Identifying Reusable Components in Software Requirements Specifications To Develop a Natural Language-Like SRS Language with a CRNLP (Compileable Restricted Natural Language Processor)

TR005

Janurary 1992

by Michael Yong and Karl Reed

Amdahl Australian Intelligent Tools Program

Department of Computer Science & Computer Engineering

La Trobe University

Bundoora, Victoria 3083

Australia

Table of Contents

Abstract

One problem in specifying user's requirements for a new computer system is the ambiguity in the Software Requirements Specifications (SRS). Natural Language (NL) is inherently ambiguous and Formal Specification Language is too difficult for users to understand. This paper outlines a process to eliminate the ambiguity of SRS written in NL. A Compileable Restricted Natural Language Processor (CRNLP) develops a NL-like SRS language that is acceptable to SRS writers. A prototype has been built to demonstrate this process.

1. Introduction

The Requirements Analysis and Specification Phase of the Software Development Life Cycle involves analysing the user's requirements and then specifying a system to satisfy them. It is important that the Requirements Specification is correct, complete and unambiguous to ensure the proposed system meets their requirements.

The Requirements Specification needs approval from users and therefore has to be written in a form understandable to them. RS is traditionally and easily written in Natural Language, with the assumption that the users are able to understand English sentences. RS written in NL is inherently ambiguous, resulting in documents that are ambiguous, inconsistent and incomplete.

This paper reports on a project that looked at some SRSs, trying to identify reusable components, and developed a compileable restricted NL-like SRS language with a CRNLP.

2. Specification Language

It is suggested that the language used to write RS should be restricted or formalised so that the problem of inherent ambiguity can be reduced or even solved [1]. Various methods have been developed to help solve the problem, such as system specification language and graphical tools [2]. This paper will examine the use of a development of system specification language.

The use of formal system specification language can help ensure ambiguity-free and consistency, but not correctness of the RS. The system might be correctly specified but it may not be what the users want [3]. Users who have to approve the RS may find it extremely difficult to determine whether a RS written in a formal language does indeed satisfy them. Our approach will avoid this problem.

3. NL-Like Specification Language

A solution to the problem associated with use of formal specification is to have a language that is formalised or restricted but still remains NL-like. It should look very much like NL (English), but not exactly so. There should be some restrictions on how a specification statement is to be written, using a restricted vocabulary. The language should be acceptable to Software Requirements Specification writers, who agree to deliberately choose their dialect accordingly. I.e. instead of some undefined domain specific dialect of an NL, the language is a restriction of software engineering implementation domain.

One way to develop such a language is by examining samples of text from the software engineering implementation domain to first discover the nature of the restricted dialect of NL that is used. While doing this, we would hope to identify a vocabulary for software engineering and a grammar governing its use. Moreover, we see similarities among RS statements, and statements which are used repeatedly throughout the system specification. Hence we can propose a 'compileable' NL-like language as a tool for writing SRS.

4. Reuse

The creation of a CRNL based specification system may enable us to achieve some degree of reuse in writing SRS, where statements specifying same requirements need not be written over and over again. Instead, user can reuse the one already existing, where appropriate. This is one of the main objectives of AAITP.

To provide a short-term deliverable, we need a very restricted approach, which is to further restrict the dialect, ie. to start with very simple SRS statements.

5. Development of the Language

A Language is a collection of Sentences generated from a 4-tuple of Terminals, Non-Terminals, Grammatical Rules and a Start Symbol. In order to develop a language, we need to be able to specify the four components, firstly the Terminals and the Grammatical Rules, in which Terminals will be the Vocabulary, and Grammatical Rules the Grammar of the language.

In order to identify the vocabulary of a language, instances of the language ought to be examined. We need to look at some typical sentences of the language. As suggested by Jacob Cybulski and Jane Philcox, we would need to examine samples of free text from the software engineering implementation domain to first discover the nature of the restricted dialect of NL that is used.

Meanwhile, as we need to be able to achieve some aspect of reuse in design in fairly short order, Karl Reed suggested that we need an attack on a more restricted approach, which, while providing short-term deliverable, will address one phase of the NLP issues to be handled.

Reed pointed out that NLP could be used to process restricted, NL-like documents for content. It has been suggested that use of NL in specifications in which the language used can be assumed to be even more restricted than the dialect that would be used in free text system design documents.

The four areas are:

1. Module Interface Specification,

2. IEEE Standard Requirements Documents,

3. Psuedo-Code Descriptions of a System Structure,

4. Annotations for Diagrams.

Reed's proposal is that, rather than assume that some domain specific dialect of an NL as may be used during development, we deliberately restrict the language even further [1].

The basis of the restriction would be as follows:

Choose a 'compileable' NL-like language that would be acceptable to a user who was performing any of the above tasks, and who was deliberately choosing their dialect so that some later (possibly automatic) analysis would allow similarities and dissimilarities to be identified.

6. Ultimate Goal: A SRS Language with a CRNLP

There is no easy way to develop a language that is applicable to all the possible SRS. We first have to look at some particular statements of some system (technical, perhaps) so that some similarities could be identified.

We will then be able to develop some parsers or perhaps compilers for each systems looked at. The more systems being looked at, the wider the language would be. The language could easily be expanded by adding more terminals (ie. expanding the vocabulary) and further developing and generalising the grammar.

However, we still need to firstly look at some SRS statements to get started. By looking at the specification of a system, we are able to identify the degree of similarity among the statements of the specifications. A language can be proposed where all the statements in the specification can be represented by sentences written in the language, with all important information being captured. At the same time, a slightly formalised way to write these statements reduces ambiguity and inconsistency.

Our ultimate goal is to design a language that is capable of capturing all SRSs. This is almost impossible considering the fact that a system can be implemented almost everywhere for anything. There are so many possibilities of what a SRS could look like. Therefore, we need to start from looking at some simple system.

While examining the SRS statements of a system, we can somehow rewrite them without changing their meaning, to a more restricted way, ie. by applying some grammatical rules and limitations on vocabulary. Then similar sentences can be grouped together, according to their nature, behaviour, function or implication. Further study can be done for each group of sentences and similarities or dissimilarities can be identified.

For a group of similar sentences (by similar, we mean they have something in common, may be their format, the object or operation they specify, etc.) a grammatical rule can be written to parse their sentences. With appropriate semantic actions, a compileable language can be generated.

Accordingly, a study was made of a number of specifications to attempt to identify such CRNLP's.

7. Method Adopted

There were a few constraints that had affected the method adopted and the approach in carrying out the study.

Given time, long-term difficulties, and some minor constraints, the following method was adopted:

1. Study a few series of SRSs and identify the possible ways to develop languages from them.

2. Build prototype of CRNLP for each SRS.
- first a parser,
- a sentence recognisor with statements in database,
- a processor that interacts with user.

3. The tool used to build prototype is Prolog.

7.1 Parsers

There were a few systems being studied. They are:

1. UNIX Commands Specifications,
2. Fortran Library Routines Specifications,
3. DOS-M Directives Specifications,
4. TAME Subsystem Requirements Specifications,
(TAME - Tools for an Ada Measurement Environment)

When a set of SRS statements are examined, definitely there would be ways similarities can be identified. For example, some sentences might be talking about a type of operation on different operands in each sentence; or an object being discussed in a few sentences.

Steps in Grouping and Classification of Sentences:

1. Group sentences that allow similarities to be identified, in another word, sentences that look alike in certain ways.

2. Rewrite the sentences, eg. re-arrange sentence structure, use common words, etc, so that there is a rule in writing the sentences.

3. Draw Syntax Diagrams for each groups of sentences. Derive Grammatical Rules from the syntax diagrams and decide the Vocabulary of what is allowed to be associated with the grammar.

4. Propose a language provided a parser is available.

7.2 Recognisors

After a parser has been written, valid sentences will be recognised by the system. The statements already in SRS were stored in a table in the system. When we have a new parseable sentence, the system would be able to tell if any sentence in the database is reusable to represent the new sentence, or perhaps some similarities or dissimilarities could be identified between the new sentence and the sentences in the table.

The way to do this is to store the compiled sentences in the database, and each being distinguished by an id. If a new sentence is identified to be the same, equivalent, or having similar characteristics with any statement in the table, the system would advise the user.

The detailed interaction method will be fully explained in the next section.

7.3 CRNLP

Our first goal is to design a compiler-like processor, CRNLP, that given a sentence, will determine if it is parseable and generates a representation of its content semantic. At the same time, the processor would have a number of sentences in the database, perhaps some of the statements in the specification. These are a few possible outcomes:

1. sentence is parseable, and recognised as something having exactly the same information as one in the database.

2. sentence is parseable, and recognised as something equivalent to one in the database. System will ask user whether they mean the same thing or if it is another way to specify the same thing.

3. sentence is parseable, and recognised as something similar to one in the database. By similar, we mean it specifies similar task, or different task on same object, or same task on different object, etc, depending to the system. The system will then ask whether they mean the same, or the new one has some additional information or alteration to the one already in the database, or whether it is a new statement all together.

4. sentence is parseable, but not recognised by the system identical or similar to those in the database. The user may then add the new statement into the database.

5. sentence unparseable, due to syntax error. The system would print error message and advise user what was wrong.

6. sentence unparseable, due to unknown, or unrecognised word. The system would advise the user and suggest some alternative words from the database that would be legal. The user may choose to use alternative suggestion, or may add the new word into the database, making the sentence parseable. This enables the database to expand and number of terminals increase.

7. sentence unparseable, in the last case, where the input sentence is complete nonsense.

We will now describe the examples of CRNLP examined in this experience.

8. Summary of Experiments on CRNLP

This section describes three CRNLP prototypes that have been built in Prolog that enable us to experiment on the development of CRNLP.

The series of SRS that had been studied are:

1. Fortran Library Routines Specifications (Mathematical Functions).

2. HP2100 Moving-Head Disc Operating System (DOS-M) User's Guide (Directives Command Specifications).

3. Tools for an Ada Measurement Environment (TAME) Requirements Document (Subsystem Specifications).

8.1 A Language for Fortran Library Routines

31/32/3300 Computer systems Library Routines is a manual that contains descriptions of FORTRAN library subroutines for the 3100, 3200, and 3300 computer. The purpose of each function is listed, along with the calling sequence, FORTRAN function, return, and error code.

To develop a language to write the specification for the routines, the purpose of each function is studied.

8.1.1 Design of Language

Step 1.

Grouping sentences where similarity can be identified.

1. ABS: to compute the floating point absolute value for a floating point number.

2. ALOG: to compute the natural algorithm of a floating point argument.

3. ATAN: to compute the floating point arctangent in radians of a floating point number.

4. AND: to find the logical product of the integer operands, A and B.

5. ITOX: to compute the result of an integer raised to a floating point power.

Step 2.

After rewriting the sentences:

1. ABS takes a floating point number and returns floating point absolute value.

2. ALOG takes a floating point number and returns natural algorithm.

3. ATAN takes a floating point number and returns floating point arctangent in radians.

4. AND takes integer, integer and returns logical product.

5. ITOX takes integer, floating point number returns floating point number raised to power.

Step 3.

Grammatical Rules:

Sentence --> take-part return-part
return-part take-part

take-part --> takes type {, type}

return-part --> returns type [result] [unit]
return-part --> returns [type] result [unit]
return-part --> returns [type] [result] unit

takes --> **takes**
| **converts**
| **finds**
| **from**
| ...
returns --> **returns**
| **computes**
| **calculates**
| **to**
| ...
type --> **floatpn**
| **fixedpn**
| **integer**
| ...
output --> **absolute-value**
| **natural-algorithm**
| **arctangent**
| **logical-product**
| **raised-to-power**
| ...
unit --> **in-radians**
| ...

Acceptable sentences are:

1. ABS: takes floatpn returns floatpn absolute-value
2. ALOG: takes floatpn returns natural-algorithm
3. ATAN: takes floatpn returns floatpn arctangent in-radians
4. AND: takes integer, integer returns logical-product
5. ITOX: takes integer, floatpn returns floatpn raised-to power.

Step 4.

A parser can be easily written in Prolog. We are more interested in a CRNLP rather than just a parser so that some semantics can be extracted from the input sentences.

The compiler extracts the relevant information from the sentences and uses this in interacting with user.

For each input sentence, its compilation will be in the form of **input declaration**, and **output declaration**.

[input-type], [output-type, output, unit]

inapplicable attribute is represented by **null**.

For examples:

1. takes floatpn returns floatpn absolute-value

compilation:
[floatpn**]**, **[**floatpn, absolute-value, **null]**

4. takes integer, integer returns logical-product

compilation:
[integer, integer**]**, **[null**, logical-product, **null]**

The sentences in database are stored in a list form, consisting of three components, ie. **statement id**, **input declaration**, and **output declaration**. In the format of:

srs (srs-id, [input-type], [output-type, output, unit]).

8.1.2 An Example Demonstration of CRNLP

Suppose we have the following statements already in the database, as Prolog clauses.

```
srs(abs,    [floatpn],          [floatpn,absolute-value,null]).
srs(alog,   [floatpn],          [null,natural-logarithm,null]).
srs(atan,   [floatpn],          [floatpn,arctangent,inradians]).
srs(not,    [integer],          [null,complement,null]).
srs(sqrt,   [floatpn],          [floatpn,squareroot,null]).
srs(and,    [integer,integer],  [null,logical-product,null]).
srs(itox,   [integer,floatpn],  [floatpn,raised-to-power,null]).
```

where

abs means **'takes floating-point number, returns floating-point number absolute-value'**,

and will be compiled as **'takes integer and integer, returns logical-product'**.

Program is invoked by calling '**parser**'.

?- **parser**.

statement: **takes integer returns complement.**
*** input type is *integer*
*** output is *complement*
statement is already in database: *not*

statement: **computes complement from integer.**
*** input type is *integer*
*** output is *complement*
statement is already in database: *not*

statement: **takes integer calculates integer complement.**
*** input type is *integer*
*** output type is *integer*
*** output is *complement*
statement looks like *not* in database
not takes integer returns complement
you might like to update database
or add into database as new function
type <m.> to modify database, or
type <a.> to add new function, or
type <q.> to quit: **m.**

not modified in database as:
not takes integer returns integer complement

statement: **takes integer returns integer square.**
statement uncompileable
--> **We would like to be able to handle this.**

statement: **takes integer returns squareroot.**
*** input type is *integer*
*** output is *squareroot*
statement compileable but not in database
do you want to add it into database? (y./n.) **y.**
name of new function: **sqrti.**
sqrti added into database

statement: **takes integer returns squareroot.**
*** input type is *integer*
*** output is *squareroot*
statement is already in database: *sqrti*

statement: **converts integer to integer squareroot.**
*** input type is *integer*
*** output type is *integer*
*** output is *squareroot*
statement looks like *sqrti* in database
sqrti takes integer returns squareroot
you might like to update database
or add into database as new function
type <m.> to modify database, or
type <a.> to add new function, or
type <q.> to quit: **a.**
name of new function: **sqrti2.**
sqrti2 added into database
statement:

8.1.3 Some Comments

1. This CRNLP is doing what we desired as a first step. It enables the user to know if a new statement can be written as one already in the database, to achieve some reuse.

2. The current construction of the system allows mathematical computation with some input parameter and only one output parameter.

The language can easily be enlarged by adding more terminals. It could be a language for all Functional Specifications which have the following format:

takes some-input and produce some-output.

3. This CRNLP stops and prints error message when it encounters some unknown word. We need a CRNLP that handles error-recovery.

8.2 A Language for DOS-M Directives

The Moving-Head Disc Operating System (DOS-M) (HP2100) User's Guide provides a list of function specifications of the directives. Directives are the direct line of communication between the keyboard or batch input device and the DOS-M.

The specification statements of the directives' functions were examined and a language is developed to write SRS for DOS-M directives.

8.2.1 Design of Language

Step 1.

Grouping sentences where similarity can be identified.

1. ABORT: terminate the current job.

2. DD: dump the entire current disc onto a disc.

3. LISTS: list all or part of a source statement file.

4. PDUMP: dump a program after normal completion.

Step 2.

After rewriting the sentences:

1. ABORT: terminate current job.

2. DD: dump entire current disc onto another disc.

3. LISTS: list all or part of source statement file.

4. PDUMP: dump program after normal completion.

Step 3.

Grammatical Rules.

Sentence	-->	Action [Quantity] Object [Description].
Description	-->	Conj Device [Condition]
	|	Condition
Action	-->	**terminate**
	|	**print**
	|	**dump**
	|	**list**
	|	...
Quantity	-->	**entire**
	|	**part_of**
	|	**all_or_part_of**
	|	...
Object	-->	**current_job**
	|	**program**
	|	**current_disc**
	|	...
Conj	-->	**to**
	|	**on**
	|	**from**
	|	**onto**
	|	...
Device	-->	**disc**
	|	**magnetic_tape**
	|	**subchannels**
	|	...
Condition	-->	**after_normal_completion**
	|	**normally**
	|	...

Acceptable sentences are:

1. ABORT: terminate current_job.

2. DD: dump entire current_disc onto another_disc.

3. LISTS: list all_or_part_of source_statement_file.

4. PDUMP: dump program after_normal_completion.

Step 4.

The sentences in database are stored in a list, consisting of two components, ie. statement id and statement information. In the format of:

srs (srs-id, [action, quantity, object, device, condition]).

For each input sentence, its compilation will be in the form of the second component.

[action, quantity, object, device, condition]

inapplicable attributes represented by **null**.

For examples:

1. terminate current_job.
[terminate, **null**, current_job, **null**, **null]**

2. dump entire current_disc onto another_disc.
[dump, entire, current_disc, another_disc, **null]**

3. list all_or_part_of source_statement_file.
[list, all_or_part_of, source_statement_file
null, **null]**

We realise the fact that NL is ambiguous. The ultimate goal of this project is to reduce the ambiguity in NL. Before we can solve the problem, we first need to identify the situation when two different statements actually mean the same thing.

In order to do this, we have a table of synonyms, where multiple words in the same category can be used to mean the

same thing. For example, **terminate** and **kill** mean the same thing as far as **job** is concerned.

Use of a limited synonym table has allowed a simple prototype to demonstrate the CRNLP.

8.2.2 An Example Demonstration of CRNLP

Suppose we have the following statements already in the database, as Prolog clauses.

```
srs(abort,      [terminate,null,current_job,null,null]).
srs(dd,         [dump,entire,current_disc,another_disc,null]).
srs(lists, [list,all_or_part_of,source_statement_file,null,null]).
srs(pdump,      [dump,null,program,null,after_normal_completion])
srs(rname,      [rename,null,file,null,null]).
srs(sa,         [dump,null,disc_in_ascii,standard_list_device,null]).
srs(sa,         [dump,null,disc_in_octal,standard_list_device,null]).
```

?- parser.

statement: **terminate current_job.**
*** action is *terminate*
*** object is *current_job*
statement already in database: abort

statement: **kill current_job.**
*** action is *kill*
*** object is *current_job*
statement equivalent to *abort* in database
abort terminate current_job

statement: **abort job.**
*** action is *abort*
*** object is *job*
statement equivalent to *abort* in database
abort terminate current_job

statement: **rename file.**
*** action is *rename*
*** object is *file*
statement already in database: *rname*

statement: **dump entire current_disc onto another_disc.**
*** action is *dump*
*** object is *entire current_disc*
*** device is *another_disc*
statement already in database: *dd*

statement: **dump current_disc to disc.**
*** action is *dump*
*** object is *current_disc*
*** device is *disc*
statement looks like *dd* in database
dd dump entire current_disc, device another_disc
you might like to update database
or add into database as new function
type <m.> to modify database, or
type <a.> to add new function, or
type <q.> to quit: **q.**
statement not added into database

statement: **dump all_or_part_of disc to disc.**
*** action is *dump*
*** object is *all_or_part_of disc*
*** device is *disc*
statement compileable but not in database
do you want to add it into database? (y./n.) **n.**
statement not added into database

statement: **dump disc_in_ascii to standard_list_device.**
*** action is *dump*
*** object is *disc_in_ascii*
*** device is *standard_list_device*
statement already in database: *sa*

statement: **dump disc_in_octal to standard_list_device normally.**
*** action is *dump*
*** object is *disc_in_octal*
*** device is *standard_list_device*
*** condition is *normally*
statement looks like *sa* in database
sa dump disc_in_octal, device standard_list_device
you might like to update database

or add into database as new function
type <m.> to modify database, or
type <a.> to add new function, or
type <q.> to quit: **a.**
name of new function: **san.**
san added into database

statement:

8.2.3 Some Comments

1. There is a problem associated with the structure of database. The **quantity** and **condition** parts do not seem to fit in very well. This problem is due to the non-uniformity of the statements. This is something that we have to take into consideration. In real life, SRS can be anything, and it is really hard to develop a language that could satisfy all kind of SRS.

2. One feature of the CRNLP is that it is able to look at an input statement and tell if there is any statement in the database which is associated to the statement. They could be talking about the same operation, or same type of operand or similar task on different object or maybe the new statement has more information than one in the database, or may be less, but still consists the essential details, etc.

In order to do this, the CRNLP, after compiling a statement, will go through the whole database of statements, matching them with the new one, and try to identify any similarity.

This seems like pure pattern-matching and is actually how the basic idea works. But there will be problems in identifying or justifying how to decide if two statements are similar enough. There should be some higher level of abstraction. This will be discussed in Section 10.

8.3 A Language for TAME Subsystems Specs

TAME, 'Tools for an ADA Measurement Environment', is a systems requirement document that was written in a standard IEEE SRS format. This, by the way, was produced by Karl

Reed, and stimulated the original SODA concept. It was the section of 'Subsystem Requirements' where many similar statements appears at many places in the SRS that was used to produce this CRNLP.

In fact, this CRNLP was obtained by taking the SRS for all subsystems and examining them manually. Common operations and operands were identified, and then propagated across all specifications.

The 'objectives' part of each 'Specific Functional Requirement' of the 'Subsystem Requirements' were studied and a language was the result of the study.

8.3.1 Design of Language

Step 1.

Grouping sentences where similarity can be identified.

Group A.

1. to allow user to create, edit, maintain and display evaluation models which will be held in the TAME DBS.

2. to allow user to create, edit, maintain and display result of analysis which will be held in the TAME DBS.

3. to allow user to create, edit, maintain and display scheduling information which will be held in the TAME DBS.

4. to allow user to create, edit, maintain and display security profiles which will be held in the TAME DBS.

5. to allow user to create, edit, maintain and display forms which will be held in the TAME DBS.

Group B.

1. to produce intermediate output capable of being processed by the g/q/m/A to evaluate the model.

2. to produce output which can be stored in the TAME database subsystem with appropriate links to current source code so that suitable configuration management can be achieved.

3. to produce output in a form that can be used by the Report Generator.

4. to place result in DBS in a form suitable for use by other tools and Report Generator.

5. to present data in a form that can be used by the g/q/m/A.

6. to produce output capable of being presented to the g/q/m subsystem for display and for further analysis.

Some Comment:

The repetitive feature is obvious and clear, and yet there is an appropriate richness in the forms of expression used. Our view is that this may be suitable for the CRNLP specification.

Step 2.

After rewriting the sentences:

<u>Group A.</u>

1. to allow user to create, edit, maintain and display evaluation models which will be held in tame dbs.

2. to allow user to create, edit, maintain and display result of analysis which will be held in tame dbs.

3. to allow user to create, edit, maintain and display scheduling information which will be held in tame dbs.

4. to allow user to create, edit, maintain and display security profiles which will be held in tame dbs.

5. to allow user to create, edit, maintain and display forms which will be held in tame dbs.

Group B.

1. to produce intermediate output capable of being processed by gqma to evaluate model.

2. to produce output which can be stored in tame dbs with appropriate links to current source code so that suitable configuration management can be achieved.

3. to produce output in a form that can be used by the report generator.

4. to place result in dbs in a form suitable for used by other tools and report generator.

5. to present data in a form that can be used by gqma.

6. to produce output capable of being presented to gqm subsystem for display and for further analysis.

Step 3.

Grammar A.

Sentence --> **to** Allow Agent **to** Actions Object [Optional]**.**

Actions --> Action {**,** Action}
| Action {**and** Action}

Optional --> **which will be held in** Place

Allow --> **allow**
| ...

Agent --> **user**
| **operator**
| ...

Action --> **create**
| **edit**
| **maintain**
| **display**
| ...

Object --> **evaluation_model**
| **result_of_analysis**
| **scheduling_information**
| **security_profiles**
| **forms**
| ...

Place --> **tame_dbs**
| ...

Acceptable sentences are:

1. to allow user to create, edit, maintain and display evaluation_models which will be held in tame_dbs.

2. to allow user to create, edit, maintain and display result_of_analysis which will be held in tame_dbs.

3. to allow user to create, edit, maintain and display scheduling_ information which will be held in tame_dbs.

4. to allow user to create, edit, maintain and display security_profiles which will be held in tame_dbs.

Grammar B.

Sentence --> **to** Produce-Part Pipe Process-Part [Extra-Part]**.**

Produce-Part --> Produce Output
Process-Part --> Processed Objects

Objects --> Object {**and** Object}

Extra-Part --> **to** Eval Model
| **with** Link **to** Code **so that** Mgt **can be** Act
| **for** Function {**and** Extra-Part}
| ...

Produce --> **produce**
| **present**
| **place**
| ...
Output --> **output**
| **intermediate_output**
| **result_in_dbs**
| **data**
| ...
Processed --> **processed_by**
| **stored_in**
| **used_by**
| **presented_to**
| ...
Object --> **gqma**
| **tame_dbs**
| **report_generator**
| **gqm_subsystem**
| **other_tools**
| ...
Pipe --> **capable of being**
| **which can be**

| **in a form that can be**
| **in a form suitable for**
| ...
Eval --> **evaluation**
| ...
Model --> **model**
| ...
Link --> **appropriate_links**
| ...

Code --> **current_source_code**
| ...
Mgt --> **suitable_configuration_management**
| ...
Act --> **achieved**
| ...
Function --> **display**
| **further_analysis**
| ...

Acceptable sentences are:

1. to produce intermediate_output capable of being processed_by gqma to evaluate model.

2. to produce output which can be stored_in tame_dbs with appropriate_ link to current_source_code so that suitable_configuration_management can be achieved.

3. to produce output in a form that can be used_by the report_ generator.

4. to place result in dbs in a form suitable for used_by other_tools and report_generator.

5. to present data in a form that can be used by gqma.

6. to produce output capable of being presented_to gqm_subsystem for display and for further_analysis.

Step 4.

The first CRNLP built to illustrate the analysis of this small system was not satisfactory however it raised a number of important issues concerning the approach being taken. This will be discussed further in Section 11.3.

For Group A, the typical statement compilation format is:

[user, action, ..., object, place]

For example:

'to allow user to create, edit, maintain and display evaluation_model which will be held in tame_dbs.'

will be compiled as

[user, create, edit, maintain, display, evaluation_model, tame_dbs]

It will then be matched against the database of existing statements to see if similarities could be identified.

The concept of similarity is based upon the object a statement is specifying. Therefore, any compileable sentence with the word '**evaluation_models**' will be detected by the above statement.

For Group B, the typical statement compilation format is:

[produce, output, processed, object, etc]

where **etc** is a list of other extra information, with arbitrary length.

For example,

'to produce intermediate_output capable of being processed_ by gqma to evaluate the model'

will be compiled as

[produce, intermediate_output, processed_by, gqma, evaluate, model]

In this case, similarity will be based on the 'processed' and 'object' parts of the statements.

8.3.2 An Example Demonstration of CRNLP

Suppose we have the following statement already in the database, as Prolog clause.

```
srs(a1,[user,create,edit,maintain,display,evaluation_models,
tame_dbs],[evaluation_models]).
```

which means:

statement 'a1' specifies: to allow user to create, edit, maintain and display evaluation_models which will be held in the tame_dbs.
and the important key word in 'a1' is 'evaluation_models'.

?- **parser.**

tame: **to allow user to create, edit, maintain and display evaluation_model which will be held in tame_dbs.**

*** user create edit maintain display evaluation_model tame_dbs.
statement is already in database: srs1.

tame: **to allow user to create, edit, maintain evaluation_model which will be held in tame_dbs.**

*** user create edit maintain evaluation_model tame_ dbs.
statement looks like srs1 in database.
srs1: user create edit maintain display evaluation_model tame_dbs.
if they are different, do you want to update database? <y./n.>
y.
<n.> for new statement or <m.> to modify database: **m.**
srs1 modified in database as:
srs1: user create edit maintain evaluation_model tame_ dbs.

tame: **to allow user to create, edit, maintain result_of_analysis which will be held in tame_dbs.**

*** user create edit maintain result_of_analysis tame_dbs.
statement compileable but not in database,
do you want to add it into database? <y./n.> **y.**
id of sentence: **srs2.**
statement srs2 added into database.

tame: **to allow user edit evaluation_model.**
statement unparseable.
*** syntax error: *to* expected

8.3.3 Some Comments

Writing the parser was not difficult, provided the grammar is given and well defined. There were, however, some problems encountered.

1. This CRNLP reads in a statement, extracts the relevant information and stores it into the database, if the statement is acceptable as far as the language is concerned

For example:

'to allow user to create, edit, maintain and display evaluation_model which will be held in tame_dbs.'

will be compiled as

[user, create, edit, maintain, display, evaluation_model, tame_dbs]

which is a Prolog list that contains all the essence of the sentence.

However, we need a way to 'properly' specify the class of the elements in the statement. For instance, 'user' is an agent, 'create, edit, maintain, display' are operators and 'evaluation_model' is operand, and 'tame_ dbs' a place.

2. The compiler should be able to identify the reusable components in statements in database. The method employed in developing this CRNLP is to have an extra attribute for each statement. This attribute contains the key element in the statement. During a new statement reusable component search, the statement with the key word will be identified if the it exists in the new statement

```
s1 -- [user, create, edit, maintain,
display, evaluation_ model, tame_dbs]
```

and a new statement after compilation,

```
s2 -- [user, edit, display, evaluation_models,
tame_dbs]
```

The extra attribute of 's1' is [evaluation_models] as specified in the database. The new statement consists of the word 'evaluation_models' and hence the two must be related, even though they are not identical. The system is able to do this automatically, but the method adopted was not desired.

9. More Facilities

9.1 Expanding CRNLP

This is what the above-mentioned compiler would do:

/*
tame: **to allow user to edit picture.**
statement unparseable.
unknown word: picture
*/

This was our next goal, and was achieved:

/*
tame: **to allow user to edit picture.**
statement unparseable.
unknown word: *picture*

try one of the following:
evaluation_model
result_of_analysis

add new word picture <y.> or use alternative <n.> **y.**
picture added into database.

*** *allow user edit picture*

statement compileable but not in database,
do you want to add it into database? <y./n.> **y.**
id of sentence: **srs3.**
statement srs3 added into database.

tame:
*/

This is how the database dictionary grows, and so does the language. Besides, this is also the error-recovery function of the CRNLP.

Method:

1. When a sentence is not parseable due to an unrecognised word, the system will provide the user with a list of synonyms and related words.

2. The user can choose whether to use an alternative or add the word into the system dictionary, so that the sentence becomes compileable, and hence expanding the database.

This is very important because our main objective of this project is to develop a restricted language, we need to restrict and limit the dictionary of the system, as to reduce ambiguity.

By providing alternative choice to the user, the user can make use of the words in the database to write the statements, hence achieving some aspect of reuse.

9.2 Statement Identification

The system is to identify the reusable components in an SRS and therefore an appropriate way to identify statements in database, as well as new statements, this is an important issue.

In the previous processors, naming of statements is simply arbitrary, and names of new statements were to be provided by user. It is realised that user has nothing to do with the identification of statements.

Statement id should be system-generated. Reason being the statement ids only tell where the statement appears in which SRS.

As we shall see in Section 9.3 and Section 11, each statement is recognised by a SRS-ID and a line number specifying which SRS it is in and which line in SRS it appears.

9.3 Reusable Vocubulary

Periodically, the system implementor needs or would like to know where some words or statements have already appeared in the system specification. The CRNLP provides a facility to take word or words from the user to let the user know where the words have occurred in the SRS in the database.

Method:

1. Each statement in an SRS is identified by its line position in the SRS (where a sentence takes a line), and each SRS is identified by its SRS-ID.

2. When a statement is stored into the database table, the first two attributes of the statement will have to be the SRS-ID and its line number to ensure each statements are distinct from each other.

3. When user queries the system with one or more words, the system will search through the whole database looking for the words. Then all the statements with those

words will be identified and their ids will be printed for user.

An illustration:

```
srs(sa,1,[user,create,...,evaluation_models,...]).
srs(sa,2,[user, edit,...,data,...]).
srs(sa,3,[user,create,edit,display,forms,...]).

words: edit.
statement #sa  line #1
statement #sa  line #2
statement #sa  line #3

words: evaluation_models.
statement #sa  line #1
```

This enables the user to know if he may use a word that he has in mind before he uses it. One way to reduce ambiguity in SRS is to have only one way to say one thing, and if user uses the same word in two places or more, he must mean exactly the same thing.

When the user thinks of using a word in a statement, he may query the system to see where the word has been used, and whether the meaning of that word is the same as what he is having in mind. If not, he would have to think of another word to specify his thought, or add it to the database.

10. Higher Level Abstraction Of Data

10.1 Limitation

The prototypes of CRNLP so far were built in Prolog. The table used for storing statements in the database is a set of lists containing information of statements. The CRNLP was able to identify similar statements with the simple statements we were looking at. However, this simple data structure would not be able to handle a more complicated SRS, and a more complicated data structure would then be required.

Despite the fact that only a few types of similarities in SRS sentences are recognised, we have a SRS language which has quite a high degree of nature feel, and is fairly effective.

There are all kinds of sentences, though they are all in the specific domain of Software Engineering. The types of similar sentences the CRNLP is able to recognise are:

1. sentences with the same object, or same class of object.
2. sentences with same operations on different objects.
3. sentences with same operation on different operands, but of same class.
4. sentences with same type of operands but different types of operations.
5. command type sentences.
6. mathematical functional sentences.
7. others

The way the earlier CRNLP was written was a very simple-minded pattern matching method that allowed only a very small subset of statements of some small SRSs. It was to produce a short-term illustration and in a long-term project, proper level of data abstraction would be required.

10.2 High Level Abstraction

We need to classify words in database to a higher order of abstraction as well as statements that are to be stored in the table. That means, instead of compiling the statements for important words (that reflect the relevant information of the statements), we should be looking at compiling the structure and then the words of each statement. Instead of storing the lists of words of statements, we should be storing the abstraction of the essence of the statements into the database [8].

During the procedure of looking for similar statements from the database, the system should first look for the generic structure that matches the frame of the statement. If that is not successful, then there is no similar statement. Otherwise, the system then looks for similarities by identifying the same words used in statements [9].

This is the appropriate approach that should be employed, and another CRNLP was written to check its feasibility (this will be discussed in the next section.)

10.3 Application of CRNLP for the Language Derived from Tame Subsystems Specs

The idea of high level abstraction of elements in a statement was put onto trial on a portion of the language derived from the Tame Subsystems Specifications.

Some sentential examples of the language:

1. to allow user to create, edit, maintain and display evaluation_models which will be held in tame_dbs.

2. to allow user to create, edit, maintain and display result_of_analysis which will be held in tame_dbs.

The old compiler would produce:

1. user create edit maintain display evaluation_models tame_dbs

2. user create edit maintain display result_of_analysis tame_dbs

A CRNLP with higher level of abstraction would produce:

1. [agent([user]), action([create,edit,maintain,display]), object([evaluation_models]), place([tame_dbs])]

2. [agent([user]), action([create,edit,maintain,display]), object [result_of_analysis]), place([tame_dbs])]

This is made possible by having all the words in the dictionary classified according to their appearances in the sentences. During the parsing procedure, words will be categorised accordingly, producing the above format of compilation.

This is also the way a statement is stored into the table. Given the compilation of a new statement, identifying similarity will be easier and correct. Two sentences will be

matched for identical (with some appropriate constraints) structure, then for same words in the frames.

In some cases, two sentences could be considered as being associated to each other for some significant similarity but having different generic structures. This is one of the aspects this project should look into. Two 'similar' statements should be written to 'look alike', or at least have the same format. Otherwise, appropriate constraints will be needed in identifying similarities and dissimilarities.

This CRNLP first parses the sentence, classifies and identifies each word. If the parsing is successful, it then searches the database for similarities. For each statement in the database, it tries to pattern match them with the new compiled statement. By doing this, it is like another parsing procedure, where the compilation is parsed against the database. This is not desired, and we need a better way of identifying similarity between statement and statements in database.

We realised that a good parser will be able to handle this with a good parsing procedure. It would be good to re-write the whole CRNLP in Lex and Yacc. The reason being that the compiler written in Prolog is a recursive descent parser, and a deterministic parser is needed.

However, due to time constraint, it was wise to push on by compromising this situation. The next goal was of course to come up with language that enables a format of standard SRS to be proposed. This will be further discussed in the next section.

11. Further Development of Language Into IEEE Standard Requirements Documents Format

11.1 Current CRNLP Prototype

The current prototype is able to read a SRS that contains a number of statements and perform semantics accordingly to each statements. Each of these statements will be recognised by the system as statements in that particular SRS. Therefore the SRS will have a identification or name.

The current prototype has the following Grammatical Rule:

Statement	-->	**start** statement-id Sentences **end** statement-id**.**
Sentences	-->	{ Sentence**;** }
Sentence	-->	the sentences specified above, not ended with '.'

For example:

start spec1

to allow user to create, edit, maintain and display evaluation_models which will be held in tame_dbs;

to allow user to edit, maintain and display result_of_analysis;

to allow user to create, edit and display scheduling_ information which will be held in tame_dbs;

end spec1.

An Example of CRNLP Process

- a.?

statement <s.> or query <q.> **s.**
tame statements:

start srs
to allow user to edit picture;
to allow user to draw picture;
to allow user to draw and display picture;

end srs.

specification: srs

line #2
statement not parseable
*** unknown word: picture
try one of the following:
evaluation_models
result_of_analysis
security_profiles
scheduling_information
forms
add new word <n.> or use alternative <a.> **n.**

*** agent([user])
action([edit])
object([picture])

statement compileable but not in database
add into database? <y./n.> **y.**
srs2 added into database

line #3
statement not parseable
*** unknown word: draw
try one of the following:
create
make
edit
change
maintain
update
display
report
add new word <n.> or use alternative <a.> **n.**

*** agent([user])
action([draw])
object([picture])

associated statement identified:
srs2: agent([user])

action([edit])
object([picture])

update database? <y./n.> **y.**
new statement <n.> or modify statement <m.> **n.**
srs3 added into database

line #4
*** agent([user])
action([draw,display])
object([picture])

associated statement identified:
srs2: agent([user])
action([edit])
object([picture])

update database? <y./n.> **y.**
new statement <n.> or modify statement <m.> **m.**
srs2 deleted from database
srs4 added into database

statement <s.> or query <q.> **q.**
words: **picture.**
statement #srs line #3
statement #srs line #4

11.2 IEEE Standard Documents Format

An American National Standard
IEEE Guide to Software Requirements Specifications [10]

Prototype SRS Outline:

Table of Contents
1. Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, and Abbreviations
1.4 References

1.5 Overview
2. General Description
2.1 Product Perspective
2.2 Product Functions
2.3 User Characteristics
2.4 General Constraints
2.5 Assumptions and Dependencies
3. Specific Requirements
...
Appendixes
Index

Prototype Outline 4 For SRS Section 3 [10]

3. Specific Requirements
3.1 Functional Requirements 1
    3.1.1 Introduction
    3.1.2 Inputs
    3.1.3 Processing
    3.1.4 Outputs
    3.1.5 External Interfaces
        3.1.5.1 User Interfaces
        3.1.5.2 Hardware Interfaces
        3.1.5.3 Software Interfaces
        3.1.5.4 Communication Interfaces
    3.1.6 Performance Requirements
    3.1.7 Design Constraints
    3.1.8 Attributes
        3.1.8.1 Security
        3.1.8.2 Maintainability
        . . . .
    3.1.9 Other Requirements
        3.1.9.1 Data Base
        3.1.9.2 Operations
        3.1.9.3 Site Adaption
        . . . .
3.2 Functional Requirement 2
. . . .
3.n Functional Requirement n

The proposed SRS language would be guided by the above outline.

11.3 The SRS Language

Our ultimate goal is to develop an SRS language that has the format of a Standard IEEE SRS Document. In order to do this, nested SRS structure would be required, and SRS-id's would be the section numbers used in the above Outline.

This is the proposed Grammatical Rule for the 'Specific Requirements' section of an SRS.

SP --> { FR-ID **Functional Requirements** FR-NO
FR-ID **Introduction** STATS
FR-ID **Inputs** STATS
FR-ID **Processing** STATS
FR-ID **Outputs** STATS
FR-ID **External Interfaces**
FR-ID **User Interfaces** STATS
FR-ID **Hardware Interfaces** STATS
FR-ID **Software Interfaces** STATS
FR-ID **Communication Interfaces** STATS
FR-ID **Performance Requirements** STATS
FR-ID **Design Constraints** STATS
FR-ID **Attributes**
FR-ID **Security** STATS
FR-ID **Maintainability** STATS
{ MORE-ATTR}
FR-ID **Other Requirements**
FR-ID **Data Base** STATS
FR-ID **Operations** STATS
FR-ID **Site Adaption** STATS
{ MORE-OREQ }
}

MORE-ATTR --> FR-ID NAME STATS

MORE-OREQ --> FR-ID NAME STATS

STATS --> { SENTENCE }

SENTENCE--> Sentence as in Section 8

FR-ID, FR-NO : number or some alphanumeric identifier

NAME : string of characters as title name

Similarly, the Grammatical Rule of SRS would look something like this:

SRS --> **Table of Contents** TEXT
SEC-ID **Introduction**
SEC-ID **Purpose** TEXT
SEC-ID **Scope** TEXT
SEC-ID **Definitions, Acronyms and Abbreviations** TEXT
SEC-ID **References** TEXT
SEC-ID **Overview** TEXT
SEC-ID **General Description**
SEC-ID **Product Perspective** TEXT
SEC-ID **Product Functions** TEXT
SEC-ID **User Characteristics** TEXT
SEC-ID **General Constraints** TEXT
SEC-ID **Assumptions and Dependencies** TEXT
SEC-ID **Specific Requirements** SP
Appendixes TEXT
Index TEXT

SEC-ID: alphanumeric identifier

TEXT: some sentences, tables, diagrams, or index, format not discussed in this paper, where we are mainly interested in the section of Specific Requirements Specifications Statements.

12. Conclusion

From the research and implementation of CRNLP prototypes, we can conclude that a compileable restricted NL-like language for writing SRS is possible.

However, this is only a starting point, with prototypes illustrating a portion of the language. For a real CRNLP, it should be general enough to be applicable to any kind of SRS, regardless the nature of Software Requirements.

13. References

[REED'91] Reed, K. (1991) "Some Aspect of the Possible Automation of the Use of Natural Language in Software Development". *AAITP Technical Note 01/91 Australia*.

[GOMA'81] Gomaa, H, Scott, D B H. (1981) "Prototyping as a Tool in the Specification of User Requirements". *The Preceeding of the 5th International Conference on Software Engineering, pp 333-342, 1981*.

[BELL'76] Bell, T, Thayer, T. (1976) "Software Requirements: Are They Really a Problem"? *Proceedings of the 5th International Software Engineering Conference, Oct 1976*.

[BELL'79] Bell Telephone Laboratories, (1979) *"Unix Programmer's Manual, 4.2BSD" Section 1 & 2. RMIT Computer Centre, 1979*.

[CONT'64] Control Data Corporation, (1964) 31/32/3300 Computer System (Fortran) Library Routines, USA 1964.

[HEWL'76] Hewlett Packard 2100A, (1976). "Moving Head Disc Operating System (DOS-M) User's Guide". *Department of Computer Science, 1976*.

[ROMB'86] Rombach, H D, Turner, J, Reed, K, (1986) "Requirements Document for TAME (Tools for an ADA Measurement Environment)", *University of Maryland, USA, 1986*.

[COAD'91] Coad, P, Yourdan, E, (1991). "Object Oriented Analysis", *Prentice-Hall, USA, 1991*.

[MATS'84] Matsumoto, Y. (1984) "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels", *IEEE Transactions on Software Engineering, pp 502-513, Vol. SE-10, No.5, Sept 1984*.

[IEEE'84] IEEE, (1984). "An American National Standard, IEEE Guide to Software Requirements Specifications", *ANSI/IEEE Std 830-1984, IEEE, 1984*.

[ACM'88] ACM, (1988). "A Comparison of Techniques for the Specification of External System Behaviour". *Communication of the ACM, pp 1098-1115, Vol. 31, No.9, 1988*.

[MYNA'90] Mynatt, B T, (1990) "Software Engineering with Student Project Guidance", *Prentice-Hall, pp 170-173, 1990*.

[FREE'87] Freeman, P. (1987) "Tutorial: Software Reusability", *IEEE, USA, 1987*.

[TRAC'88] Tracz, W. (1988) "Tutorial: Software Reuse: Emerging Technology", *IEEE USA, 1988*.

[AGRE'86] Agresti, W. (1986) "Tutorial: New Paradigms for Software Development", *IEEE, USA, 1986*.

[COOP'85] Cooper, D, Clancy, M. (1985) "Oh! Pascal!", *Second Edition, pp 561-566, Nortan & Company, Inc. USA, 1985*.

[CLOC'81] Clocksin, W, Mellish, C, (1981) "Programming in Prolog". *Springer-Verlag Berlin Heidelberg,Germany, 1981.*

[MARC'86] Marcus, C. (1986) "Prolog Programming: Applications for Database Systems, Expert Systems and Natural Language Systems, *Arity Corporation, USA, 1986.*