Extending Visual Programming:

Graphical Component Based Design

TR029

by

Jason Baragry & Karl Reed

Amdahl Australian Intelligent Tools Programme

Dept of Computer Science and Computer Engineering

La Trobe University, Bundoora

Victoria, Australia 3083

INDEX

Abstract

This paper argues that a major weakness in the software development process is the absence of a coherent Graphical Component Based Design methodology. We examine existing, re-use oriented engineering design approaches and conclude that a major feature on many of these is the interaction between graphical process of design and the sets of pre-defined components available. We also argue that, in general, diagrammatic representation of a system is the system itself, since no further design work is required to implement that system.[1] This is so because the diagrams consist of direct representations of the building blocks of that discipline. Software design diagrams, on the other hand, represent abstract concepts such as processes and states and arbitrarily chosen procedures, rather than actual software components. It is suggested that this prevents designers from constructing software diagrams in the same manner and with the same intent and result as in other engineering disciplines. There are usually significant, non-deterministic processes required to complete the implementation of a software system given its final diagrams.

The paper goes on to suggest that visual programming languages allow the construction of diagrams which represent a completed design directly. However, we propose that their low level of granularity be raised to allow "larger" components to be represented. This, it is proposed, will lead to a Graphical Component Based Design methodology for software, combining the cognitive advantages of visual programming with the economic and efficiency advantages promised by software reuse. We demonstrate this, developing a component based visual language for the proprietary programming language Huron, Amdahl Corporation's rule-based development vehicle for large-scale data intensive systems. The prototype shows how a complete graphical development environment would provide links between the component level of design (graphical components) and the source code level of design (visual languages).

[1] In making this observation, we accept there may be additional steps needed to realise the design physically. For example, it may be necessary to produce a wiring or lay-out diagram of an electronic circuit showing the physical juxtapositioning of the components.

1. Introduction

1.1. Limitations of Software Diagramming Techniques.

Diagramming techniques are an established part of software development, having been in continuous use since the beginning of programming [1]. Despite the existence of a large number of diagramming systems, there has been very little research directed at either measuring their effectiveness or their relationship to diagramming techniques used in traditional engineering disciplines.

Diagramming forms an integral part of engineering design. Most designers will think in terms of images rather than words, and many great designers have also been great draftsman - Leonardo da Vinci being an outstanding case [2]. There are a number of specialised types of drawings in the traditional engineering disciplines which could be replaced by calculations (and are usually checked by them), but often the calculation does not provide the all important physical insights the drawing does.

While the methods used to develop engineering systems are varied and are still being investigated [3], the diagramming techniques used to represent the designs during system development generally have an important property - the components represented in the design diagram are either direct representations or have a simple mapping to the components used to implement the system in that engineering discipline. The design diagram is therefore a direct representation of the implementation. As a consequence, when a designer completes the diagrammatic design of a system, no further design work is required to implement that system. At most, there may be an implementation step which involves describing the physical interconnections of components shown in a circuit diagram for instance. In many cases, analysis techniques required to test the system and establish its integrity can be applied to the design diagram using simulation techniques (cf. digital electronics, automotive component testing, distribution systems) or are already implied by the design process (cf. civil, mechanical engineering, etc.). We argue that such diagrams could be considered to be 'executable' [4] in the sense that they are complete representations of the system requiring no further design activity.

Software engineering is often compared with other engineering disciplines. This reflects the widely held view that software development may become more tractable if it becomes an engineering discipline. Amongst the particular aspects of engineering practice which receive continuous attention are those relating to the impact of components on design [5], and the nature of re-use in design [6]. In our view, these two issues are linked, and, we now add to them the issue of diagramming systems and their relationship to components, and their impact on the design process.

Graphical design constitutes a major part of engineering design. In civil, mechanical, structural (etc) engineering, it performs two functions. It allows the physical interconnections of components to be determined, and to be described. In this (and other cases) the result is a representation of the final artefact which shows directly its physical appearance. This must be contrasted with electronic, electrical, and process control engineering in which the complete design may be represented by a schematic diagram which shows the interconnections between and the nature of components but not always their actual physical arrangement. We believe traditional software diagramming systems are similar in conceptual function to these, but that they are not used in the same manner as traditional engineering systems. This is due to their nature, the design methods with which they are used, and the absence of predefined components.

A major factor limiting the effectiveness of some software diagramming systems is that the symbols used do not represent pre-defined components. In some cases, (eg., data-flow-diagrams, state-charts and structure charts) component-based design could be used while in others (eg. state-transition-diagrams, petri-nets and Warnier-Orr diagrams), the very nature of the concepts represented precludes this. While the existence and use of pre-defined components may not be an inherent property of any particular diagramming system, there is no doubt in our mind that their presence makes a major contribution to their effectiveness. In fact, the existence of these predefined components is a major attribute of engineering disciplines such as electronics and helps determine the design strategies.

Fundamental problems also exist in terms of the definition and properties of a 'component'[2] and their impact on design. In some areas of engineering, the use and influence of components is so persuasive that it could be said that those disciplines use a "graphical component based approach" to system development. We believe that this must be integrated into software development techniques if they are to realise the same levels of component reuse. Some clarification of this remark is required, since we are assuming that some volume of functionality is encapsulated in the component. We do this for the simple reason that if the definition of a 'component' is simple enough to incorporate any of the building blocks the developer uses to design a system, programming language constructs can be considered to be software components. In fact they are analogous to the use of low-level components such as resistors, capacitors etc in electronics and beams, reinforcing rods, and concrete in civil engineering for example. There are clearly diagramming systems in other engineering disciplines that operate at that level.

In Section 2 we argue that visual languages can be considered to be the beginnings of a graphical component based development approach for software. The difference is software components have been traditionally stuck at the level of the source code while the graphical-component-based design engineering disciplines have developed components with a larger granularity.

[2] This is part of on-going research in AAITP.

From this point of view, the visual languages are closer to the engineering drawings used in the construction industry than in digital electronics. Our research goal is to develop a graphical component based approaches to software development which utilise pre-defined components that are of a larger granularity than the programming level constructs we presently develop with, coupled with a diagramming systems which allows other attributes of the system structure to be visualised.

1.2. Towards a Graphical Component Based Approach to Software Development

This paper presents the concept of Graphical Component Based Design for Software (GCBDS) as an evolutionary step from visual programming, which incorporates the cognitive advantages of visual development with the efficiency benefits achieved through the utilisation of large scale software reuse. GCBDS is intended to be similar to the development approaches employed by other engineering disciplines, such as electronic and mechanical engineering, who utilise Graphical, Component-Based (GCBD) approaches to system development and achieve high levels of component reuse.

2. Diagramming Techniques and Methods in Software Development

Diagramming techniques used in software development are simply the visual manifestations of the components / concepts we manipulate at a particular level of the prescribed development process. Using this basic definition there are two broad categories into which software development diagram systems can be classified[3] : design level diagrams and visual languages.

2.1. Design Level Diagrams

Traditional software development methods have emphasised a 'top-down' approach to system design. These methods were adequate for producing systems from scratch or for novice designers with no previous experience. However, top-down approaches do not arrive at reusable components [7] nor do they preserve the series of abstractions through which the developer created the system [8]. The top-down approaches assume no systems have been previously developed and no usable components exist above the programming language constructs. Consequently these approaches manipulate abstract concepts, such as processes, states and arbitrary procedural entities, and refine these until the developer can implement these abstract concepts using the assumed highest level implementation components - programming language constructs.

Graphical tools have been developed to represent these abstract concepts. There are a large number of computer based tools which have allowed these essentially paper-based methods to be handled by computer.[4] However at no stage are the graphical objects a direct representation of an implementation component nor is there a simple mapping from the design diagram to the 'physical' implementation medium. Consequently, there is a discernible cognitive gap between the design and implementation stages of software development. This manifests itself in two ways. Firstly, the level of abstraction between levels in a system description, and, in the absence of prescriptive design procedures for moving from a higher level to another. The information needed to bridge this gap between remains in the developers head rather than being recorded in the system design / implementation diagrams[5] , and the steps necessary to perform the transformations (from one level to another) as loosely defined informal process. Consider, for example, the

[3] Naturally, as soon as any classification scheme is mentioned someone will develop or produce a technique which blurs the boundaries. The basic classification should be able to be applied to most general software diagramming techniques.

[4] Most common diagramming systems pre-date both CASE and SDE's.

[5] Of course, a designer can record this information in the system's documentation but this rarely happens.

following 'definition' of the point to which data-flow diagram processes should be refined .

"How long should levelling continue for? Generally, until a set of processes that can be described by about one page of detailed process specifications are reached" [9].

This has led these design level diagramming techniques to be labelled as merely documentation tools [10].

2.2. Visual Languages

Although a concise definition of what is a visual language is hard to find, it is generally agreed that it consists of the use of graphical notions, such as icons, either exclusively or at least as a major component, in the specification of a computer program.

Visual languages differ from traditional software design level diagrams because the graphical objects they manipulate are actual representations of the components used to implement the system. In this regard they are similar to the diagramming techniques of traditional engineering disciplines, ie., "no additional design work is required to implement the system". The use of visual languages to create software systems is increasing, especially in very specific application domains. This can be seen in Glinert's books [11] [12]. Visual languages have not yet received wide acceptance for the development of general purpose software systems. However, tools such as Prograph [13] are gaining an increased utilisation and are promoting the potential benefits of graphical development techniques over their textual counterparts[6] . Despite this, it has been suggested that visual languages suffer from two major deficiencies which have stopped them being widely utilised [14]: (a) they cause a lot of screen clutter, and (b) they do not scale up for use in the development of general purpose, large-scale software systems. We believe this is due to their failure to provide an environment which facilitates the promotion, creation, and utilisation of larger granularity components.

The "screen clutter" is likely to be due to the fact that the graphical components used are often of a granularity close to that of programming language statements. For instance, a visual language program which represents an additional symbol in its own icon is obviously going to be larger than its textual counterpart. Visual languages will be utilised because of their cognitive advantages over textual methods when they begin to represent components of a larger granularity than existing code.

It has been previously stated that visual programming is a primitive, graphical component based approach to design where the objects depicted in the visual language are of a granularity similar to that of programming language

[6] If the creation of a dedicated UseNet newsgroup, comp.lang.prograph, can be used a measure ot this popularity or utilisation.

constructs. In order to provide a visual development environment for the creation of general purpose, large-scale applications the visual language must represent software components of a larger granularity than is currently available. As traditional engineering disciplines have matured, the complexity of the systems being produced has increased. The efficiency of the discipline's development process has subsequently improved by increasing the granularity of the components thereby reusing the knowledge gained in producing the original systems. The diagramming techniques used in system development merely represent the components used by the discipline regardless of their granularity.

Our view is that efficient diagramming techniques depend in part on the existence of pre-defined components encapsulating some abstract, higher level of functionality. (Higher than the atomic level of granularity, and abstract in the sense that some level of reuse across applications is needed). The problem is creation of these higher level components and the standardisation of those components for use by other developers. There are two impediments, as we have pointed out. The first is the absence of standard module producing design strategies, the second is the absence of component based design. Large scale reuse is sometimes achieved in well defined problem domains, such as user interface development and operating system interfaces, because it is possible to have well defined and universally understood pieces of functionality. With a standard set of high level functional components it has been seen how visual languages can be used in the development of large scale applications in a limited problem domain. For visual languages to be utilised for the development of large scale general purpose software systems, a graphical component based approach to development is required which incorporates a collection of pre-defined and understood components which have a granularity larger than the traditional programming language constructs.

3. A Graphical Component Based Design Approach to Software Development

We will now outline the issues involved in developing a Graphical Component Based Design approach which is intended to make some of the component based approaches used by traditional engineering disciplines accessible to software designers. Our approach is to use graphical notations to support a design process which uses predefined building blocks, accepted as standard within some section of the discipline, instead of the arbitrary abstract concepts yielded by current software methodologies. Our belief is that this will result in significantly higher levels of reuse, at least at the lowest levels, while improving the understandability (and hence the maintainability) of systems. The design approach should also lead to reuse at higher levels of functional abstraction.

In what follows, we assert that the interconnection technology needed to join components together already exists, either in the form of direct procedure calls, or through the use of software busses and/or module interconnection languages [5, 15, 16].

The initial step in the design of large software systems is to break the problem into manageable sections. The architecture level of design, as noted by Shaw [17], may determine the major modules of the systems and their method of communication regardless of the implementation medium. Ideally, this should lead to a degree of design reuse since Shaw's architectures are composed of very high level functional abstractions. Each module is subsequently designed using a suitable methodology to allow implementation using a particular programming language. It is claimed software designers need to develop an engineering mindset in order to improve their development techniques [18].

An engineering mindset is simply the method of creating systems by utilising the component base of that engineering discipline. Given this definition, there is a sense in which software developers already have an engineering mindset. The difference is that other engineering fields maintain a progressive improvement, utilisation, and standardisation of that component base. The component base of traditional engineering disciplines has continued to expand through the evolution of components and the invention of new functionality [2].

This process of improvement includes the identification and creation of new functional elements. In some cases (cf electronics), this process is itself technology driven where improved implementation techniques allow increasingly large (standard) functional units to be 'componentised'. Moreover, the process of locating, understanding, and utilising those components is standard and has remained relatively unchanged component data books. Alternatively, the component base used to develop general purpose software systems has remained at the level of granularity of the programming language construct with no design method for utilising any component above this level. There have been well-defined problem domains in which the granularity of the software components has progressed, for instance Fortran math libraries and graphical user interface development kits[7] . However, standard software development components which are applicable across problem domains have not been widely adopted although the discipline of software engineering accepts the need for large scale modules along with the concepts of information hiding and encapsulation It is this progressive improvement of usable, standard, well-defined, and understood engineering components which is the difference between the software development community and other engineering disciplines. To utilise a graphical component based design approach which provides the benefits promised by reuse with the cognitive advantages of visual development, methods must be developed to utilise an evolving set of general purpose, well-defined, and understood components.

With the number of different programming languages currently available (functional, procedural, object-orientated, hybrids, etc) it is often hard to generalise about the development strategies used in the development of general purpose software systems. It is possible to argue that object orientated programming languages already offer some of the attributes of GCBD that we are submitting. Indeed the notion of GCBD was initiated in the object orientated community [19]. However, we believe that object orientated programming languages are not the panacea to the problems software development and that the principles of GCBD need to be applied to all software development regardless of whether some of those attributes are inherent in the development language or not.

The authors have identified a number of attributes of traditional engineering development practices which, we believe, must be incorporated into a graphical component based approach to large scale software development. Firstly, the developer must be able to manipulate graphical objects which represent software components with a larger granularity than that of programming language statements. Secondly, the tools used to manipulate those components must allow connections between different types of high level components. Thirdly, the development environment must allow the user to easily locate these high level components and provide the designer with an ability to determine the functionality of any component. This is analogous to engineering data books which provide the engineer with a standard method of locating components and also standard ways of modelling the functionality of that component so the user can easily determine if it is suitable for the required purpose.

[7] There are numerous other examples of component kits in specific application domains.

4. A Component Based Development Approach to Huron Systems

A prototype for a graphical component based development environment is currently being developed as part of the AAITP HyperCASE project [20]. The prototype utilises the HyperCASE tools to implement the graphical component based approach to developing systems using Amdahl Corporation's Huron Rule Language [21].

The Concept of Huron

The goal of Huron is to provide the developer with a comprehensive, integrated software approach to application development by facilitating system creation from start to finish within one environment.

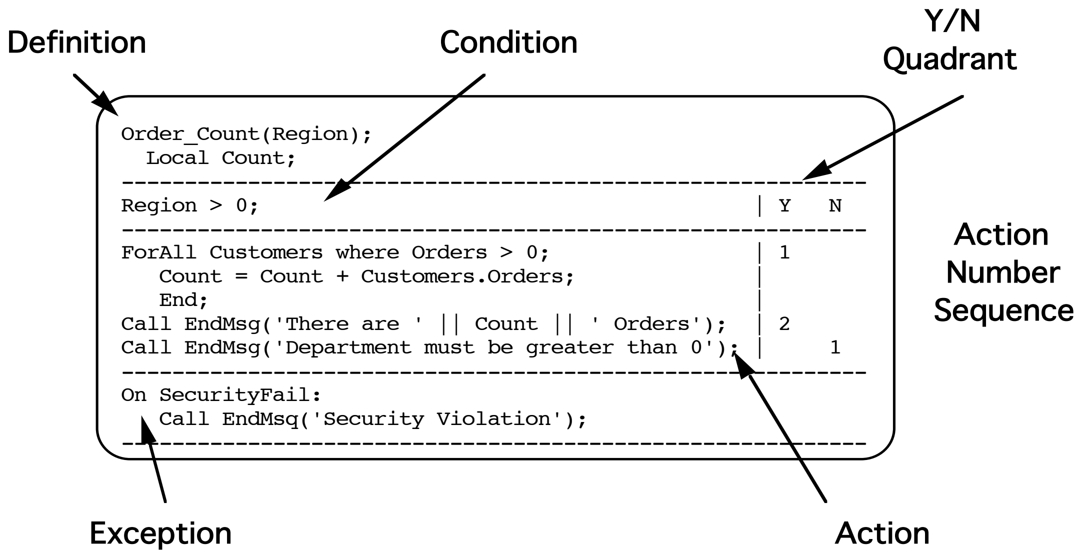
Huron provides an application prototyping method for transaction based processing systems by comprising the following features: Internal database and links to external databases, rules language, screen painter, report definer/generator, security management and administration, as well as customisable design interfaces.

Huron Rule Language

The Rule Language is the programming language used to build application within the Huron environment.

Rule Features

Rules are short, structures programming modules which facilitate condition processing and branching through a simple Yes/No decision tree.



```
Order_Count(Region);
  Local Count;
-----------------------------------------------------------
Region > 0;                                       | Y   N
-----------------------------------------------------------
ForAll Customers where Orders > 0;                | 1
   Count = Count + Customers.Orders;              |
   End;                                           |
Call EndMsg('There are ' || Count || ' Orders');  | 2
Call EndMsg('Department must be greater than 0'); |     1
-----------------------------------------------------------
On SecurityFail:
   Call EndMsq('Security Violation');
-----------------------------------------------------------
```

When a rule is invoked, it evaluates the condition and processes the action statements following the action number sequence of the appropriate column in the Y/N Quadrant. For example, if Region > 0 the rule will execute the action statements associated with the 'Y' column (the ForAll statement and the first Call statement). At any time an exception can act as an interrupt and terminate the execution of the rule, for instance if there is a database security problem.

As a result of their structure, Huron rules are small and function specific. This allows them to be used in a very modular manner which promotes the chances of component reuse.

Huron Data Access and Manipulation
The database (Table Data Store) is an entity-relationship DB which also provides a number of additional facilities: event rules to allow other actions to be triggered as a result of data access, validation of data to ensure the integrity of the DB, table parameters to allow sets of data to be grouped together.

The TDS is stored as B+ trees with support for hashed storage of data rows.

The MetaStor is used to hold all information about the environment. For instance, the MetaStor can hold information about other DBMS's which allows Huron to access external databases.

Huron Workbench
The Huron Workbench contains utilities to create rules, tables, screens, and reports. It also provides access to internal tables and remote databases. Reports can also be generated from the Workbench.

Huron Operating Environment
Huron currently operates in the System 390 environment under MVS as well as under UTS (Amdahl's mainframe implementation of Unix) and SCO Unix on PCs.

Huron: Concepts and Facilities. 1991, Amdahl Corporation:

Initially, high level components were identified. Huron is a very modular language with extremely structured rules. This property of the language made it possible to easily identify three basic high level components: rules, database tables, and user interface screens. These components could be considered to be analogous to digital logic gates in electronic engineering. An additional component was included into the system to represent Huron subsystems. These represent a combination of many of the primitive components. Again, an analogy can be made with the electronic engineering discipline with the Huron subsystem component being similar to an VLSI chip. The developer can think in terms of the functionality of the subsystem during the design regardless of the combination of rules, tables, and screens which actually comprise the component. Similarly an electronic engineer thinks in terms of the functionality of the VLSI chip regardless of the multitude of logic gates and other low-level components within the chip.

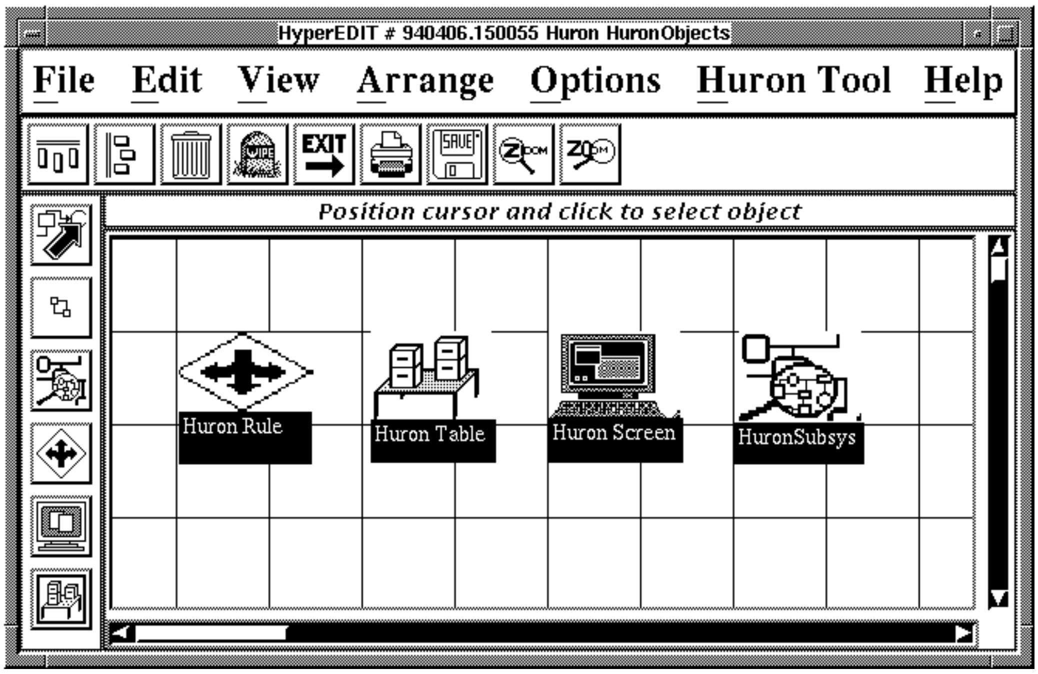

HyperEDIT # 940406.150055 Huron HuronObjects
File
Edit
View
Arrange
Options
Huron Tool
Help
Position cursor and click to select object
Huron Rule
Huron Table
Huron Screen
HuronSubsys

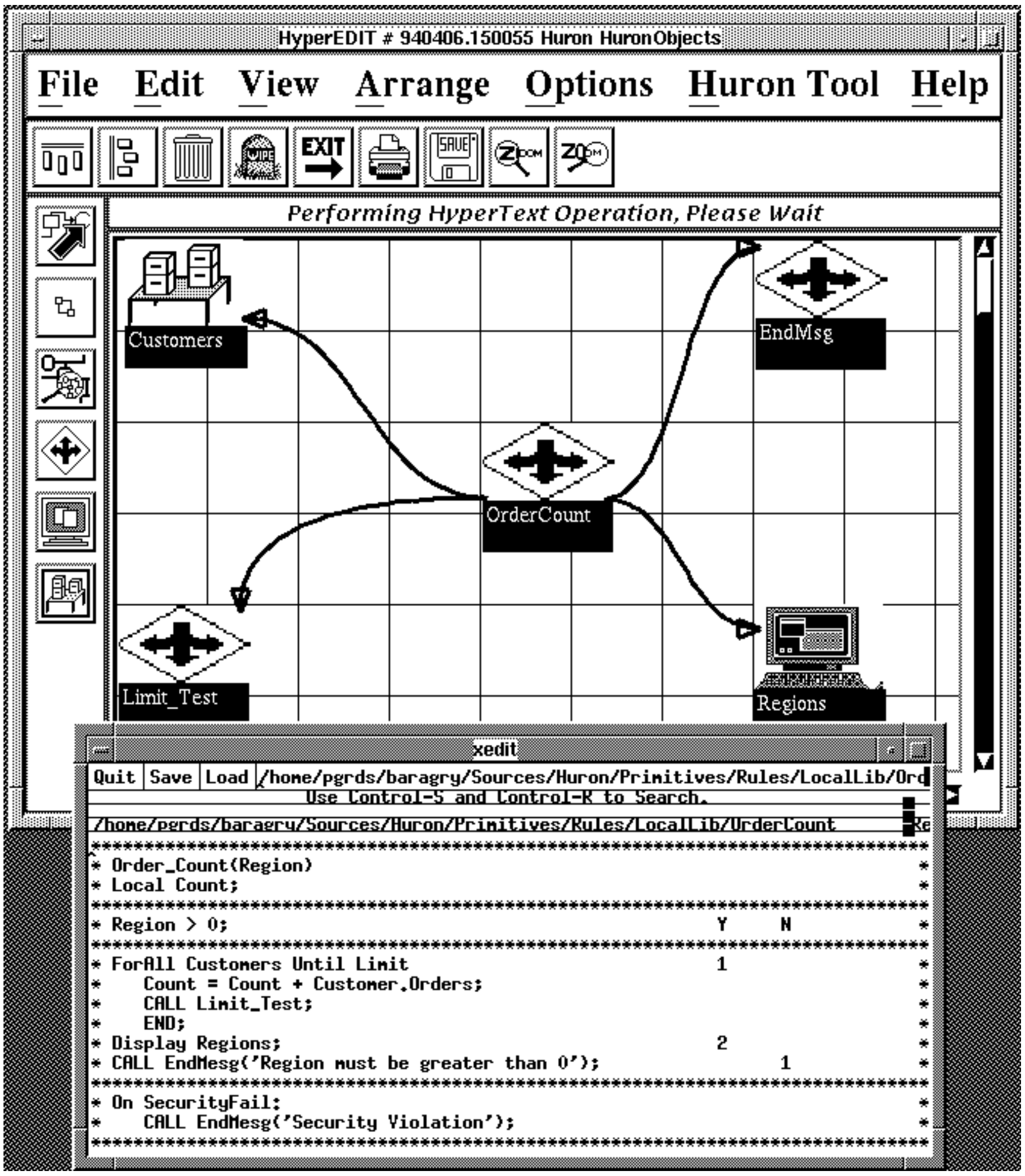
Figure 1: Huron Components

These diagrams would be used in the same manner as an electronic engineer uses schematic circuit diagrams. The diagram is comprised of a number of different components with different levels of granularity. For instance rules, table, and screens can be connected to each other or to Huron subsystems. The resulting Huron 'schematic' diagram can be utilised in the same manner as electronic schematic diagrams. The developer can fully understand how the system works by viewing the schematic because the functionality of the displayed components is understood. The developer is then given a good representation of the overall system function by abstracting away the detail of each components implementation. Software maintainers could also benefit from the system because the implementation of the system can be viewed at a higher level than that of the source code without resorting to reverse engineering tools. That cognitive distance between the design and implementation is reduced.

To allow the user to locate usable components, the prototype has a Tool Library which is categorised into the identified components: rules, tables, screens, and subsystems. Further categories can be made to categorise the components into problem domains and related functionality. In addition, when the developer chooses a Huron component to include in the current diagram, the components which it invokes as part of its implementation are also automatically added to the diagram. Software reuse is limited by the fact that developers only reuse software they are familiar with and there is an enormous amount of diverse software which exists over an ever increasing problem domain. In addition to the traditional component retrieval capabilities such as keyword searching, it is envisaged that the Tool Library will eventually use other aspects of HyperCASE to assist the designer by using the current design reasoning information and system requirements to suggest relevant, usable software components to the developer. This would assist the designers by informing them of useful functionality without attempting to automate the design process.

Designers in traditional engineering disciplines have utilised component data books and standard modelling techniques to understand how components work and determine if they meet the required functionality. Software does not have standard modelling techniques or general purpose data books[8] . To overcome this deficiency, ExDess utilises the HyperText facility of HyperCASE to provide links between the software components and the next level of their implementation. The developer can simply click on a Huron rule, for example, and see how it is implemented (figure 2). Figure 2 represents the rule's implementation in text format although a visual language has also been developed which could be utilised to depict the rule's implementation. Moreover, Huron subsystem components have links to diagrams which show how they are comprised of particular rules, tables, and screens. This facility provides the user with the same features that standard data books and modelling techniques have provided for designers in traditional engineering disciplines.

[8] There are data books for this purpose for selected software domains. eg: X windows manual for GUI development and module libraries for operating system calls. However, these are not available for general purpose software components.



```
/home/pgrds/baragry/Sources/Huron/Primitives/Rules/LocalLib/Ord
                Use Control-S and Control-R to Search.
/home/pgrds/baragry/Sources/Huron/Primitives/Rules/LocalLib/OrderCount
*****************************************************************
* Order_Count(Region)                                           *
* Local Count;                                                  *
*****************************************************************
* Region > 0;                                      Y     N      *
*****************************************************************
* ForAll Customers Until Limit                     1            *
*    Count = Count + Customer.Orders;                           *
*    CALL Limit_Test;                                           *
*    END;                                                       *
* Display Regions;                                 2            *
* CALL EndMesg('Region must be greater than 0');         1      *
*****************************************************************
* On SecurityFail:                                              *
*    CALL EndMesg('Security Violation');                        *
*****************************************************************
```

Figure 2: HyperText Links to display the next level of detail.

In Figure 2, the Huron schematic diagram represents the OrderCount rule and the other Huron components its interacts with. At present, the HyperEDIT tool [23] which is being used to implement the system only allows connections to be specified between components. Consequently, the only method of distinguishing between the different calls to the EndMsg routine is through the use of flow labels (these have been hidden in the diagram to reduce the size). However, extensions are currently being made to allow connections between specific communication ports in each component. This would allow connections between components to be more accurately specified by using the physical connection points of the object to infer knowledge about the type of connection. For example it would then be easier to show the difference in Call statements for a normal action and on an exception interrupt. Again, this would be a similar functionality to an electronic schematic diagram where the connections occur between certain pins on the components thereby providing the developer with an indication of the purpose of the connection.

In addition to locating and utilising components, the Huron prototype provides the developer with the ability to create components which can be used by other developers. The user can simply choose a number of components in an existing diagram and 'componentise' this into a Huron Subsystem component. For example, in Figure 3a the components in the bottom half of the diagram have been identified as a useful subsystem which should be 'componentised' into a reusable system. The user simply clicks on those components and chooses the 'componentise' function from the menu system. The tool then saves these components and their interconnections as a separate diagram and represents them with a single Huron Subsystem component in the original diagram (figure 3b). A HyperText link is automatically inserted between the Huron Subsystem component and the newly saved diagram which depicts its implementation.

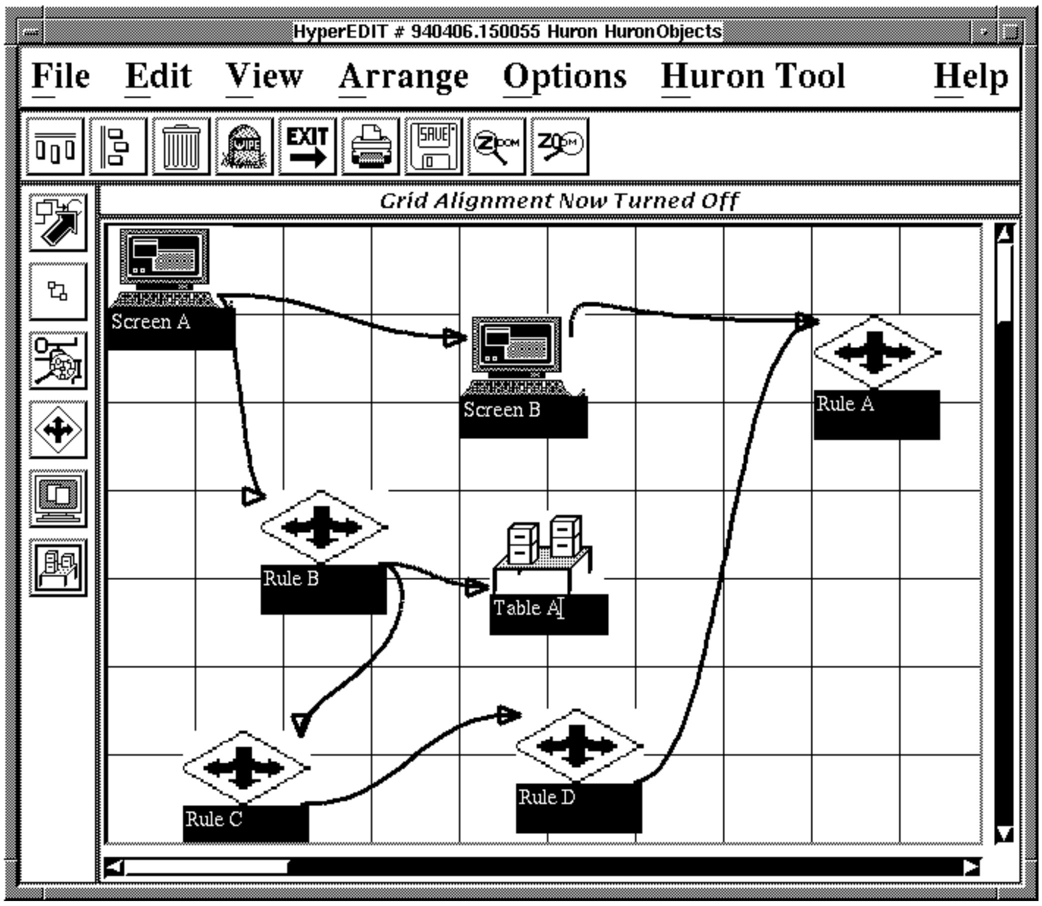


Figure 3a: Huron Subsystem Creation

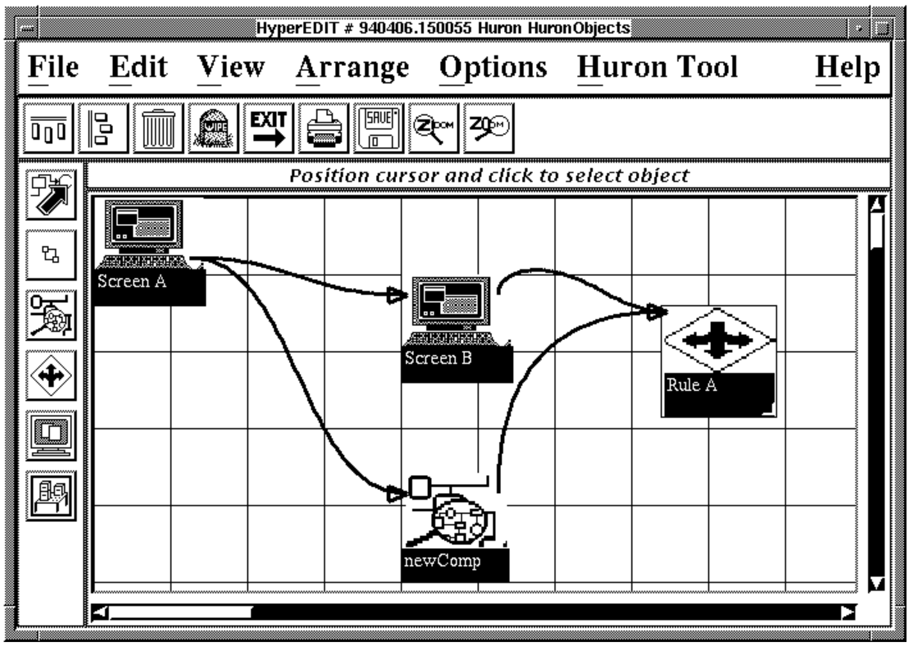

HyperEDIT # 940406.150055 Huron HuronObjects
File
Edit
View
Arrange
Options
Huron Tool
Help
EXIT
SAVE
Position cursor and click to select object
Screen A
Screen B
Rule A
newComp

Figure 3b: Huron Subsystem Creation

The first benefit of this tool is to allow developers to create components with a higher level of granularity which can be reused by others. The second benefit is the reduction in the size of the implementation diagram by utilising abstraction and information hiding to reduce the complexity of the diagram.

5. Conclusion

This work has been prompted by the realisation that diagramming systems play a major role in engineering design. The extent of this role varies from discipline to discipline in that the diagramming systems either support the design process by representing its results, or they are actually part of the design process itself. In all cases, as we have pointed out, engineering diagrams are executable in the sense that the system described can be constructed without further design. In some cases, (eg. civil, mechanical, construction engineering), the diagrams are actually representations of the physical systems, while in others (electrical, electronic) they show the interconnections between the components.

Our observations were that in general, software diagrams do not have this property. Our suggestion that the "good" diagramming system seems to be a derivative of component based re-use requires further elucidation. Like-wise our assertion that software diagramming systems inhibit re-use by their very nature. We concede that such a view is arguable. It is clear that data-flow diagram process bubbles may be pre-defined modules, as may actions in a state-transition diagram. Our point, ultimately, however, is that these approaches are not capable of representing the final system directly in the sense that we described.

In addition, the observation that there exists a GCBD "mind-set", (due to the first author), requires further investigation, as does the origin and raison de'etre of GCBD's themselves. Any such investigation will show that the concept of high-level diagrammatic representations of functional modules in electronics and control systems existed well before these modules existed as independent, reusable modules (see for example an early electronic design handbook). In this sense, our extension of visual languages (proposed by the first author), is a direct analogue of electronic design as it existed prior to the advent of integrated circuits. In fact, it could be said that we are making a technology driven step here, in much the same way, and integrating this with diagramming standards in much the same way. The graphic design tools which we are now able to develop preform the same integrating function that advances in semi-conductor technology did in the late 1950's.

6. References

1. Martin and McClure, *Diagramming Techniques for Analysts and Programmers*. .

2. French, M.J., *Invention and Evolution: Design in Nature and Engineering*. 1988, Cambridge University Press.

3. Rosenman, M.A., J.S. Gero, and M.L. Maher, *Knowledge-Based Design Research at the Key Centre of Design Computing*. 1993, Key Centre of Design Computing Department of Architectural and Design Science University of Sydney:

4. Reed, K., *Personal Communication with the Author*. 1992,

5. Cox, B.J., *Planning the Software Industrial Revolution*. IEEE Software, 1990. (November): p. 25-33.

6. Krueger, C.W., *Software Reuse*. ACM Computing Surveys, 1992. 24(2): p. 131-183.

7. Nierstrasz, O., *et al. Objects + Scripts = Applications*. in *Esprit 1991 Conference*. 1991. Kluwer Academic Publishers.

8. Shaw, M., *Abstraction Techniques in Modern Programming Languages*. IEEE Software, 1984. (Oct): p. 10-26.

9. Hawryszkiewycz, I.T., *Introduction to Systems Analysis and Design*. 2nd ed. 1991, Prentice Hall.

10. Dillon, L.K., *et al. Graphical Specifications for Concurrent Software Systems* in *International Conference on Software Engineering*. 1992. World Congress Centre, Melbourne Australia:

11. Glinert, E.P., ed. *Visual Programming Environments: Paradigms and Systems*. Vol. 1. 1990, IEEE Computer Society Press: Los Alamitos, California.

12. Glinert, E.P., ed. *Visual Programming Environments: Applications and Issues*. Vol. 2. 1990, IEEE Computer Society Press: Los Alamitos, California.

13. Cox, P.T. and T. Pietrzkowski, *Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independant Programming Mechanism*. Proceedings International Computer Science Conference, 1988.: p. 695704.

14. Myers, B.A., *Visual Programming, Programming by Example, and Program Visualization: A Taxonomy*. Conference Proceedings, CHI '86: Human factors in Computing Systems. ACM., 1986.: p. 59-66.

15. Purtilo, J., R.T. Snodgrass, and A.L. Wolf, *Software Business Organization: Reference Model and Comparison of Two Existing Systems*. 1991, DARPA Module Interconnection Formalism Working Group:

16. Beach, B.W. *Connecting Software Components with Declarative Glue* in *International Conference on Software Engineering*. 1992. World Congress Centre, Melbourne Australia: IEEE Computer Society Press.

17. Shaw, M., *Large Scale Systems Require Higher-Level Abstraction*. Proceedings of Fifth International Workshop on Software Specification and Design, IEEE Computer Society., 1989.: p. 143-146.

18. D'Ippolito, R.S. and K. Lee, *Putting the Engineering into Software Engineering*. 1992,

19. Nierstrasz, O., S. Gibbs, and D. Tsichritzis, *Component-Oriented Software Development*. Communications of the ACM, 1992. (September): p. 160-165.

20. Cybulski, J.L. and K. Reed, *A Hypertext Based Software Engineering Environment*. IEEE Software, 1992. (March): p. 62-68.

21. Amdahl, *Huron Reference Manuals*. Release 1.15, 1992, Amdahl Corporation.

22. Amdahl, *Huron: Concepts and Facilities*. 1991, Amdahl Corporation:

23. Proestakis, *A Diagram Editor Generation System*. 1991, La Trobe University: