36	45	0	3	36	0
Good	15	18	12	21	15	12
Very Good	24	9	58	55	18	70
Excellent	6	0	30	21	3	18
Frequency	**Values**					
Mean	3.61	3.00	5.18	4.94	3.35	5.03
Std Dev	1.27	1.06	0.64	0.75	1.25	0.56
Missing	*0*	*0*	*0*	*0*	*1*	*0*

In addition, 82% of students in 2005 rated their understanding of Atomic Rules as Good to Excellent, compared to only 54% for Myers's representation (Table 23, cols 2-5). Furthermore, a significant difference was found in the student's self-rated understanding of the two representations, where the mean was again higher for Atomic Rules; $t(26) = -3.22$, $p = .03$. Thus, the students in 2005 also reported that they were able to gain a better understanding of the Atomic Rules representation.

Table 23: Self-rated understanding in 2005 (n = 28).

	Understanding of Black-Box Testing Methods				Understanding of Approaches	
	Initial		Final			
	EP	BVA	EP	BVA	Myers	Atomic Rules
Rating	**Percentages (%)**					
Very Poor	32	21	4	4	4	4
Poor	21	18	0	0	21	0
Average	29	29	7	11	21	11
Good	7	21	36	25	29	46
Very Good	11	7	50	46	25	29
Excellent	0	4	4	14	0	7
Frequency	**Values**					
Mean	2.43	2.86	4.39	4.54	3.50	4.22
Std Dev	1.32	1.38	.96	1.11	1.20	1.01
Missing	*0*	*0*	*0*	*0*	*0*	*1*

4.1 Related Research

In this section we discuss a number of case studies that assess testing methods taught at university. We also explore a number of other empirical studies that compare the effectiveness of black-box methods to other testing methods and reflect on the approaches they use to assess test method effectiveness, as well as the number of

participants used in those studies. We then examine the continuing debate in the literature as to whether students should be used in software engineering experimentation.

Roper *et al.* suggest that one way to progress towards firmer concepts of test method effectiveness is to develop tighter definitions of the methods themselves so that the experimental derivation of test data becomes predictable and repeatable [21]. This was of our main objectives in developing the Atomic Rules approach and one of our primary motivations in assessing the learnability and usability of traditional black-box methods. In our experiments we examined learnability in terms of the ease at which novice testers gained knowledge of particular concepts and usability in terms of the satisfaction the novice testers felt when using the representations. This included assessment of the level of completeness and correctness of derived test cases. Chen and Poon used similar measures when reviewing forty-eight student projects for the types of classifications that students missed as well as they numbers and types of mistakes they made when using the black-box Classification Tree Method (CTM) [7]. In another case study on CTM that was run 104 students and rerun with fifty-eight students, participants were asked to test programs that they had developed themselves using whatever test methods they felt were appropriate [25]. Their programs were graded by an automated test suite in terms the number of test cases that resulted in correct program output[4]. Then, the students were taught CTM and were asked to retest their programs using that method, to critically evaluate CTM, compare it to the test methods they previously used and to rate their future preference of test methods. With the exception of the critical evaluation, these measures are similar to those that were used in our experiment and to those we plan to use in our industry case study.

Other studies use metrics such as fault-detection effectiveness to assess test method effectiveness. Basili and Selby conducted an experiment involving a total of forty-two students (twenty-nine juniors, thirteen intermediates) and thirty-two industry professionals, in which they compared the fault detection effectiveness, fault detection rate and classes of faults detected by three testing techniques: black-box testing (EP and BVA), white-box testing (100% statement coverage) and code reading (by stepwise abstraction) [3]. They found that the industry professionals were able to detect the most faults with code reading and did so at a faster rate. They were also able to detect more faults with black-box methods than white-box methods; however, these did not differ in fault detection rate. In one university group the same numbers of faults were detected with code reading and black-box methods and both detected more faults than white-box methods. The rate at which students detected faults did not differ for any technique.

Kamsties and Lott repeated this experiment with fifty students and found that while the defect detection effectiveness of the two dynamic approaches (white-box and black-box) were comparable to that of the static approach (code reading), participants detected more faults using black-box methods [15]. This experiment was also repeated by Wood *et al.* with forty-seven student participants [24]. They found that participants detected similar numbers of faults for all three techniques; however, their effectiveness depended on the nature of the program under test and the program faults.

In a different experiment that compared the probability that test cases derived by EP, BVA and RT would be capable of detecting specific program faults, only one person was involved in test case derivation [20]. Reed noted that participants using black-box methods during experiments often select test cases that are not representative of other testers, therefore experiments could not be generalised unless large enough groups of testers and test cases were used. Thus, Reed sought to derive every test case that satisfied the black-box methods under study.

In our industry case study we plan to use some of the more commonly used metrics for assessing test method effectiveness, such as comparing the number of defects detected by experienced software testers when using the Atomic Rules approach to the number that are detected when testers use their own systematic and exploratory black-box methods. We also plan to compare the results of our university experiments to that of the industry case study, to determine whether experienced software testers are able to derive complete test sets using Atomic Rules.

Our university experiments involved a total of seventy-two students and we hope to obtain participation from at least thirty industry professionals. These figures are comparable with subject numbers from other studies, both inside and outside the domain of software testing. For example, in one study outside the domain of testing, an experiment was run with thirty-six students was rerun by a different researcher with fifty-nine students and ninety-nine industry professionals who were paid standard consultancy rates [2]. Remuneration could account for this relatively high number of industry participants. For example, in another experiment outside testing, only twelve industry professionals participated [12] and they did not appear to be remunerated. Thus, remuneration may be an effective approach of obtaining more industry participation in our future case study.

Carver *et al.* state that running pilot experiments with students is effective preparation for industry-based experiments [6]. In addition, Tichy stated that student experiments could be used to predict future trends in experiments that are rerun with industry professionals [22]. Tichy also stated that graduate computer science students are only marginally different from industry professionals [22]. In addition, Carver *et al.* discuss a study in which a significant difference was found between graduate and undergraduate students, but only small differences were found between graduates and industry professionals [6]. Thus, one negative aspect that has been reported on the use of students as experimental subjects is that experiment results may not be able to be

[4] Hoffman *et al.* also used automated testing tools to grade student's work [11].

generalised to industry professionals [6]. However, our university experiments have been excellent preparation for our industry case study, as they have facilitated an initial assessment of the learnability and usability of Atomic Rules and have identified threats to validity that need to be considered and controlled. Also, we consider our results to be suggestive of what the general population of professional testers might experience when learning and using the Atomic Rules approach, and this will be properly verified through the industry case study.

5. Discussion

Students in the Atomic Rules group in 2004 were able to produce significantly more complete and correct EP equivalence classes and test cases, although the BVA results were inconclusive. On the other hand, students in Myers' group in 2005 produced more complete EP equivalence classes and test cases and BVA boundary values. However, we believe this was due to the 2005 students being given a longer and more complicated specification during tutorials. As test cases generally take longer to hand-write using the Atomic Rules approach, significantly more students in the Atomic Rules group in both 2004 and 2005 did not have enough time to complete the tutorial tests. Despite this, in 2004 the Atomic Rules group still achieved much higher mean coverage levels than those using Myers' approach.

The setting of a potentially overly-complicated test in the 2005 tutorials could have been caused by the second system effect, which is where system engineers, having developed small, elegant solutions the first time around, have a tendency to design overly complicated solutions the second time [5]. It is only by the design of a third system that the engineer will develop an effective solution that is not under or over designed [5]. As students did well with the Atomic Rules approach in the 2004 tutorials, we felt it was reasonable to increase the length and complexity of the specifications used in 2005. A third experiment using a complex and a non-complex specification in the one experiment could clarify whether specification complexity caused students in the Atomic Rules group to produce less complete test sets than Myers' group in 2005.

Nonetheless, students in the Atomic Rules group produced more correct answers than Myers' group in both years, in that they made fewer mistakes during test case derivation. In addition, significantly more students in 2004 reported that they would prefer to use the Atomic Rules approach in future. Also, more students in 2005 used Atomic Rules in their assignment and achieved higher assignment marks than those that used Myers' representation. Furthermore, in both years a significant difference was found in student's self-rated understanding of the two approaches, where the mean was higher for the Atomic Rules representation. Thus, students in both years felt that they had gained a better understanding of the Atomic Rules representation by the end of the experiment.

These results suggest that Atomic Rules could be a more effective representation to teach to novice software testers at university. However, the collection of more data would allow us to draw more solid conclusions. It is hoped that the industry-based case study we are currently planning will provide us with such data.

One observation that was made during the experiment was that the structure of the Atomic Rules approach can stifle tester creativity, even for novice testers. As Atomic Rules is much more systematic than Myers' original definitions, it did not allow the students to derive test cases based on their own knowledge and experience. During data analysis, it became apparent that some testers in Myers' group created test cases that were not derivable from Myers' representation. As noted by Kaner *et al.*, prior testing experience can be used to identify effective test cases through error guessing in similar testing scenarios, even if a tester cannot remember where they gained the domain knowledge [15]. We developed an approach called Systematic Method Tailoring which allows testers to systematically define new test case selection rules and new black-box testing methods based on their own knowledge and experience, allowing these to be retained for future reuse. In our industry case study, we will determine whether the ad hoc test case selection rules that are used by industry testers can be captured as Atomic Rules.

We do not believe that student preference for lecturer had an affect on the experiment results, as it would be fair to assume that the results of the two groups would have swapped in 2005 if this was the case.

6. Conclusions and Future Work

In this paper we presented the results of two similar classroom experiments that compared the learnability and usability of two representations of Equivalence Partitioning and Boundary Value Analysis: Myers' original definitions and the corresponding Atomic Rules representation. The aim was to compare the learnability and usability of these two representations. The experiments were run with two groups of novice testers over two years at La Trobe University. While our results cannot be generalised across the entire population of novice testers, program specifications or black-box testing methods, they do suggest that the Atomic Rules representation makes black-box testing methods easier to learn and use in some situations and that students feel that they are able to gain a better understanding of Atomic Rules than Myers' representation. They also suggest that this topic warrants further investigation.

We feel that these experiments have been excellent preparation for an industry-based case study that we are currently planning for 2007, in which we will compare the fault detection effectiveness of the Atomic Rules approach to the effectiveness of the systematic and exploratory black-box testing methods that are used by experienced software testers in industry. In addition, in our university experiments we found that the Atomic Rules approach can stifle tester creativity. However, in

previous research we proposed Systematic Method Tailoring as an approach for defining new Atomic Rules and new black-box testing methods [16]. Thus, in the industry-based case study we will also determine whether professional testers use any test case selection rules that are not covered by traditional black-box methods, and if so, we will determine whether they can be described via Systematic Method Tailoring as Atomic Rules. This verification process will also allow us to further assess whether Atomic Rules and Systematic Method Tailoring are effective representations to teach to novice and professional testers in both academia and industry. We will also seek to compare the findings of our university experiments to our industry-based case study, to analyse the differences in the test sets that are derived by experienced and inexperienced testers, such as differences in the completeness and quality of the test sets and the speed at which they are derived.

During our industry case study we will also evaluate the effectiveness of an automated testing tool that we are developing, which implements Atomic Rules and Systematic Method Tailoring. This tool automatically generates a set of black-box test cases for specifications input either through a graphical user interface or via an upload facility for specifications expressed in Backus-Naur Form. This tool could increase the efficiency of the Atomic Rules approach and make black-box test case selection more efficient and precise.

Acknowledgements

We sincerely thank the students of the Department of Computer Science and Computer Engineering at La Trobe University who participated in our experiment. We also thank the postgraduate students and staff of the Department who gave constructive feedback during a seminar on the preliminary findings of our experiment. We would also like to thank the reviewers of this paper for their constructive feedback.

Lastly, we thank Sue Cook of the Human Ethics Committee of the Faculty of Science, Technology and Engineering, La Trobe University, for her valuable feedback on our experiment design.

References

[1] Anastasi, A., and Urbina, S. *Psychological Testing*. Prentice Hall, New Jersey, USA, 1997.

[2] Arisholm, E., and Sjøberg, D. Evaluating the Effect of a Deligated versus Centralized Control Style on the Maintainability of Object-Oriented Software. *IEEE Transactions on Software Engineering*, **30**(8):521-534, 2004.

[3] Basili, V., and Selby, R. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, **SE-13**(12): 1278–1296, 1987.

[4] Berry, D.M. and Tichy, W.F. Comments on Formal methods application: an empirical tale of software development. *IEEE Transactions on Software Engineering*, **29**(6):567-571, June 2003.

[5] Brooks, F. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing, 1975.

[6] Carver, J., Jaccheri, L., and Morasca, S. Issues in Using Students in Empirical Studies in Software Engineering Education. *Proceedings of the 9th International Software Metrics Symposium*, Sydney, Australia, 2003.

[7] Chen, T. Y., and Poon, P. L. Experience with Teaching Black-Box Testing in a Computer Science/Software Engineering Curriculum. *IEEE Transactions on Education*, **47**(1):42-50, February 2004.

[8] Christensen, L. *Experimental Methodology*. Pearson/Allyn and Bacon, Boston, Massachusetts, USA, 2004.

[9] Creswell, J. *Educational Research. Planning, Conducting and Evaluating Quantitative and Qualitative Research*. Pearson Education, New Jersey, USA, 2002.

[10] Gorard, S. *Quantitative Methods in Educational Research: the Role of Numbers Made Easy*. Continuum, UK, 2001.

[11] Hoffman, D., Strooper, P., and Walsh, P. Teaching and Testing. *Proceedings of the 9th Conference on Software Engineering Education*, IEEE, pp. 248-258, April 1996.

[12] Hungerford, B., Henver, A., and Collins, R. Reviewing Software Diagrams: A Cognitive Study. *IEEE Transactions on Software Engineering*, **30**(2):82-30, February 2004.

[13] Johnson, B., and Christensen L. *Educational Research. Quantitative, Qualitative and Mixed Approaches*. Pearson Education, USA, 2004.

[14] Kamsties, E., and Lott, C. An Empirical Evaluation of Three Defect-Detection Techniques. *Proceedings of the 5th European Software Engineering Conference*, September 1995.

[15] Kaner, C., Falk, J., and Nguyen, H. *Testing Computer Software*. John Wiley & Sons, 2nd edition, 1999.

[16] Mujis, D. *Doing Quantitative Research in Education with SPSS*. SAGE publications Ltd., London, UK, 2004.

[17] Murnane, T., Hall, R., and Reed, K. Towards Describing Black-Box Testing Methods as Atomic Rules. *Proceedings of the 29th Annual International Computer Software and Applications Conference*, Scotland, pp. 437-442, July 2005.

[18] Murnane, T., Hall, R., and Reed, K. Tailoring of Black-Box Testing Methods. *Proceedings of the 2006 Australian Software Engineering Conference*, Sydney, Australia, IEEE, pp. 292-299, April 2006.

[19] Myers, G. *The Art of Software Testing*. John Wiley & Sons Inc, USA, 1979.

[20] Reid, S. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. *Proceedings of the 4th International Software Metrics Symposium*, IEEE, 1997.

[21] Roper, M., Miller, J., Brooks, A., and Wood, M. Towards the Experimental Evaluation of Software Testing Techniques. *Technical Report, Department of Computer Science*, University of Strathclyde, Glasgow, 1993.

[22] Tichy, W. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Software Engineering*, **5**:309-312, 2000.

[23] Vegas, S., Juristo, N., and Basili, V. *Implementing Relevant Information for Testing Technique Selection. An Instantiated Characterization Schema*. Kluwer Academic Publishers, USA, 2003.

[24] Wood, M., Roper, M., Brooks, A., and Miller, J. Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. *Proceedings of the 6th European Software Engineering Conference / 5th AGM SIGSOFT Symposium on the Foundations of Software Engineering*, Switzerland, pp. 262-277, September 1997.

[25] Yuen, T. Y., Ng, S. P., Poon, P. L., and Chen, T. Y. On the Testing Methods Used by Beginning Software Testers. *Information and Software Technology*, Elsevier, **46**:329-335, 2004.

Chapter 9

References

"Last, but not least, avoid clichés like the plague."
William Safire, American Grammarian and Writer, 1990.

Abbott, J 1986, *Software Testing Techniques*, NCC Publications, Manchester, England, UK.

Acree, A 1980, 'On Mutation', PhD Thesis and *Technical Report GIT-ICS-80/12*, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA.

Agresti, A 2007, *An Introduction to Categorical Data Analysis*, 2nd Edition, Wiley, USA.

Agruss, C & Johnson, B 2000 'Ad Hoc Software Testing: A Perspective on Exploration and Improvisation', accessed 25 March 2008, from <http://www.testingcraft.com/ad_hoc_testing.pdf>.

Aliprand, J, Allen, J, Becker, J, Davis, M, Everson, M, Freytag, A, Jenkins, J, Ksar, M, McGowan, R, Muller, E, Moore, L, Suignard, M, & Whistler, K (eds.) 2003, *The Unicode Standard Version 4.0*, Addison-Wesley, USA.

Ammann, P & Black, P 1999, 'Abstracting Formal Specifications to Generate Software Tests via Model Checking', *Proceedings of the 18th Digital Avionics Systems Conference (DASC'99)*, volume 2, October, pp. 10.A.6, Missouri, USA, IEEE.

Ammann, P, Black, P & Majurski, W 1998, 'Using Model Checking to Generate Tests from Specifications', *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, IEEE Computer Society, pp. 46.

Ammann, P & Offutt, J 1994, 'Using Formal Methods to Derive Test Frames in Category-Partition Testing', *Proceedings of the Ninth Annual Conference on Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security (COMPASS '94)*, IEEE, pp. 69-79.

Anastasi, A & Urbina, S 1997, *Psychological Testing*, Prentice Hall, New Jersey, USA.

Andersson, C Thelin, T, Runeson, P & Dzamashvili, N 2003, 'An Experimental Evaluation of Inspection and Testing for Detection of Design Faults', *Proceedings of the 2003 International Symposium on Empirical Software Engineering*, pp. 174-184.

Andriole, S. J. (ed.) 1986, *Software Validation, Verification, Testing and Documentation*, Petrochelli Books Inc., USA.

Arisholm, E & Sjøberg, D 2004, 'Evaluating the Effect of a Deligated versus Centralized Control Style on the Maintainability of Object-Oriented Software', *IEEE Transactions on Software Engineering*, 30(8):521-534.

Australia Post 2008, 'Postcode Search', *Australia Post*, accessed 8 June 2008, from <http://www1.auspost.com.au/postcodes/>.

Abbott, J 1986, *Software Testing Techniques*, NCC Publications, UK.

Bach, J 2000, 'Session-Based Test Management', *Software Testing and Quality Engineering*, issue 11.

Bach, J 2001, 'What is Exploratory Testing?', *Sticky Minds*, accessed 4 January 2007, from <http://www.stickyminds.com/sitewide.asp?Function=edetail&Object=2555>.

Bach, J 2003, 'Exploratory Testing Explained', *Satisfice*, accessed 4 January 2007, from <www.satisfice.com>.

Bach, J & Bach, J 2006, 'Dynamics of Exploratory Testing', paper to support presentation at PNSQX 2006, titled *Exploratory Testing as Competitive Sport*, accessed February 12 2008, from <http://www.quardev.com/content/whitepapers/ExploratoryTestingasSport_JonBach_PNSQC06pdf.pdf>.

Backus, J 1958, 'The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACMGAMM Conference', *Proceedings International Conference on Information Processing*, UNESCO, London, UK, pp. 125-132.

Balcer, M, Hasling, W & Ostrand, T 1989, 'Automatic Generation of Test Scripts from Formal Test Specifications', *ACM SIGSOFT Software Engineering Notes*, 14(8):210-218, December.

Barutchu, A 2004, private communication with the author.

Barber, S 2007, 'What Software Testers can Learn from Children', accessed 28 January 2008, from <http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1261232,00.html>.

Barnes, R 1979, *PL/I for Programmers*, North-Holland.

Basili, V 1991, 'Software Modeling and Measurement: the Goal/Question/Metric Paradigm', *Technical Report*, Department of Computer Science, University of Maryland, USA.

Basili, V 1992, 'Software Modeling and Measurement: The Goal/Question/Metric Paradigm', *Technical Report UMIACS-TR-92-96*, University of Maryland, USA, September, pp. 1-24.

Basili, V, Gainluigi, C & Rombach, H 1994, 'The Goal Question Metric Approach', *The Encyclopedia of Software Engineering*, Wiley & Sons Inc., 1:528-532.

Basili, V & Selby, R 1984, 'Data Collection and Analysis in Software Research and Management', *Proceedings of the American Statistical Association of Biomeasure Society Joint Statistical Meetings*, Philadelphia, USA, August: 13-16.

Basili, V & Selby, R 1987, 'Comparing the Effectiveness of Software Testing Strategies', *IEEE Transactions on Software Engineering*, SE-13(12): 1278–1296.

Basili, V & Weiss, D 1984, 'A Methodology for Collecting Valid Software Engineering Data', *IEEE Transactions on Software Engineering*, SE-10(6):728-738, November.

Bauer, J & Finger, A 1979, 'Test Plan Generation using Formal Grammars', *Proceedings of the 4th International Conference on Software Engineering*, Germany, ACM, pp. 425-432.

Bazzichi, F & Spadafora, I 1982, 'An Automatic Generator for Compiler Testing', *IEEE Transactions on Software Engineering*, 8(4):343-353.

Beizer, B 1984, *Software System Testing and Quality Assurance*. Van Nostrand Reinhold, New York, USA.

Beizer, B 1990, *Software Testing Techniques*, Von Nostrand Reinhold, USA.

Beizer, B 1995, *Black Box Testing. Techniques for Functional Testing of Software and Systems,*. John Wiley & Sons Inc., USA.

Berry, D & Tichy, W 2003, 'Comments on Formal Methods Application: an Empirical Tale of Software Development', *IEEE Transactions on Software Engineering*, 29(6):567-571, June.

Bertolino, A 2004, 'Software Testing', *Software Engineering Body of Knowledge (SWEBOK)*, Chapter 5, IEEE.

Bertolino, A 2007, 'Software Testing Research: Achievements, Challenges, Dreams', *Proceedings of the 2007 International Conference on the Future of Software Engineering*, pp. 85-103.

Bidgoli, H 2004, *The Internet Encyclopedia*. John Wiley and Sons.

Birk, A 1997, 'Modelling the Application Domains of Software Engineering Technologies', *IESE Report No. 014.97/E*, Fraunhofer IESE, Germany.

Black, R 2007, *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*, John Wiley & Sons, USA.

Bonifati, A, Cattaneo, F, Ceri, S, Fuggetta, A, & Paraboschi, S 2001, 'Designing Data Marts for Data Warehouses', *ACM Transactions on Software Engineering and Methodology*, 10(4):452-483, October.

Bottaci, L & Mresa, E 1999, 'Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study', *Software Testing, Verification and Reliability*, 9:205-232.

Bouquet, F, Dadeau, F & Legeard, B 2006, 'Automated Boundary Test Generation from JML Specifications', *Lecture Notes in Computer Science*, Springer-Verlag, 4085:428-443.

British Standards Institute (BS) 2008, 'Non-Functional Testing', accessed 20 March 2008, from <http://www.testingstandards.co.uk/non_functional_testing_techniques.htm>.

Brooks, F 1975, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley Publishing.

Brown, P 1979, *Writing Interactive Compilers and Interpreters*, John Wiley & Sons Ltd.

British Standards Institute (BS) 2001, *BS-EN 50128:2001 Railway Applications – Communications, Signalling and Processing Systems – Software for Railway Control and Protection Systems*, British Standards Institute.

British Standards Institute (BS) 2009, 'BS 7925-1:2009 Glossary of Terms Used in Software Testing', British Computer Society, accessed 24 February 2009, from <http://www.testingstandards.co.uk/bs_7925-1_online.htm>.

British Standards Institute (BS) 2001, *BS 7925-2:2001 Software Testing: Software Component Testing Standard*, British Computer Society.

Budd, T & Gopal, A 1984, 'Program Testing by Specification Mutation', *Computer Languages*, 10(1):63-73, Penguin Press, UK.

Burnstein, I 2003, *Practical Software Testing: A Process-Oriented Approach*, Springer-Verlag, USA.

Carver, J, Jaccheri, L, Morasca, S & Shull, F 2003, 'Issues in Using Students in Empirical Studies in Software Engineering Education', *Proceedings of the 9th International Software Metrics Symposium*, Sydney, Australia.

Case Maker (Part 3) 2007, 'Case Maker User Manual. Part 3: Test Cases', accessed 12 April 2009, from <http://www.casemaker.de/download.htm>.

Case Maker (Part 4) 2007, 'Case Maker User Manual. Part 4: Test Data', accessed 12 April 2009, from <http://www.casemaker.de/download.htm>.

Celentano, A, Reghizzi, S, Vigna, P, Ghezzi, C, Granata, G & Savoretti, F 1980, 'Compiler Testing using a Sentence Generator', *Software Practice and Experience*, 10:897-918.

Chen, T & Poon, P 1996, 'Classification-Hierarchy Table: a Methodology for Constructing the Classification Tree', *Proceedings of the 1996 Australian Software Engineering Conference (ASWEC'96)*, IEEE Computer Society.

Chen, T & Poon, P 2004, 'Experience with Teaching Black-Box Testing in a Computer Science/Software Engineering Curriculum', *IEEE Transactions on Education*, 47(1):42-50, February.

Chen, T, Poon, P & Tse, T 1999, 'A New Restructuring Algorithm for the Classification-Tree Method', *Proceedings of the Software Technology and Engineering Practice*, IEEE Computer Society.

Chen, T & Yu, Y 1994, 'On the Relationship Between Partition and Random Testing', *IEEE Transactions on Software Engineering*, 20(12):977-980, December.

Chen, T & Yu, Y 1996, 'On the Expected Number of Failures Detected by Subdomain Testing and Random Testing', *IEEE Transactions on Software Engineering*, 22(2):109-119, February.

Chonoles, M & Schardt, J 2003, *UML 2 for Dummies*, Wiley, USA.

Chow, T 1978, 'Testing Software Design Modelled by Finite-State Machines', *IEEE Transactions on Software Engineering*, SE-4(3), May.

Christensen, L 2004, *Experimental Methodology*. Pearson/Allyn and Bacon, Boston, Massachusetts, USA.

Codework Solutions 2009, 'JCover. Java Code Coverage Testing and Analysis', JCover, accessed 12 April 2009, from <http://www.codework.com/JCover/product.html>.

Cohen, M, Gibbons, P, Mugridge, W & Colbourn, C 2003, 'Constructing Test Suites for Interaction Testing', *Proceedings of the 25th International Conference on Software Engineering* (ICSE'03), Portland, Oregon, pp. 38-48.

Copeland, L 2004, *A Practitioner's Guide to Software Test Design*, Artech House, Inc., USA.

Craig, R & Jaskiel, S 2002, *Systematic Software Testing*, Artech House Inc., USA.

Creswell, J 2002, *Educational Research. Planning, Conducting and Evaluating Quantitative and Qualitative Research*, Pearson Education, New Jersey, USA.

K. J. Ross & Associates 2007, 'Certified Software Test Professional Foundation, Module 2 – Black-Box Testing Methods', K. J. Ross & Associates, Melbourne, Australia.

DeMarco, T & Lister, T 1999, *Peopleware: Productive Projects and Teams*, 2nd edition, Dorset House Publishing, USA.

DeMillo, R, Lipton, R & Sayward, F 1978, 'Hints and Tips for the Practicing Programmer', *IEEE Computer*, 11(4):34-41, April.

DeMillo, R, McCracken, W, Martin, J & Passafiume, J 1987, *Software Testing and Evaluation*, Benjamin/Cummings, USA.

Díaz & Hilterscheid Unternehmensberatung GmbH (date unknown), CaseMaker Car Insurance Tutorial, Version 3.071, Berlin, Germany.

Dijkstra, E 1969, 'Notes on Structured Programming', *Technical Report 70-WSK-03*, Technological University Eindhoven, Netherlands.

Dubois, E, Yu, E, & Petit, M 1998, 'From Early to Late Formal Requirements: a Process-Control Case Study', *Proceedings of the 9th International Workshop on Software Specification and Design*, IEEE, pp. 34-42, April.

Duncan, A & Hutchison, J 1981, 'Using Attributed Grammars to Test Designs and Implementations', *Proceedings of the 5th International Conference on Software Engineering*, California, USA, IEEE, pp. 170-178.

Kruger, J & Dunning, D 1999, 'Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments', *Journal of Personality and Social Psychology*, 77(6):1121–34.

Dustin, E 2003, *Effective Software Testing: 50 Specific Ways to Improve Your Testing*, Addison-Wesley.

Duran, J & Ntafos, S 1981, 'A Report on Random Testing', *Proceedings of the 5th International Conference on Software Engineering*, IEEE, pp. 179-183.

Duran, J & Ntafos, S 1984, 'An Evaluation of Random Testing', *IEEE Transactions on Software Engineering*, 10(4):438–444.

Dyer, M 1992, *The Cleanroom Approach to Quality Software Development*. Wiley, USA.

Elmendorf, W 1974, 'Functional Analysis using Cause-Effect Graphs', *Proceedings of SHARE XLJII*, New York, Share.

Everett, G & McLeod, R 2007, S*oftware Testing: Testing Across the Entire Software Development Life Cycle*. John Wiley & Sons, Inc., USA.

Fabbri, S, Maldonado, J, Sugeta, T & Masiero, P 1999, 'Mutation Testing Applied to Validate Specifications Based on Statecharts', *Proceedings of the 10th International Symposium on Software Reliability Engineering*, IEEE, pp. 210.

Fielding, Dr M 2004, private communication with the author.

Fewster, M & Graham, D 2000, *Software Test Automation. Effective Use of Test Execution Tools*, Addison-Wesley, USA.

Flanagan, D 2002, *JavaScript: The Definitive Guide* 4th edition, O'Reilly & Associates, Inc., USA.

Fultyn, R 1982, 'Computer-Assisted Software Testing', *Proceedings of the 1982 ACM Annual Conference/Annual Meeting*, ACM, USA, pp. 7-12.

Fuchs, N & Schwitter, R 1996, 'Attempto Controlled English (ACE)', *Proceedings of the First International Workshop on Controlled Language Applications (CLAW 96)*, Katholieke Universiteit Leuven, March.

Goodenough, J & Gerhart, S 1975, 'Toward a Theory of Test Data Selection', *IEEE Transactions Software Engineering*, SE-2(2):156-173, June.

Gorard, S 2001, *Quantitative Methods in Educational Research: the Role of Numbers Made Easy*, Continuum, UK.

Graham, D1994, *Testing. In Encyclopedia of Software Engineering*, 2:1003-1353, John Wiley & Sons, USA..

Grindal, M 2007, 'Handling Combinatorial Explosion in Software Testing', PhD Thesis, University of Skövde.

Grindal, M, Lindström, B, Offutt, A, & Andler, S 2004, 'An Evaluation of Combination Strategies for Test Case Selection', *Technical Report HS-IDA-TR-03-001*, Department of Computer Science, University of Skövde, October.

Grindal, M, Offutt, J & Andler, S 2004, 'An Evaluation of Combination Strategies for Test Case Selection', *Technical Report HS-ISA-TR-03-001*, Department of Computer Science, University of Skövde, October.

Grindal, M, Offutt, J & Andler, S 2005, 'Combination Testing Strategies: A Survey', *Software Testing, Verification and Reliability*, 15:167-199, John Wiley & Sons Ltd, USA.

Grochtmann, M, & Grimm, K 1993, 'Classification Tress for Partition Testing', *Software Testing, Verification and Reliability*, 3(2):63-82.

Grochtmann, M, Grimm, K, & Wegener, J 1993, 'Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor', *Proceedings of the 1st European International Conference on Software Testing, Analysis and Review* (EuroSTAR 1993), London, UK, Qualtech Conferences, October, pp. 169-176.

Gutjahr, W 1999, 'Partition Testing vs. Random Testing: the Influence of Uncertainty', *IEEE Transactions on Software Engineering*, 25(5):661-674, September/October.

Hackman J & Oldham, G 1980, *Work Redesign*, Addison-Wesley, USA.

Hamlet, D & Taylor, R 1988, 'Partition Testing does not Inspire Confidence', *Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*, Canada, July, pp. 206-215.

Hamlet, D & Taylor, R 1990, 'Partition Testing does not Inspire Confidence', *IEEE Transactions on Software Engineering*, 16(12):1402–1411.

Hass, K, Wessels, D & Brennan, K 2007, *Getting it Right: Business Requirement Analysis Tools and Techniques*, Management Concepts.

Healey, J F 2005, *Statistics: A Tool for Social Research*, 5th Edition, Wadsworth Cengage Learning, USA.

Hetzel, W 1988, *The Complete Guide to Software Testing*, QED Information Sciences Inc, USA.

Hoffman, D, Strooper, P, & Walsh, P 1996, 'Teaching and Testing', *Proceedings of the 9th Conference on Software Engineering Education*, IEEE, April, pp. 248-258.

Homer, W & Schooler, R 1989, 'Independent Testing of Compiler Phases Using a Test Case Generator', *Software Practice and Experience*, 19(1):53-62, January.

Houssais, B 1977, 'Verification of an ALGOL 68 Implementation', *Proceedings of the Strathclyde ALGOL 68 Conference*, Glasgow, Scotland, March.

Howden, W 1976, 'Reliability of the Path Analysis Testing Strategy', *IEEE Transactions on Software Engineering*, SE-2(3):208-215, September.

Howden, W 1980, 'Functional Program Testing', *IEEE Transactions on Software Engineering*, SE-6(2):162-169, March.

Howden, W 1981, 'A Survey of Dynamic Analysis Methods', In *Software Testing and Validation Techniques*, 2nd edition, IEEE Computer Society Press, USA.

Hungerford, B, Henver, A, & Collins, R 2004, 'Reviewing Software Diagrams: A Cognitive Study', *IEEE Transactions on Software Engineering*, 30(2):82-30, February.

Hutcheson, M 2003, *Software Testing Fundamentals, Methods and Metrics*, Wiley Publishing Inc., USA.

Institute of Electrical and Electronic Engineers (IEEE) 2008, *IEEE 829:2008 IEEE Standard for Software and System Test Documentation*, IEEE, USA.

Institute of Electrical and Electronic Engineers (IEEE), 2004, *IEEE 1012:2004 Standard for Software Verification and Validation*, IEEE, USA.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 2008, *ISO/IEC 12207:2008 Software and Systems Engineering – Software Life Cycle Processes*, International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), 2008, *ISO/IEC 15288:2008 Software and Systems Engineering – System Life Cycle Processes*, International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission, (ISO/IEC) 2008, *ISO/IEC 24765 Software and Systems Engineering – Vocabulary*, International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 2009, *ISO/IEC 24765:2009 Software and Systems Engineering – Vocabulary*. International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 2006, *ISO/IEC 25062:2006 Software Engineering – Software Product Quality Requirements and Evaluation (SQuaRE) – Common Industry Format (CIF) for Usability Test Reports*, International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 2008, *ISO/IEC 29119 Software Testing, Part 1, Concepts and Vocabulary*, Draft version available on request from ISO/IEC JTC1/SC7 (Software and Systems Engineering) Working Group 26 (Software Testing) and the author, International Organization for Standardization.

International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) 2005, *ISO/IEC 9126:2005, Software Engineering – Product Quality, Part 1: Quality Model*, International Organization for Standardization.

International Software Testing Qualifications Board (ISTQB) & Erik van Veenendall (ed.) 2005, *Standard Glossary of Terms Used in Software Testing*, Glossary Working Party of the ISTQB, Version 101, 29 September.

Itkonen, J, Mäntylä, M & Lassenius, C 2007, 'Defect Detection Efficiency: Test Case Based vs. Exploratory Testing', *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pp. 61-70.

Itkonen, J & Rautiainen, K 2005, 'Exploratory Testing: A Multiple Case Study', *Proceedings of the 4th International Symposium on Empirical Software Engineering*, IEEE, pp. 84-93.

Jeng, B & Weyuker, E 1989, 'Some Observations on Partition Testing', *Proceedings of the ACM SIGSOFT 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)*, Key West, USA, December, pp. 38–47.

Johnson, B, & Christensen L 2004, *Educational Research. Quantitative, Qualitative and Mixed Approaches*, Pearson Education, USA.

Jorgensen, P 1995, *Software Testing: A Craftsman's Approach*, Department of Computer Science and Information Systems, Grand State University Allendale, Michigan and Software Paradigms, Rockford, Michigan, CRC Press, USA.

Jorgensen, P 2002, *Software Testing: A Craftsman's Approach*, 2nd edition, CRC Press, USA.

JSynTest, 'Man Machine Systems. Automated Syntax Testing using JSynTest', accessed 26 February 2007, from <http://www.mmsindia.com/JSynTest-Overview.pdf>.

JUnit 2009, 'Resources for Test Driven Development', accessed 20 July 2009, from <http://www.junit.org/>.

Kamsties, E & Lott, C 1995, 'An Empirical Evaluation of Three Defect-Detection Techniques', *Lecture Notes In Computer Science*, 9:362-383.

Kaksonen, R, Laakso, M & Takanen, A 2008, 'Vulnerability Analysis of Software through Syntax Testing', accessed Sunday 9 March 2008, from <http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/>.

Kaner, C 1988, *Testing Computer Software*, 1st edition, TAB Books Inc, USA.

Kaner, C, Bach, J, & Pettichord, B 2001, *Lessons Learned in Software Testing: A Context-Driven Approach*, John Wiley & Sons, Inc., USA.

Kaner, C, Falk, J, & Nguyen, H 1999, *Testing Computer Software*, 2nd edition, John Wiley & Sons, USA.

Kent, J 2008, 'An Entity Model of Software Testing: A Foundation for the Future', *Proceedings of the 18th European International Conference on Software Testing, Analysis and Review* (EuroSTAR 2008), Stockholm, Sweden, 30 November to 3 December.

Kit, E 1995, *Software Testing in the Real World: Improving the Process*, ACM Press, USA.

Klugh, H E 1986, Statistics: The Essential for Research, 3rd Edition, Lawrence Erlbaum Associates, USA.

Kroll, L, & Gildman, L (eds.) 2005, 'The World's Billionaires', accessed 4 November 2005, from <http://www.forbes.com/billionaires/>.

Knuth, D 1964, 'Backus Normal Form vs. Backus Naur Form', *Communications of the ACM*, 7(12):735-736, December.

Knuth, D 1973, 'Seminumerical Algorithms', *The Art of Computer Programming*, Volume 2, Addison-Wesley, USA.

Koslowski, B 1996, *Theory and Evidence: the Development of Scientific Reasoning*, MIT Press, Boston.

Lamsweerde, A van & Willemet, L 1998, 'Inferring Declarative Requirements Specifications from Operational Scenarios', *IEEE Transactions on Software Engineering*, 24(12):1089-1114, December.

Lauterbach, L & Randell, W 1989, 'Experimental Evaluation of Six Test Techniques', *Proceedings of the Fourth Annual Conference on Computer Assurance* (COMPASS '89), June, pp. 19-23.

Lee, J & Dorocak, J 1973, 'Conditional Syntactic Specification', *Proceedings of the ACM Annual Conference/Annual Meeting*, ACM, USA, pp. 101-105.

Lee, S & Offutt, J 2001, 'Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis', *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, IEEE Computer Society, pp. 200.

Lehmann, E, & Wegener, J 2000, 'Test Case Design by Mean of the CTE XL', *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review* (EuroSTAR 2000), Copenhagen, Denmark, December 2000, Qualtech Conferences, December, pp. 1-10.

Lewis, W 2000, *Software Testing and Continuous Quality Improvement*, CRC Press, Florida, USA.

Letier, E, & Lamsweerde, A van 2004, 'Reasoning About Partial Goal Satisfaction for Requirements and Design Engineering', *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM Press, USA, pp. 53-62.

Liskov, B & Zilles, S 1975, 'Specification Techniques for Data Abstractions', *Proceedings of the International Conference on Reliable Software*, ACM, pp. 72-87.

Maiden, N & Rugg, G 1996, 'ACRE: Selecting Methods for Requirements Acquisition', *Software Engineering Journal*, 11(3):183-192, May.

Mandl, R 1985, 'Orthogonal Latin Squares: An Application of Experiment Design to Compiler Testing', *Communications of the ACM*, 28(10):1054-1058, October.

Marick, B 1995, *The Craft of Software Testing: Subsystem Testing, Including Object-Based and Object-Oriented Testing*, Prentice Hall PTR, Englewood Cliffs, New Jersey, USA.

Marr, J & Lawlis, P 1991, 'Automatic Determination of Recommended Test Combinations for Ada Compilers', *Proceedings of the 8th Annual Washington Ada Symposium & Summer SIGAda Meeting*, Virginia, USA, pp. 77-89.

McDermid, J (ed.) 1991, *Software Engineer's Reference Book*, Butterworth-Heinmann Ltd., UK.

Meek, B 1994, 'A Taxonomy of Datatypes', *ACM SIGPLAN Notices*, 24(9):159-167.

Merkel, R 2005, 'Analysis and Enhancements of Adaptive Random Testing', PhD dissertation, Swinburne University, Hawthorn, Australia.

Microsoft 2003, 'Using the Windows Applications Exploratory Test Methodology', Microsoft TechNet, accessed 4 January 2007, from <http://technet2.microsoft.com/WindowsServer/en/library/5d448271-1c45-4b5d-a800-57a7545c99f41033.mspx?pf=true>.

Miller, J & Maloney, C 1963, 'Systematic Mistake Analysis of Computer Programs', *Communications of the ACM*, 6(2):58-63.

Mosley, D 1993, *The Handbook of MIS Application Software Testing. Methods, Techniques, and Tools for Assuring Quality through Testing*, Prentice-Hall Inc., USA.

Mosley, D & Posey, B 2002, *Just Enough Software Test Automation*, Prentice-Hall Inc., USA.

Mujis, D 2004, *Doing Quantitative Research in Education with SPSS*, SAGE publications Ltd., UK.

Murnane, T 1999, 'The Application of Mutation Techniques to Specification Testing', Honours thesis, Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Australia.

Murnane, T, Hall R & Reed, K 2005, 'Towards Describing Black-Box Testing Methods as Atomic Rules', *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC '05)*, Scotland, IEEE Computer Society, pp. 437-442.

Murnane, T & Reed, K 2001, 'On the Effectiveness of Mutation Analysis and a Black-Box Testing Technique', *Proceedings of the 2001 Australian Software Engineering Conference (ASWEC '01)*, Australia, IEEE Computer Society, pp. 12-20.

Murnane, T, Reed, K & Hall, R 2006, 'Tailoring of Black-Box Testing Methods', *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC '06)*, Australia, IEEE, April, pp. 292-299.

Murnane, T, Reed, K & Hall, R 2007, 'On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis', *Proceedings of the 2007 Australian Software Engineering Conference (ASWEC '07)*, Australia, IEEE Computer Society, pp. 274-283.

Musa, J 1993, 'Operational Profiles in Software Reliability Engineering', *IEEE Software*, 10(2):14-32, March.

Myers, G 1978, 'A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections', *Communications of the ACM*, 21(9):760–768.

Myers, G 1979, *The Art of Software Testing*, John Wiley & Sons Inc, USA.

Myers, G 2004, *The Art of Software Testing*, 2nd Edition, revised and updated by Badgett, T & Thomas, T with Sandler, C, John Wiley & Sons Inc, USA.

Naik, K & Tripathy, P 2008, *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, Inc., USA.

Naur, P (ed.) 1960, 'Report on the Algorithmic Language ALGOL 60', *Communications of the ACM*, 3(5):299-314, May.

Ng, S, Murnane, T, Reed, K, Grant, D, & Chen, T 2004, 'A Preliminary Survey of Software Testing in Australia', *Proceedings of the 2004 Australian Software Engineering Conference*, Australia, IEEE, April, pp. 116-125.

Nguyen, H, Hackett, M, Johnson, B & Johnson, R 2003, *Testing Applications on the Web: Test Planning for Mobile and Internet-based Systems*, 2nd edition, John Wiley and Sons, USA.

Ntafos, S 1988, 'A Comparison of Some Structural Testing Strategies', *IEEE Transactions on Software Engineering*, 14:868-874.

Ntafos, S 1998, 'On Random and Partition Testing', *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, Florida, USA, March, pp. 42-48.

Offutt, A & Lee, S 1994, 'An Empirical Evaluation of Weak Mutation', *IEEE Transactions on Software Engineering*, 20(5), May.

Offutt, A & Liu, S 1999, 'Generating Test Data from SOFL Specifications', *Journal of Systems and Software*, 49(1):49-62.

Ostrand, T 2002, 'Generating Formal Specifications from Test Information', *Proceedings of the Workshop on Formal Approaches to Testing*, Masaryk University, Brno, Czech Republic.

Ostrand, T & Balcer, M 1988, 'The Category-Partition Method for Specifying and Generating Functional Tests', *Communications of the ACM*, 31(6):676-686, June.

Ould, M & Urwin, C (eds.) 1986, *Testing in Software Development*, Press Syndicate of the University of Cambridge, UK.

Oualline, S 2003, *Practical C++ Programming*, 2nd Edition, O'Reilly, USA.

The Oxford English Dictionary 1970, volume V, H-K, Oxford University Press, Ely House, London.

Page, A, Johnston, K & Rollison, B 2009, *How We Test Software at Microsoft*, Microsoft Press, USA.

Paakki, J 1995, 'Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation', *ACM Computing Surveys*, 27(2):196-255, June.

Parrington, N & Roper, M 1989, *Understanding Software Testing*, Ellis Horwood Ltd., UK.

Patton, R 2006, *Software Testing*, Sams Publishing, USA.

Perry, W 1983, *A Structured Approach to Systems Testing*, Prentice-Hall, USA.

Perry, W 2000, *Effective Methods for Software Testing*, 2nd edition, John Wiley & Sons Inc., USA.

Perry, W 2006, *Effective Methods for Software Testing*, John Wiley & Sons Inc., USA.

Pfleeger, S 2001, *Software Engineering: Theory and Practice*, Prentice-Hall, USA.

Project Management Institute (PMI) 2004, *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 4th edition, Project Management Institute.

Potts, C & Bruns, G 1988, 'Recording Reasons for Design Decisions', *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pp. 418-427, IEEE Press.

Pressman, R 1992, *Software Engineering: A Practitioners* Approach, 3rd edition, McGraw-Hill, USA.

Prieto-Díaz, R 1991, 'Implementing Faceted Classification for Software Reuse', *Communications of the ACM*, 34(5):89-97.

Prieto-Díaz, R & Freeman, P 1987, 'Classifying Software for Reusability', *IEEE Software*, 4(1):6-16.

Pezzand, M & Young, M 2008, *Software Testing and Analysis: Process, Principles and Techniques*, John Wiley & Sons Inc., USA.

Rae, A, Hausen, H & Robert, P 1995, *Software Evaluation for Certification: Principles, Practice and Legal Liability*, McGraw-Hill Book Company, UK.

Reed, K 1981, 'Software Testing and Reliability Class Assignment', Royal Melbourne Institute of Technology (RMIT), , Bundoora, Victoria, Australia.

Reed, K 1990, 'An Outline of a Knowledge Acquisition Based Approach to Software Project Planning. Position paper for CASE90', Department of Computer Science and Computer Engineering, La Trobe University, , Bundoora, Victoria, Australia..

Reed, K 1998, 'Software Testing and Reliability (CSE32STR) Course Notes', Department of Computer Science and Computer Engineering, La Trobe University, Bundoora, Victoria, Australia.

Reed, K 2007, private communication with the author.

Reid, S 1994, 'Test Effectiveness in Software Module Testing', *Proceedings of the 2nd European International Conference on Software Testing, Analysis and Review (EuroSTAR 1994), Brussels*.

Reid, S 1997, 'An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing', *Proceedings of the 4th International Software Metrics Symposium (METRICS'97)*, pp. 64.

Reid, S 2007, 'The Personal Test Maturity Model', Software Quality Systems Conference UK (SQC-UK), Software Quality Systems, UK.

Reid, S, Harman, M, Hierons, R, Holcombe, M, Jones, B, Roper, M & Woodward, M 1999, 'A Framework for Measurement in Software Testing', *7th European International Conference Software Testing Analysis & Review* (EuroSTAR'99), November 8-12, Spain.

Reserve Bank of Australia 2005, 'Exchange Rates', accessed 4 November 2005, from <http://www.rba.gov.au/Statistics/exchange_rates.html>.

Richardson, D & Clarke, L 1981, 'A Partition Analysis Method to Increase Program Reliability', *Proceedings of the 5th International Conference on Software Engineering*, California, USA, pp. 244-253.

Richardson, D & Clarke, L 1985, 'Partition Analysis: A Method Combining Testing and Verification', *IEEE Transactions on Software Engineering*, 11(12):1477-1490.

Roper, M, Miller, J, Brooks, A, & Wood, M 1993, 'Towards the Experimental Evaluation of Software Testing Techniques', *Technical Report, Department of Computer Science*, University of Strathclyde, Glasgow.

Rosen, K & Michaels, J 2000, *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, USA.

Rubin, J 1994, *Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests*, John Wiley & Sons Inc., USA.

Runeson, P 2003, 'Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data', *Proceedings 7th International Conference on Empirical Assessment and Evaluation in Software Engineering (EASE'03)*, pp 95-102.

Rugg, G, McGeorge, P, & Maiden, N 2000, 'Method Fragments', *Expert Systems*, 17(5):248-257, November.

Safire, W 1990, *Fumblerules: A Lighthearted Guide to Grammar and Good Usage*, Doubleday, New York.

Daley, N, Hoffman, D & Strooper, P 2002, 'A Framework for Table Driven Testing of Java Classes', *Software Practice and Experience*, 32:465-493.

Salas, P 2007, 'Using MBT for Privacy Testing', *Presentation at the Test Automation Workshop 2007 (TAW'07)*, Centre for Software Assurance, Bond University, Gold Coast, Australia.

Sauder, L 1962, 'A General Test Data Generator for COBOL', *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 21, American Federation of Information Processing Societies, pp. 317-323, May.

Selby, R, Basili, V & Baker, F 1987, 'Cleanroom Software Development: An Empirical Evaluation', *IEEE Transactions on Software Engineering*, SE-13(9):1027-1037, September.

Seo, K & Choi, E 2006, 'Comparison of Five Black-Box Testing Methods for Object-Oriented Software', *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, (SERA'06), IEEE Computer Society, August, pp. 213-220.

Singh, H, Conrad, M & Sadeghipour, S 1997, 'Test Case Design Based on Z and the Classification-Tree Method', *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, IEEE Computer Society, pp. 81-90.

Sommerville, I 1994, *Software Engineering*, Pearson Education Ltd., UK.

Sommerville, I 2001, *Software Engineering*, 6th edition, Pearson Education Limited, UK.

Sommerville, I, Sawyer, P, & Viller, S 1998, 'Viewpoints for Requirements Elicitation: a Practical Approach', *Proceedings of the 3rd International Conference on Requirements Engineering* (ICRE '98), Colorado Springs, USA, IEEE, pp. 74-81.

Special Broadcasting Service (SBS) 2003, *World Guide: The Complete Fact File on Every Country*, Hardie Grant Books, Australia.

St George Bank 2005, 'St George Bank Calculators', accessed 4 November 2005, from <http://www.stgeorge.com.au/calculators/default.asp?orc=business>.

St George Bank Foreign Exchange Services 2005. 'Foreign Exchange Services', accessed 4 November 2005, from <http://www.stgeorge.com.au/smallbus/intern_soln/foreign_xservice/default.asp?orc=business>.

Tamres, L 2002, *Introducing Software Testing*, Pearson Education Ltd., Great Britain.

Tichy, W 2000, 'Hints for Reviewing Empirical Work in Software Engineering', *Empirical Software Engineering*, 5:309-312.

Thayer, R, Lipow, M, & Nelson, E 1978, *Software Reliability*, North-Holland, Amsterdam.

Tsoukalas, M, Duran. J & Ntafos, 1993, 'On Some Reliability Estimation Problems in Random and Partition Testing', *IEEE Transactions on Software Engineering*, 19(7):687-697, July.

Tsubery, Y 2007, 'Implementation Of CaseMaker in MIS BU – Comverse', accessed 9 August 2009, from <http://www.casemaker.eu/casestudies/casestudy_comverse.pdf>.

Vegas, S, Juristo, N, & Basili, V 2003, *Implementing Relevant Information for Testing Technique Selection. An Instantiated Characterization Schema*, Kluwer Academic Publishers, USA.

Vegas, S 2004, 'Identifying the Relevant Information for Software Testing Technique Selection', *Proceedings of the 2004 International Symposium on Empirical Software Engineering* (ISESE'04), IEEE.

VModell® XT 2008, 'Part 1: Fundamentals of the V-Modell', accessed 24 February 2009, <http://ftp.uni-kl.de/pub/v-modell-xt/Release-1.2/Dokumentation/pdf/V-Modell-XT-Teil1.pdf>.

Offutt, A & Voas, J 1996, 'Subsumption of Conditional Coverage Techniques by Mutation Testing', *Technical Report ISSE-TR-96-01*, Department of Information and Software Engineering, George Mason University, January.

von Mayrhauser, A 1990, *Software Engineering: Methods and Management*. Academic Press, USA.

Watkins, J 2001, *Testing IT: Off-the-Shelf Software Testing Processes*. Cambridge University Press, UK.

Weiser, M, Gannon, J, & McMullin, P 1985, 'Comparison of Test Coverage Metrics', *IEEE Software*, 19(2):80-85, March.

Weyuker, E & Jeng, B 1991, 'Analyzing Partition Testing Strategies', *IEEE Transactions on Software Engineering*, 17(7):703-711, July.

Weyuker, E & Thoman, J 1980, 'Theories of Program Testing and the Application of Revealing Subdomains', *IEEE Transactions on Software Engineering*, SE-6(3):236-246, May.

Wikipedia Unicode 2008, 'Mapping of Unicode Characters', accessed 29 February 2009, from <http://en.wikipedia.org/wiki/Mapping_of_Unicode_characters>.

Wikipedia Context Free 2008, 'Context Free Grammars', accessed 9 March 2008, from <http://en.wikipedia.org/wiki/Context-free_grammar>.

Wild, C & Eckhardt, D 1989, 'Reasoning About Software Specifications: A Case Study', *Proceedings of the AIAA Computers in Aerospace VII Conference*, pp. 297-306.

Wild, C, Zeil, S, Feng, G & Chen, J 1992, 'Employing Accumulated Knowledge to Refine Test Descriptions', *Software Testing, Verification and Reliability*, 2(2):53-68, July.

Wolpe, H 1958, 'Algorithm for Analyzing Logical Statements to Produce a Truth Function Table', *Communications of the ACM*, 1(3):4-13, March.

Wood, M, Roper, M, Brooks, A & Miller, J 1997, 'Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study', *Proceedings of the 6th European Software Engineering Conference / 5th AGM SIGSOFT Symposium on the Foundations of Software Engineering*, Switzerland, September, pp. 262-277.

Woodward, M 1993, 'Errors in Algebraic Specifications and an Experimental Mutation Testing Tool', *Software Engineering Journal*, July, pp. 211–224.

Younessi, H 2002, *Object-Oriented Defect Management of Software*, Prentice Hall PTR, USA.

Yu, Y, Ng, S, Poon, P & Chen, T 2003, 'On the Use of the Classification-Tree Method by Beginning Software Testers', *Proceedings of the 2003 ACM Symposium on Applied Computing*. ACM, pp. 1123-1127.

Yuen, T, Ng, S, Poon, P & Chen, T 2004, 'On the Testing Methods Used by Beginning Software Testers', *Information and Software Technology*, Elsevier, 46:329-335.

Zhu, H, Hall, P & May, J 1997, 'Software Unit Test Coverage Adequacy', *ACM Computing Surveys*, 29(4):366-427, December.