A domain-specific language based approach to component composition, error-detection, and fault prediction

Ingham, James

A Domain-Specific Language Based Approach to Component Composition, Error-Detection, and Fault Prediction

PhD Thesis

James Ingham

Abstract

Current methods of software production are resource-intensive and often require a number of highly skilled professionals. To develop a well-designed and effectively implemented system requires a large investment of resources, often numbering into millions of pounds. The time required may also prove to be prohibitive. However, many parts of the new systems being currently developed already exist, either in the form of whole or parts of existing systems. It is therefore attractive to reuse existing code when developing new software, in order to reduce the time and resources required.

This thesis proposes the application of a domain-specific language (DSL) to automatic component composition, testing and fault-prediction. The DSL is inherently based on a domain-model which should aid users of the system in knowing how the system is structured and what responsibilities the system fulfils. The DSL structure proposed in this thesis uses a type system and grammar hence enabling the early detection of syntactically incorrect system usage. Each DSL construct's behaviour can also be defined in a testing DSL, described here as DSL-test. This can take the form of input and output parameters, which should suffice for specifying stateless components, or may necessitate the use of a special method call, described here as a White-Box Test (WBT), which allows the external observer to view the abstract state of a component.

Each DSL-construct can be mapped to its implementing components i.e. the component, or amalgamation of components, that implement(s) the behaviour as prescribed by the DSL-construct. User-requirements are described using the DSL and appropriate implementing components (if sufficient exist) are automatically located and integrated. That is to say, given a requirement described in terms of the DSL and sufficient components, the architecture (which was named Hydra) will be able to generate an executable which should behave as desired. The DSL-construct behaviour description language (DSL-test) is designed in such a way that it can be translated into a computer programming language, and so code can be inserted between the system automatically to verify that the implementing component is acting in a way consistent with the model of its expected behaviour.

Upon detection of an error, the system examines available data (i.e. where the error occurred, what sort of error was it, and what was the structure of the executable), to attempt to predict the location of the fault and, where possible, make remedial action.

A number of case studies have been investigated and it was found that, if applied to the appropriate problem domain, the approach proposed in this thesis shows promise in terms of full automation and integration of black-box or grey-box software. However, further work is required before it can be claimed that this approach should be used in real scale systems.

Acknowledgements

First, I would like to thank my supervisor, Professor Munro, for his help, advice, and tolerance to my “individual” style of working. Also thanks to the following for their varied and interesting discussions: Dr Steven Glover, Mark Jarvis, Dr Nigel Thomas , and Dr Uwe Krohn.

Thanks to my family for their help and support, especially Heather (especially for her help in proof reading), Gordon (for his advice and support), Stuart (you’ll have to read it now!), Rosie (for her voice of sanity, in an otherwise rather confused world), and Adrian (for being short). Also thanks to Robert and Anne for their encouragement. I got there in the end.

Cheers to the following for their friendship and in some cases advice: Andrew “you’re not wearing that?” Frith, Antony “classic” Hofton, Dr Pete “have you installed quake yet?” Young, Dr Claire Knight (for putting up with watching United/Everton games with me) and everyone else at the University of Durham.

Honourable mentions also go to: Steve, Anna, Tim, Siobhan, Claire, Lorna, Gaynor, Dr Andreas and Sara, Dr Nathan, Ben, Dr Loc, Sam, and everyone else I missed out...

Some thanks should also go to the staff at Ryburn Valley High School, Ripponden J&I and Stones school, for their often overlooked contributions. Special thanks to Mrs Rachowski (hope I spelt it right), who seemed to know I’d end up doing something like this long before I did.

Table of contents

1. Introduction

The aim of this chapter is to discuss the underlying driving forces of this research, namely that the development of software is resource intensive and the resulting software is often unreliable. One group of methods designed to combat these undesirable characteristics is the field of software reuse. High quality software reflects an investment in confidence that the code will act as desired, and in the future maintainability and usefulness of the code. Software reuse aims to reuse existing high-quality software when developing similar or new systems. However, software reuse is non-trivial. This chapter aims to provide an overview, at abstract level, of the underlying concepts in this thesis. It then outlines the thesis deliverables and criteria for success before describing the structure of the rest of this thesis.

1.1 Problem Statement

Current methods of software production are resource-intensive and often require a number of highly skilled professionals. To develop a well-designed and effectively implemented system requires a large investment of resources, often numbering into millions of pounds. The time required may also prove to be prohibitive. However, many parts of the new systems being currently developed already exist, either in the form of whole or parts of existing systems. It is therefore attractive to reuse existing code when developing new software, in order to reduce the time and resources required. If the required investment for developing software is considered in conjunction with the savings made when the software is reused, there is also a strong case for suggesting that software reuse is a method of improving software quality without necessitating increased investment.

Although it is widely acknowledged that software reuse is a desirable concept, most new software is still produced by redeveloping previously existing solutions.

There are a number of factors which may make software reuse difficult. These are discussed in depth in chapter 2.

1.2 Research Aims

It is the aim of this research to focus on one way of addressing some or all of the re-use problems. It is acknowledged that a number of other methods for addressing reuse have been applied. They are discussed in Chapters 2 and 3. However, when undertaking this research, a conscious effort was made to not be constrained by current approaches. An important question that should be asked at this stage is “do we really need a new method?”. It is the contention of this thesis that current methods are not successful in achieving reuse, a point which can be illustrated by the fact that most software is still re-developed from first principles. It should, however, be noted that applying the approach outlined to some or all of a problem domain will not necessarily exclude the application of other techniques and technologies which are also found to alleviate problems in reuse. In fact it is the author’s intention that, where appropriate, other reuse focussed approaches should be applied.

1.2.1 Hypothesis overview

This thesis proposes the application of a domain-specific language (DSL) to automatic component composition, testing and fault-prediction. The DSL is inherently based on a domain-model which should aid users of the system in knowing how the system is structured and what responsibilities the system fulfils. The DSL structure proposed in this thesis uses a type system and grammar hence enabling the early detection of syntactically incorrect system usage. Each DSL construct’s behaviour can also be defined in a testing DSL, described here as DSL-test. This can take the form of input and output parameters, which should suffice for specifying stateless components, or may necessitate the use of a special method call, described here as a White-Box Test (WBT), which allows the external observer to view the abstract state[1] of a component.

[1] A term used to describe a view of the data which must be stored, in some form, by all implementing components. It is abstract state because the result from the WBT for all components implementing the same persistent (i.e. requires state) DSL-construct should be the same, independent of actual representation.

Each DSL-construct can be mapped to its implementing components i.e. the component, or amalgamation of components, that implements the behaviour as prescribed by the DSL-construct. User-requirements are described using the DSL and appropriate implementing components (if sufficient exist) are automatically located and integrated. That is to say, given a requirement described in terms of the DSL and sufficient components, the architecture (which was named Hydra) will be able to generate an executable which should behave as desired. The DSL-construct behaviour description language (DSL-test) is designed in such a way that it can be translated into a computer programming language, and so code can be inserted by the system automatically to verify that the implementing component is acting in a way consistent with the model of its expected behaviour.

It is also important that, although the system of code generation from a DSL description is automated, it should work in a traceable manner. In particular, the mechanisms underlying the system must be dependable.

A final aim of the research is to, upon detection of an error, make the system examine available data (i.e. where the error occurred, what sort of error was it, and what was the structure of the executable), to attempt to predict the location of the fault and, where possible, make remedial action.

To summarise, the aims of this thesis are to:

- Use a DSL as a component specification mechanism. This mechanism will be used to define behaviour in terms of semantic naming and behavioural pre-post conditions.
- Translate the requirements, which are in terms of the DSL, into an executable program, assuming sufficient components exist.
- Upon detection of an error in the resulting system, ensure the supporting architecture will use all available data to predict the location of the fault. The supporting architecture (Hydra) should then attempt to generate an alternative solution.

1.3 Thesis Deliverables

The overall deliverables for this project are as follows:

1. The definition of an architecture that will enable the automated composition of components, and to also include self-testing and structural information in any executables developed. This is shown in chapter 4.
2. To provide guidelines on how to apply the concepts and techniques as proposed in this thesis. These are described in chapter 4 and amended in chapter 7.
3. A prototype, or prototypes, which illustrate the ideas and enable evaluation of concepts. These are described in chapter 6.

It was deemed important that the theoretical architecture should be implemented and tested in order to evaluate the success of the proposed system.

1.4 Criteria for success

The criteria for success in the context of this research are the following factors, which should be evaluated as they pertain to the hypothesis:

1. The investment required to develop the proposed system in comparison to a conventional system.
2. The return on the investment for the proposed system.
3. The types of error which can be detected by the proposed system.
4. The types of fault which the system can handle.
5. The 'evolvability' of the proposed system.
6. The overall feasibility of using this type of system.

It is acknowledged that these factors are not easily quantified. They were selected because it was hoped that they would give an accurate view of whether the approach could be successful, in preference to easily quantifiable, but not necessarily relevant, factors or criteria.

1.5 Thesis structure

The following chapters of this thesis are structured as follows:

Chapter	Topics covered
2	**Reuse** – Software reuse in general, including reasons for reusing, barriers to software reuse and some common techniques for software reuse.
3	**Other related work** – covering techniques which are, although not usually considered part of the software reuse discipline, have relevant areas of commonality.
4	**Concept** – A description of the hypotheses which form the backbone of this research, along with detailed description of the approach and architecture.
5	**Implementation** – A discussion of some of the relevant implementation issues and technologies which were applied in the prototype.
6	**Case studies** – A walk-through of the prototype and then outlining issues which are discussed in later chapters.
7	**Evaluation** – The concept and implemented prototypes are evaluated in terms of the success criteria and in terms of other criteria, the importance of which became evident during the research.
8	**Conclusion** – The research is overviewed, and suggestions for further work are made.
9	**References**.
10	**Appendix A**.

2. Reuse

The investment required to develop certain types of software can be financially significant. It is not uncommon for software projects to cost millions of pounds. However there are many existing pieces of software that could perform, all or part of, the required tasks. It is an attractive proposition to leverage the existing software when developing new solutions. This a major concern for the field of **software reuse**.

It is a mistake to believe that reusable software is easy to achieve. In fact there have been many different attempts to achieve software reuse, which have met with limited success. This chapter aims to highlight types of software reuse, why exactly reuse is important, why software reuse is so hard to achieve and finally some current approaches to software reuse.

At this stage the concept of a **software artefact** is introduced. This is used to describe the notion of software which is not necessarily structured in some way and to prevent ambiguity by using a term which means a number of different concepts to different disciplines. The informal definition of a **software artefact** is:

A collection of code which may be reused.

The main purpose of this chapter is to provide a general overview of software reuse issues and some common methods of addressing these issues.

2.1 Types of re-use

There are a number of different important factors in reuse. These include:

- Encapsulation
- Code domain
- Designed for reuse versus designed with reuse.

2.1.1 Encapsulation

For the purposes of this thesis, encapsulation is used to refer to the accessibility of a software artefact's implementation. The main ways to encapsulate code are black-box, white-box, and grey-box.

The definition of black-box reuse that has been used during this research is:

"A style of reuse based on ... [software artefact] *composition. Composed* ...[software artefacts] *reveal no internal details to each other and are thus analogous to "black boxes""* Gamma et al. [1]

A black-box reusable software artefact should have three parts. First, the object code, which is typically not accompanied by the source code in cases where the company which developed and the company which reuse the software artefact are different. The second part is the interface specification which is a means to access the functionality of the software artefact. The final, often neglected, part is the documentation which should describe how the software artefact is to be reused.

There are two main assumptions to black-box reuse.

1. The reuser of the code is able to gain the information required in order to reuse the code without having access to the implementation.
2. The code provides the correct functionality, within required non-functional characteristics, to perform some or all of the required task.

Pre/post condition semantics have been applied in order to aid in the documentation of black-box software artefacts. However it has been argued, particularly by Buichi and Weck [2], that this is insufficient for describing certain types of problems. Szyperski [3] found that although pre/post condition specification was very important for black-box reuse, it was almost totally unused in industry.

In addition to specifying pre/post conditions, the context of a software artefact may also be documented. This may consist of environmental requirements, such as which operating system the reusable software will function correctly on, or other such requirements.

The importance of knowing, and in some cases controlling, certain design and implementation decisions for the reusable software has been described by Kiczales [4].

Black-box reuse suffers from a number of problems:

- The software artefacts often have design decisions made too early in development. This means that otherwise functionally correct reusable software fails to perform satisfactorily, requiring a duplicate to be developed. The problem of multiple versions being required due to differing non-functional characteristics are described by Kiczales [4] as "haematomas of duplication".
- It is very rare that the design decisions are documented. This means that the reuser may not have the relevant data at time of reusing the software.
- Black-box reuse works only if the environment in which it is situated conforms to what it requires. Because the code cannot be altered (without extensive reverse engineering) even very simple changes such as exchanging "dir" with "ls" are impossible. To avoid this category of problem, commonly changing factors are parameterised. Unfortunately this relies on the developer of the software artefact predicting all the parts of the program which need to change.
- It is extremely difficult to ascertain the actual behaviour of a black-box software artefact without having to devise and study interface tests. Therefore if the behavioural specification is insufficient for the reuser to know whether the software artefact is appropriate then considerable effort must be expended in acquiring sufficient information.

The definition of white-box reuse that has been used during this research is:

> Reuse where access to the source code and object code is not restricted. That is to say that the source code is available and the software artefact may be addressed directly rather than through an interface.

White-box reuse relies, to a certain extent, on the "re-user's" ability to comprehend the software's structure when attempting to re-use the code. This places a strain on the re-user and requires that they are fluent in the language in which the original software artefact was implemented. Often this also means that the new software artefact will have to be implemented in the same language as the original.

It has been shown that white-box reuse relying on the actual implementation of the code rather than interfaces (as in Black-Box) can produce systems which are more fragile to changes. Szyperski, in particular, discusses this in [3].

White-box reuse has the additional problem that due to the lack of hiding of internal workings, there is an implicit risk to the intellectual property. This is of great concern where the software may reflect a substantial investment. There is a strong case that black-box reuse, although flawed in the ways described earlier in this chapter, is the only realistic option for reuse of software artefacts across separate companies.

A middle ground between the two polarities of black-box and white-box reuse has been described by some authors, e.g. Buichi and Weck [2], as grey-box software reuse. They define it as:

> *"A grey box reveals parts of its internal workings, not just relations between input and output. The information can become as detailed as necessary where needed, for instance, to state under what conditions external ...*[software artefacts] *are called. In other places it may remain very abstract and simply state a condition that is established"*

This breaking, or at least reducing, of abstraction can be used to aid in improving performance, Kiczales [4], or pre/post condition specification testing, Principle 5 of Hollingsworth [5].

2.1.2 Code Domain

Another important classification of code reuse is by how the code will be reused, in particular whether the code will belong to an application family or will be used generically. There are two categories:

Vertical re-use: the reuse of software artefacts which are used in a series of systems that all belong to the same application family.

Vertical reuse has a focused domain. Vertically reusable software artefacts are often quite specialised in functionality or in optimised performance. It has been suggested by the Honeywell institute [6] that vertical reuse has a much greater potential saving than horizontal reuse. A substantial quantity of vertical reuse research has been in the field of Avionics, Poulin [7], overviewed in Taylor et al [8], Batory et al. [9], and an overview in Batory et al [10].

Horizontal reuse: the reuse of code which spans several different types of applications.

Horizontal reuse is the reuse of code whose functionality crosses many application domains. Examples of horizontal reuse include user interfaces and databases. Since horizontal reuse is not designed for a particular application, horizontally reusable components usually cannot be tailored to particular circumstances and as such may have unfavourable non-functional characteristics. The need for horizontally reusable software artefacts is often reduced because modern languages are designed with constructs or libraries which perform these tasks (for example S.T.L. in C++ Stroustrup [11])

2.1.3 Designed for reuse versus designed with reuse.

Another important consideration is the method by which the software artefact was created. Software artefacts which are designed for reuse (DfR), as per Becker [12], have been designed specifically to be reused. They are typically "heavily parameterised", as per Becker [12], that is to say they usually have as many options as possible at the function interface level. The context requirements for the software artefacts are usually quite undemanding in order to maximise the reuse potential, due to the specific context being unknown at development time.

Software artefacts which are designed with reuse (DwR Becker [12]) have been developed for a previous application and then generalised to allow them to become reusable. The context is usually quite focused. One technique used to recover reusable software is to reverse engineer part of the software resulting in reusable objects as described in Burd and Munro [13].

2.2 Why re-use?

There are many possible advantages to successful re-use. Sommerville [14] states the following reasons:

- Increased system reliability
- Reduced overall risk
- Effective use of specialists
- Organisational standards can be used.
- Reduced development time

2.2.1 Increased system reliability

It is more cost-effective to produce high quality software if it is to be reused as stated by Tracz [15]. The cost of undergoing the process of specifying, designing, implementing and testing can be prohibitive if the artefact is not necessarily going to be used again. If the software has been reused a number of times already, limitations and dependencies may have been discovered which otherwise may have been unknown. Therefore the need for a high quality software product is an incentive to reuse.

Whether software reuse actually improves system reliability is still a matter of debate, since although software quality can increase, incompatibilities and unforeseen problems may arise. Some of these reuse issues are discussed further in Section 2.3.

2.2.2 Reduced overall risk

For many companies the risk of investing in software development may not pay-off. Therefore it is attractive to share the investment risk with other companies. This can be achieved by reusing software artefacts developed by an external company who will have invested in the software artefact.

In addition to allowing other companies to take risks on behalf of code developers, investment in software development may also be reduced by leveraging existing internally produced software. However this is usually a long-term goal. Leveraging existing software may be achieved by developing reusable software, by restructuring as in Brooke et al. [16] or by recovering reusable objects as in Burd and Munro [13].

By reusing existing code which may have been developed internally or externally to the company, a new software artefact may be developed using significantly less resources[2].

Figure 2:1 shows savings achieved in a vertical reuse project, according to the Honeywell institute [6]:

Stage in the software lifecycle	**Minimum reduction observed**	**Maximum reduction observed**
Requirements	50%	80%
Design	90%	95%
Implementation	95%	95%
Testing	90%	95%
Documentation	50%	80%

Figure 2:1

2.2.3 Effective use of specialists

The reusable software artefact may belong to a domain which the software developers have no experience in. It is beneficial for this highly specialised part of

[2] The word "resources" at this point is used to refer to concepts such as cost, time and personnel, not computer system resources.

the system to be developed by experts in that domain, assuming that the rest of the system can abstract the complexity of that part of the system.

An illustrative example of successful abstraction of expert domains is security. According to Robben et al. [17] encryption/decryption code can be re-used with only a very basic understanding of how the encryption works.

2.2.4 Organisational standards can be used

The use of standards across an organisation can provide a uniform way of performing certain tasks rather than re-solving the same problems, where each software artefact would require maintaining independently. Another useful attribute of this mechanism is that intra-company code should be structured in a more familiar manner and this has been shown to ease the comprehension task for new code developed within that organisation.

2.2.5 Reduced development time

Rapid Application Development (RAD) is very attractive to business, especially those that rely on software to support the services they provide. Shortening the time between problem identification and solution development will constitute a significant competitive advantage. It is the opinion of the author that reducing the development time of new functionality will become the primary concern for companies reliant on IT in the near future.

2.3 What prevents Software reuse

It is well accepted that software reuse is a desirable undertaking for a company which relies on I.T. However it has been found that reuse of assets is in many cases not performed. According to Tracz [15], and Judicibus [18] there are technical and sociological barriers to software reuse. However, reuse of modern code is non-trivial and it should be noted that taking an overly-negative view of programming practice may be counter-productive.

2.3.1 Quality

There have been a number of attempts to quantify the quality of a piece of software, commonly known as **software metrics**. Although software metrics do exist that address reusability and software quality as in Fenton [19], they are by no means definitive. A good software metric rating is an indicator that the software fulfils the desired criteria. However, there are many factors which the metrics do not take into account. Poulin [20] takes a contrary stance, stating that:

"We have solved the Metrics Problem...This work has succeeded in that (Poulin [21])

- *We have defined what to count as reuse and how to count it.*
- *We have defined metrics for reuse and can recommend these metrics for immediate use on projects within organizations.*
- *We know how to assess the financial benefits of reuse both in short and long term."*

However, it is the author's opinion that this is overly optimistic. Although metrics are helpful in measuring software reusability, as documented in Poulin [22], they are a long way from solving the problem of evaluating how "good" software really is. To re-iterate, currently existing metrics can not be used as the sole basis to make reliable informed decisions on how and when to reuse.

Uncertainty over software quality is both a sociological and technical barrier to software reuse. It is also cited as one of the main contributing factors to the "not invented here syndrome"[3].

Possible software quality measures include software process measures, SEI [23] , or run-time measures (such mean-time to failure as described in Sommerville [14]).

[3] The phrase used to describe the reluctance of a software engineers to reuse code which they did not develop themselves.

2.3.2 Behavioural Specification

It has been documented by a number of authors, including Syzperski [3], that it is extremely difficult to document the actual function of software. Formal methods such as Z (e.g. Lightfoot [24] or Bowen [25]) have been applied to this problem, as have algebraic specifications (as in Bergstra et al. [26] or Horebreek [27]). These have been found hard to read or even incomprehensible by many programmers. The specification of interfaces and semantic naming schemes is prevalent in most imperative programming languages. However, this relies on the re-user and original developer having a common terminology, which is not necessarily the case.

A number of authors, including Kiczales [4], suggest that specification using Pre/Post condition semantics (as in Black box reuse) is only effective at describing certain aspects of reusable software.

2.3.3 Design assumptions

When the developer(s) of the re-usable code makes design or implementation decisions, these decisions may restrict the conditions that the code can be re-used. Often it is very difficult to be able to predict exactly how one decision can impinge on future decisions so it is possible that the developer will not be fully aware of the consequences. That is to say each decision may affect many other areas simultaneously, and there may be many such decisions made implicitly throughout the development phase, for example due to choice of implementation technology. Figure 2:2 lists a number of examples of design assumptions.

Examples of a design or implementation decisions causing unforeseen clashes

- A deadlock scenario, where two software artefacts require each other's resources before they will release their resources.
- A software artefact being used concurrently with itself, when it was not designed for such a task.
- One piece of code relying on the state of an external system to remain unchanged even though this is not necessarily true.
- A system having a fixed memory size which the software artefact does not always obey.

Figure 2:2

It is very unlikely that the developer can exhaustively document these in-built assumptions. Therefore even when a re-user has a reasonable amount of confidence in the actual function of the software artefact, the underlying implementation may act unexpectedly due to hidden dependencies or other assumptions. The problem of such decisions being made too early, i.e. at design time, is described from the perspective of performance in Kiczales [4].

2.3.4 Adaptation required for reuse

There is rarely a direct mapping between the code which can be reused and the problem which is needed to be solved, as described in Samentinger [28].

Therefore certain types of software reuse will rely heavily on the innovations of the reusers, rather than the plug and play system usually proposed for black-box reuse. There may be a narrow line between appropriate and inappropriate software artefacts.

A number of different code adaptation and combination techniques exist, most of which are described in Bosch [29] and Szyperski [3]. However further discussion is outside the scope of this research.

An important point, raised in Neighbors [30], is the difficulty involved in joining different systems which encompass differing abstractions. This often results in unmaintainable code.

2.3.5 Location of relevant code

The decision to reuse code should be based on the motivation of economy. That is to say it should be less effort to reuse code than to redevelop it.

Independent of which requirements and design processes are undertaken, the location of existing resources is one of the most important tasks in the development of a system which is based on reused code. If the effort taken to find and reuse the code is greater than that of redeveloping the solution then the decision to reuse is uneconomical. That is to say if a software engineer has to waste time and effort looking for code to reuse then s/he will be less willing to attempt to reuse code next time due to frustration alone.

Another important, yet often implicit, consideration is how to shorten the time in which a software artefact can be found to be appropriate or inappropriate for reuse after the location of possible candidates. The amount of effort required to perform this task can be made prohibitive if the documentation of the software artefact is of insufficient detail.

2.4 Software Components.

A software component is a piece of reusable code. There are many descriptions of what exactly a component is, often from differing perspectives. Many of these are summarised in Szyperski [3]. In the same book, Szyperski defines a component to be:

> *"...a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

It should be noted that although this definition is useful, it is unclear whether any code can realistically have only explicit context dependencies (See 2.3.3 - "Design assumptions" earlier in this chapter). Current difficulties with documenting components are described in Brown and Wallnau [31]. However, the notion of components being units of composition is helpful in that it emphasises the reusability aspect.

Tracz [32] defines components in terms of their traits:

> *"...A component is defined in terms of the following attributes:*
>
> - *name,*
> - *responsibility or capability*
> - *constraints*
> - *dependencies*
> - *interface"*

Component naming schemes should be relatively free to allow for components which were designed for a different system to be integrated. However a sensible naming scheme would be preferable as is good practice in programming.

Capabilities can be described in a number of ways. The attempt to describe "What do I do" has also been undertaken by, in research into software agents, using architectures such as Beliefs, Desires and Intentions. However, the usual method of documentation is to describe the component with respect to some terminology that is assumed to be universal across all the systems for which the component is used.

Constraints are the conditions in which the component can be expected to perform satisfactorily, other than dependencies on other components. Common constraints include the underlying architecture issues such as the OS the component is expected to execute on, methods of communication and the sort of structures which the component will rely on. Further discussion of architectural issues is in section 2.7.

Component dependencies, or to be more accurate component inter-dependencies, should document the reliance of components in terms of other external components.

The component interface is a list of the services that the component offers. The interface conventionally describes the type and name of any externally accessible variables or methods. Some form of documentation to aid the re-user to understand what the component is, and in some cases how it works, should also accompany them. In some research component interfaces also include pre/post conditions, although in other work pre/post conditions may be described in a place which approximates to the constraints field.

It should be emphasised that not all software artefacts which may be reused are components. Components are a sub-category of software artefacts which fulfil the criteria described in this section.

Components can be organised in a number of methods, including libraries (section 2.5) and frameworks (section 2.6). According to Traas [33], a survey of component reuse, most component collections are currently offered by third party resellers.

However, component behaviour specification is currently not standardised (i.e. not universal) and this is considered to be a substantial barrier to component reuse. However third party resellers are expected to classify by a standard software library indexing method or framework.

In Baggiolini and Harms [34], the process of automatic fault-management is described as it relates to component-based applications. Of particular relevance is the identification of the problems of fault-propagation and distributed failure. Henderson [35] describes architectures which can be used in dynamic systems (system where new components can be added at run-time without restarting) using a architecture modelling language called ARC.

Components attempt to document the relevant data at the component level, hence reducing the amount of time required to decide whether the component is appropriate for the problem at hand. However, it may be difficult to know certain aspects of the component (for example the exact component constraints for a semi-generic, cross-platform component with a few hard-coded requirements.)

2.5 Software libraries

Software libraries are a well founded method for achieving reuse. They aim to collect commonly used fragments of code in an easy to find manner. A potential reuser looks for code that will fulfil their needs by navigating through the library.

The main difficulties for software libraries in general are

- It is difficult to document all of the relevant data that a user may need, especially when this information may not be known about the software artefacts.
- The user must know what they are looking for, in terms of the methods by which the library is indexed.
- A significant investment is required to produce a software library, independent of whether developing new software artefacts or recovering existing software artefacts.

According to Ostertag [36] the main methods of indexing software libraries are:

- The Keyword library system, by which the user locates data by searching for certain words which the user expects to be present in the desired software artefacts. The library may have no solid structure.
- The Classification library system, examples of which are classification bygeneral type (as used in traditional libraries) and facetted reuse libraries.

2.5.1 Keyword libraries

Keyword libraries are informal and unstructured. There is no guarantee that necessary information will be present and the library itself provides no guidance to the type of information required. This means there is poor support for finding software artefacts that require adaptation before reuse. A user would have to guess what sorts of software artefacts existed and how they were referred to in the software library. The indexing mechanism is reliant on the user knowing the terminology used by the documentation of the software artefact. This is not always a sensible assumption.

2.5.2 Classification libraries

There are a number of different methods of indexing in a classification library. The two main variants are:

- Classification by general type
- Facetted classification.

Classification by general type is based on a universal description hierarchy. A familiar example of a classification hierarchy is shown in Figure 2:3:

```
Animal
      ➔ Mammal
        ➔      Dog
        ➔      Cat
      ➔ Reptile
        ➔      Lizard
```

Figure 2:3

This type of classification is easy to understand but is inherently fragile to changes in categorisation. This makes this form of categorisation only suitable for very well known domains where the underlying categorisation is highly stable.

Facetted classification, as in Prieto-Diaz [37], is a method of categorising software by criteria which are considered important for this particular software library. This method is less vulnerable to changes because new facets can be added where needed, although updating existing software artefacts can be problematic. It has been found, as by Mili [38] that facetted systems are often hard to use and the effort of classification is not always justified. There is considerable disagreement, for example Poulin [20], to whether the invested effort of classification results in a sufficient rise in productivity to justify the classification methods.

A classification software library can provide more guidance to what information should be recorded about a software artefact. However, it is still unusual for a classification software library to contain all the necessary data that would be required for a user to know if a software artefact is appropriate. Classification systems also require the user to learn how the library is structured before use.

2.6 Frameworks

An alternative method of organising reusable software artefacts is via frameworks:

> *" A framework helps developers provide solutions for problem domains and better maintain those solutions. It provides a well-designed and thought out infrastructure so that when new pieces are created, they can be substituted with minimal impact on other pieces in the framework. "* Nelson [39]

According to Jacobson et al.[40] and Pree [41] frameworks are not simply collections of software artefacts. A framework element has a place in the system. If it is specialised (i.e. it is altered by some mechanism) the behaviour of that particular context will be altered.

The most common usage of the term object-oriented framework is to describe a framework designed for white-box reuse of classes where the form of specialisation used is totally, or at least predominantly, inheritance (described in Mauth [42] as whitebox frameworks) . A more exhaustive definition is provided by Taligent software [43] who classify frameworks on a number of categorisations as shown in Figure 2:4:

Categorisation	**Examples**
Domain	Horizontal, Vertical, System level (as in an OS API)
Architecture	Top-down, peer-peer, or many other variants.
Specialisation	Architecture driven – mostly via inheritance Data-Driven – mostly via composition.

Figure 2:4

The specialisation mechanism of a framework reflects a number of other framework design decisions. Architecture driven specialisation, that is specialisation based on inheritance, indicates that the framework will be mostly focussing on white-box reuse. Data-driven specialisation, that is specialisation based on composition, indicates that the framework may be able to support both black-box and white-box reuse but usually is more focussed on black-box reuse.

Taligent classifies the two different specialisation mechanisms in terms of their usability and extendibility:

	Architecture-driven	Data-driven
Usability	Often very hard to initially comprehend the framework.	Found to be initially much easier than architecture-driven.
Extendibility	Very flexible, as long as architectural issues are met.	Often quite limited as the framework is encoded into the component.

Figure 2:5

They also suggest that a successful framework should initially combine the two specialisation mechanisms.

However, a contradictory stance over the flexibility of black-box and white-box frameworks is taken in Fayad and Schmidt [44]

> *"Many framework experts (Johnson and Foote [45]) favor black-box frameworks over white-box since black-box frameworks emphasize dynamic object relationships (via patterns like Bridge and Strategy in Gamma at al. [1]) rather than static class relationships. Thus, it is easier to extend and reconfigure black-box frameworks dynamically. "*

There are also instances of component frameworks, such described in Szyperski [3], where the framework is focussed at the component abstraction level rather than at the level of objects/classes:

> *"A component framework is a software entity which supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances." Szyperski* [3]

The most common form of reuse in component frameworks is black-box although grey-box has been applied too, as by Buichi and Weck [2]. White-box, as described earlier in this chapter, is not normally used for components because it has a tendency to create unnecessary and implicit program inter-dependencies that black and grey-box reuse control more strictly.

Frameworks are useful for behavioural specification, as each place in the framework has a documented purpose. The documentation is extremely important as stated in Traas [33]. Software frameworks are based on the assumption that the individual framework elements cannot interact in unforeseen ways. Although the assumption holds in well designed frameworks, unpredictable behaviour can still occur. As frameworks are often organised in complex ways, for example with non-hierarchical models, faults can be difficult to locate. Frameworks support specialisation, although the exact method of specialisation is dependent upon the type of framework used. The additional advantage is that frameworks have a notion of where certain functionality should be placed, in structural terms.

It has been stated, e.g. Poulin [20], that further research is necessitated on how to integrate frameworks with other frameworks or different systems.

2.7 Software Architectures

The field of software architectures (Shaw and Garlan [46], [47], Hofman et al.[48]) can be defined to involve:

> *"...the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns..."*

Software architectures are useful because they encourage well-documented high-level system designs. One implication is that these designs can be reused (Clements and Northrop [49]).

The documentation of system-wide architectural decisions is important in terms of consistent maintenance and of development of new systems. More relevantly, it is only feasible to reuse code when it is based on an architecture which is, or can be made to be, compatible with the current system. Difficulties with differing architectural constraints has been documented for Windows™ components by Richardson [50].

Software components are developed upon the assumption that these architectural constraints will hold. Examples of these architectural decisions are event and resource handling. The system architecture may require a component which is interested in a certain event to register with the event handler or alternatively it may require the component to launch its own event monitor. The policy upon which a system resource may be re-allocated could vary between architectures, for example by reference counting or only on explicit release of the resource.

If these policies are not known at development time, then this reduces the certainty that the component can be successfully reused. Such a reduced certainty could also be viewed as a contributing factor to the "not invented here" syndrome.

2.7.1 Domain-specific software architectures

A number of domain-specific software architectures (DSSAs) have been developed according to Mettala and Graham [51], for example in the field of avionics as in Coglianese et al [52], Lockheed [53], and Lockheed [54] and in the field of on ship command and control Clements [55].

According to Tracz [32] a DSSA consists of:

- "...*a software architecture with reference requirements and domain model*
- ...[an] *infrastructure to support it*
- ...[a] *process to instantiate/refine it.*"

The improvement of a DSSA (Honeywell [6]) over a Software Architecture is that the system is situated, that is to say certain design and implementation decisions have been explicitly made. Certain aspects can be specialised to improve suitability for the system, for example in terms of focussing the system on the class of problems usually found in that domain, or in terms of optimising the system to perform in a manner acceptable to the application of the system. This removes many of the main barriers to re-using black-boxes which are described in Section 2.1.1 and in detail in Kiczales [4].

Czarnecki [56] is very definite about the importance of DSSAs in reuse:

> *"We believe that domain-specific software architectures are currently the only feasible way to achieve ``plug-and-play" component reuse. A domain-specific software architecture provides a common framework for component interoperability within a domain."*

2.8 Summary

This chapter has described issues which the author considers particularly salient to modern software reuse.

The different categories of reuse were found to be:

- Types of code reuse (e.g. Black box, White box)
- The domain of code to be reused (e.g. Horizontal, Vertical domain)
- The origin of the code to be reused (developed to be reused or reused afterwards)

It was found that black-box reuse was often difficult to implement partially due to the intrinsic difficulties of describing software. However, white-box reuse has the problem of not protecting intellectual property. A middle ground of grey-box reuse was also discussed (leaving some of the implementation details available). It was also claimed that for inter-company reuse only black-box and grey-box reuse are feasible.

Vertical reuse allows for more tailored solutions to be produced but horizontal reuse has the potential to cover more application domains. As discussed in the context of DSSAs, vertical reuse makes black-box reuse feasible due to many of the implementation decisions which would not be documented at software artefact level being documented at architectural level. The implication is that horizontal reuse is overly general, often resulting in un-reusable software due to poor non-functional characteristics.

The consideration of how the software artefact was produced either specifically to be reused (DfR), or by re-engineering an existing artefact was also discussed (DwR). Typically DfR artefacts are more generally applicable, but are more restricted in which implementation assumptions they are allowed to make than DwR artefacts.

A number of factors which need to be addressed in order for successful reuse were found. They were:

- The quality of the software.

- The behaviour of the software
- Hidden design assumptions in the software
- The code requiring changes before reuse
- The identification of relevant code

It was also found reuse needed to be addressed both in terms of technical and sociological terms.

Software quality is a well accepted pre-cursor to reuse. If the software appears to be of inferior quality then software developers have been found to be, with good reason, reluctant to reuse it. Also previously stated, behavioural specification of software is often difficult due in part to its abstract nature. There must be a way of conveying "what the software artefact does" between developer and reuser or re-use will be infeasible. Hidden design assumptions are inevitable in software reuse. However, steps can be made to minimise them, by for example DSSAs that document many of the decisions at architectural level. Better standards on what needs to be present in interface specifications are needed.

In general software reuse, it is unreasonable to expect that the re-user will have exactly the right software artefacts in the form in which s/he expect. In fact there will often be some modification required before reuse as described in Bosch [29]. Important issues such as how to change software artefacts in order for them to fulfil the new requirements without creating new maintenance problems and how to store software artefacts for retrieval need to be addressed. Another way of addressing the reuse issue is by communicating to the re-user what functionality they should expect. Also there should be an effort to reduce the time between finding a potential reuse candidate and knowing whether the candidate will fulfil the re-users requirements.

The following methods have been identified as addressing reuse:

- Software Components
- Software Libraries
- Software Frameworks
- Software Architectures (and DSSAs)

These methods have been discussed in terms of the reuse issues, in order to see how they address software reuse.

3. Related work

There are a number of other areas in computer science that have an influencing factor on software reuse whilst not usually referred to as being part of the field. One of the aims of this chapter is to describe one such important factor, the field of software testing.

A number of alternative approaches to software reuse also exist. This chapter highlights a number of related examples which although often not considered directly related to reuse, aim to achieve some or all of the same goals.

3.1 Software Testing

The aims of the testing section are as follows:

- to clarify terminology
- to provide the reader with a brief background into testing methods
- to outline basic concepts so they can be built upon in later chapters
- to outline related research in automating software component testing/testability

As the last chapter noted, it is important to maintain confidence that the reusable software artefact will perform as expected. However, the reuser often has not developed the code that is to be reused. Therefore it is important to attempt to maintain the reuser's confidence by other means, such as verifying that the software is of high quality.

The field of software process metrics defines a number of ways of quantifying controlling quality throughout the phases of software production. In addition to these metrics, to ensure that the finished product behaves as expected, the component should be tested. However, unexpected problems may arise due to substantially different environments or unexpected usage cases. Hence, component testing is a very difficult task in practice. There may be multiple target environment configurations, all of which could potentially cause difficulties for the implemented system. There is a combinatorial explosion of variants to be tested, including component inter-compatibility testing, OS variant compatibility and hardware environment compatibility. In general, it is impractical to test components to this high level. Hollingsworth [5] researched methods for localised testing and certification of components although could not eliminate all possible unforeseen interactions. Before further discussion, this section will explicitly define terms in order to avoid ambiguity.

3.1.1 Errors

An error is defined as a category of event or state as per Beizer [57]. For example an error may occur due to incorrect data being entered into a system, or by the system reacting inconsistently or incorrectly. It is an indicator that something has not performed as expected. This terminology should not be mistaken for the statistical notion of the value of the difference between the expected and the observed values e.g. the IEEE/ANSI definition. (Std 610.12-1990 in [58])

Examples of errors include 'Out of range values', incorrect values, exceptions and the systems internal state deviating from what is expected.

3.1.2 Faults

Beizer's definition of fault is appropriate for this research:

> A deficiency in a program that causes the resulting effect to be in some way incorrect and hence causes an Error. This is conventionally held to be a inconsistency between the specification and the implementation. [57]

Testing can be performed in a number of ways, dependent upon what is known about the software being tested. Black-box testing is based on interface specification and consists of a number of tests which are only concerned with the external behaviour of the code being tested. Software for which there is no source code available can only be tested by black-box methods. White-box testing, that is testing based on the actual code, consists of tests which are based on the actual internal structure of the code.

The purpose of testing is to evaluate the quality of the software being tested by ensuring it performs as expected (Beizer [57]). Testing is performed in order to detect errors. Another, closely related concept to testing is debugging. Debugging is performed if a system is known to perform incorrectly, i.e. an error has been detected, in order to locate a fault. Many of the techniques used in testing can be used in the debugging process.

The following section aims to summarise a number of appropriate testing techniques.

3.1.3 Black-Box tests

In the context of this thesis, this category of testing is appropriate for ensuring that externally developed applications behave as expected.

A common black-box testing technique is Boundary condition testing. The aim is to ensure that the code behaves as expected in locations where faults are often prevalent, the boundaries of the acceptable values for input. This category of test catches common mistakes such as in Figure 3:1.

```
if (a>=5)
    {
    print "Hello world\n";
    };
```

Figure 3:1

When the program should actually be as in Figure 3:2:

```
if (a>5)
    {
    print "Hello world\n";
    };
```

Figure 3:2

There are a number of techniques for automatically or semi-automatically generating black-box tests. For example, it is possible to generate random test data from interface specifications alone, and this has been performed by Grossman [59]. However, this category of testing is by no means exhaustive, or even satisfactory for a moderately complex system.

Other research into automated testing include rule-based test generation as in Deason et al. [60] and extended-UML based automatic test generation as implemented by Aonix [61].

It should be noted that Black-box tests are limited in so far as they can only test upon factors which are externally visible. Therefore it may be very difficult to test components which contain a large amount of internal state without a way of making certain parts of the internal state externally visible.

As stated in Beizer [57], these techniques should be combined with White-box tests where possible.

3.1.4 White-Box tests

This category of tests requires that in addition to the executable code, the source code is available to the testers. This means that white-box tests are usually performed by the developers[4] of the code. White-box testing techniques include

- Path testing, which attempts to ensure that some/all of the possible execution paths occur. It is not usually possible to test all paths, especially when the program being tested contains loops.
- Data flow testing, where the tester attempts to ensure that the data in the system conforms to the expected pattern of data-flow in a system i.e. being defined, being manipulated and de-allocated. This sort of test is implementation dependent, because certain programming languages have automatic initial values and may also have automatic data de-allocation.
- Transaction flow testing, where the tester runs test cases along use cases of the system. That is to say the tester will ensure that commonly performed operations are functioning correctly.

These techniques are very useful for testing a system when the source code is available. However, they are not applicable on interface level only e.g. when the developer and reuser are different people working for different companies. White-box tests are also more difficult to automate. As previously stated, 100% coverage for Path testing may not be possible for a number of reasons, such as the combinatorial explosion or that the structure of program may prevent it.

3.1.5 Debugging

If the result of the system / subsystem is found to be incorrect, a technique such as program slicing is useful. A program slice (Weiser [62]) is performed with respect to a certain variable, or set of variables, typically when such a variable has been found to be incorrect. The underlying concept is that if a certain result is incorrect then it is useful to examine the parts of the system which could have affected the result. The resulting code is known as a program dice

There are many different program slicing techniques but conventionally they are separated into static and dynamic program slicing. Dynamic program slicing can remove more code, due to run-time decision paths being available.

3.1.6 Self-testing

Self-testing software, as discussed in this thesis, aims to verify its behaviour. In order for this to be performed, there must exist an alternative route or method by which a result can be generated.

A typical example of self-testing in the field of software is N-version programming. Different methods of producing the same system are employed in a parallel development process, often with no interaction between the groups developing the code. These systems are run concurrently and the results compared. If they produce the same answer[5] then there is agreement and the system can assume to be acting reliably. If there are alternative answers, a mechanism for deciding which answer is most likely to be correct will be used (polling).

In the domain of mathematics, the concept of self-testing functions has been researched, as in Rubinfeld [63] and Blum [64]. These functions can be

[4] By developers, it is mean the department or company which developed the code, not necessarily the actual people who implemented the system.
[5] The concept of same is not necessarily simple here. For example there may be numerical inaccuracies which differ dependent upon what method is used.

approximated, within a n% degree of accuracy, by less complex alternatives. The general characteristic of mathematical functions that can be approximated in this way is called random self-reducible functions. One disadvantage of this approach is errors may be detected when none actually exist.

In the field of components and software agents the function of the software artefact may be encoded along with the actual functionality. This quality is often referred to as reflection. However reflection is usually applied to components and agents as a guide to the software artefact can be used for, rather than in order to check that it is functioning correctly.

A different approach to increasing component testability is taken in Hollingsworth [5], that every component should export operations sufficient for its pre/post conditions to be tested by an external client. However, Hollingsworth proposes two versions of each component to be produced, one for testing and one for use. Although this is attractive in terms of performance, it carries the added risk that the component behaviour could change between versions.

3.2 Alternative approaches to reuse

One of the main driving factors for software reuse is to increase the productivity, flexibility and reliability of software by leveraging existing investment. There has been significant related research into achieving the same objective by alternative means. The aim of this section is to survey some of the alternative ways of addressing the same issues along with an overview of how they achieve it.

The following approaches are described:

- Software agents
- Domain-Specific languages
- Automated programming

3.2.1 Software Agents

Software agents address reuse by aiming to increase software flexibility. That is to say by creating systems that can alter their behaviour, software agents aim to increase the domain of applications that a system can provide services for without the need for human intervention.

The traditional procedural method of creating such a system is to write a sub-program to solve the problem for each differing case. It should be noted, however, that extra complexity will almost certainly be introduced by doing this.

One way of viewing software agents is that they are structured in a way which is analogous to how humans work together to solve problems. Rather than a software artefact dealing with one task and being called when necessary by the controlling program (as is typical in hierarchical systems) each agent may act when necessary and in the manner in which it "believes" to be appropriate.

As previously stated, agent technology has been advocated as a solution for developing flexible software. However the field of agents, although extremely promising, has not yet fulfilled this claim. It can be argued that agents are successful in describing problems that map well to the architecture that they use.

Agent architecture, as with many new research fields, lacks a universal definition mechanism. In fact what actually constitutes an agent is still under discussion. For this reason, this thesis will discuss common attributes of agents, in conjunction with how they achieve or, as is more usual, aim to achieve dynamically re-configurable behaviour.

The following are often considered to be defining properties of agents:

- Autonomy
- Pro-Active and/or Reactive behaviour
- Temporal continuity

When an agent is described as Autonomous, it is meant that there is a non-hierarchical flow of control. Each agent has the capacity to behave independently to stimuli. There exists debate to what extent agents have to be autonomous. At one extreme are agents that will simply perform their tasks with no interaction of any type with other agents (human or otherwise). These could be said to be highly autonomous because all decisions are made by that agent only. At the other extreme is an agent which asks for advice when reaching any form of decision, which could be said to have no autonomy. It is unclear if the autonomy characteristic increases the flexibility of the resulting system.

Wooldridge and Jennings [65] describes an agent to be reactive if it only acts upon external stimuli. That is to say it only acts in response to its environment. An agent is Pro-Active if, rather than waiting for certain conditions to arise, it affects the environment in order to achieve an effect.

There are a number of supporters for the argument that agents must not just react to the environment. Instead at some point in the agent's "life" it must take control and behave proactively (e.g. Franklin and Graesser [66] and also as the "weak notion of agency" Wooldridge and Jennings [67]). This implies a combination of an event driven (**Reactive**) and a goal driven (**Pro-Active**) control system rather than simply an event driven one. However other authors claim that Pro-Active agents are just one of many different agent system types as in Farhoodi and Graham [68].

The temporal continuity quality of agents is often described informally as "*...a continuously running process*". However, there is no reason why an agent should not be able to suspend execution for a specified amount of time or terminate. If the system contains mobile agents, that is to say agents which can move from one system to another by some mechanism, then there is definitely a time when the agent is actually in transit and not actually present on any machine. The concept that the agent must be able to maintain its execution state is preferred here. As has been pointed out in the agents mailing list [69], it may be unnecessary to be able to suspend execution at every state, rather that the state may be stored at certain points in an agent's execution and the agent re-assembled at another point with the same state. To re-iterate the point that temporal continuity concerns more of continuity of the program execution state rather than the continuity of the actual program which of course is an illusion in a multitasking uni-processor system. Temporal continuity does not appear to have any direct bearing on software agent's flexibility.

Having discussed potentially defining properties of agents, it is also important to consider possible attributes that may be included in agents. These attributes directly address the flexibility issue:

- Communicative agents
- Co-operative agents
- Learning agents
- Mobile agents
- Negotiating agents

Communicative agents communicate with other agents to help solve distributed tasks. Communication can be achieved by using an Agent Communication Language, such as KQML (Knowledge Query Manipulation Language) and KIF (Knowledge Interchange Format) or FIPA (Foundation for Intelligent Physical Agents[70]). Communicative agents may achieve flexibility by requesting services from other agents, rather than calling specific code, in a similar way that Object Oriented programs maintain a degree of flexibility by requesting for an object of a base-class type rather than simply identifying the specific class of the original usage.

Co-operative agents have the ability to team up to solve problems. Much research has been performed on the problems associated with agent cooperation in the field of planning. Co-operation is possible over systems where there exists a common understanding. Agent co-operation has great promise in improving flexibility as new behaviour could be produced when necessary. However, conflicting ontologies, lack of a method of expressing the task, or ambiguity can make co-operation infeasible.

Learning agents interpret and maintain an abstract model of the environment in which they are situated. Learning agents can be seen as flexible in that they can tailor themselves to their environment and may act, in a non-repetitive manner, to these external stimuli. The success of the agent is dependent upon the learning algorithm used, and whether the abstract model is sufficiently accurate in representing the external conditions. Learning agents have the added disadvantage that they often act in unforeseen ways that are undesirable as documented by Caglayan et al [71]. An example of a learning agent is the Microsoft Office assistant, which monitors the way in which the user interacts and "helps" the user.

Mobile agents can suspend their state and then move themselves to a new computer system and restart with the same internal state. This mobility can also be seen as a type of flexibility, as the software is not confined to one computer. Mobile agents can have very serious security implications.

Negotiating agents are useful in systems where there is no clear optimal solution, or where the optimal solution is dependent on perspective. For example, if an agent is trying to sell a commodity to a second agent, then each agent has a different concept of the best result. The seller would prefer to sell at a high price, and the buyer would prefer to buy at a low price. Negotiation allows for flexibility between two parties with quite different goals and has been implemented in market simulations as in Chavez and Maes [72] and for network quality of service Nygren et al. [73].

In some agent systems the knowledge bases are distributed and maintained separately for different agents. This is analogous to people who work in a company having different knowledge and views. Problems may arise when different agents share information, especially when they may use different ontologies creating contradictions or at least inconsistencies. The analogy continues in that certain people may not agree with each other despite their individual goals being compatible.

Distributed agents are a promising method of providing a natural metaphor with which to describe certain types of systems which otherwise would probably be made centralised. This is helpful because systems which are developed using appropriate metaphors and architectures can be easier to understand and hence to maintain.

One area of difficulty in software agents is the way in which the requirements are described and for communicative and co-operative systems, how the subrequirements are interchanged. One attempt to solve this is contract-based agents as in Krogh [74]. Contract-based agents are based on the way that humans actually offer services, i.e. the two parties enter a contract of service. There are clear analogies between interface specifications for components and these contract-based agents.

It is unclear whether software agents are not currently delivering upon their promise due to impossible expectations, or because new design and implementation methods are required in order to make them feasible. This topic is discussed in greater depth on agents in general (Wooldridge and Jennings [67], Shoham [75], UMBC [76] Ingham [77]), on agent communication (Labrou [78], Pearson [79]), on agent architectures (Wooldridge and Jennings [65], [67] and Franklin [66]) and designing and building agents (Farhoodi and Graham [68])

3.2.2 Domain-Specific languages

A common question asked by people who have never programmed a computer is "What is the best language?". Usually the answer is that different languages are good at performing different tasks. This is true even of so called generic languages. C is good for developing low-level applications and for maintaining control of the performance of the resulting system. However there are a number of applications for which C is particularly weak. Arrays, for example, are not bound-checked. However if C was "fixed" by altering array implementation then its applicability to other problem domains may be hampered, i.e. array access may impose a higher overhead.

The underlying philosophy of Domain-Specific languages (DSLs) is that although there is a place for generic, cross-application languages, the most optimal way of expressing a new system is often by using specialised languages.

There seem to be two differing structures to DSLs:

- Extension to other languages such as P++ as in Singhal [80] (called DSL extensions)

- A restricted language which only deals with the problem domain (as in Bentley [81], Ward [82], Deursen and Klint [83], and Spinellis and Guruprasad [84]).

By extending languages, certain weaknesses in the language can be overcome. For example the weak support for component interface description in C++ has been enhanced by P++.

Alternatively, creating restricted languages emphasises a clear separation of concerns. It must be possible to integrate the DSL system with other commonly occurring systems or it will not be accepted into general use. A possible solution is to provide a mechanism in which this language can plug into a generic language, as sometimes performed in SQL. According to Deursen and Klint [83], a restricted DSL should:

- have a focussed domain with clear boundaries.
- have as few constructs as possible whilst retaining the ability to define the range of problems for the target user.
- be unambiguous.

3.2.3 Automated Programming

For the purposes of this research automated programming is the attempt to reduce the effort and associated financial risk when developing a new system/adding a new functionality. By this definition, automated programming is a wide field that sometimes is defined to include reuse and software development technology.

Many of the systems developed under the label of automated programming have become very well accepted, such as high-level languages including COBOL and FORTRAN. An excellent survey of some automatic programming techniques is given by Rich and Waters [85]. Many older surveys exist, a number of which are outlined in Neighbors [86]. However there is an inherent difficulty in surveying software production techniques as per Feldman [87].

"Almost anything in computer science can be made relevant to the problem of helping to automate programming."

A few, well known, examples of automated programming systems have been chosen to illustrated concepts relevant to this thesis:

- DRACO
- RESOLVE
- GenVoca

3.2.3.1 DRACO

DRACO (Neighbors [88],[86]) is a semi-automated transformation-driven software generation process. The user specifies their problem in terms of a domain-specific language, after performing domain analysis. This description is then transformed, with help from an expert user, into a functioning system. This process is useful in that domain-analysis and problem refinement process can be seen as an investment which, if performed correctly, can be reused later in the product lifecycle. That is to say DRACO can be used to develop similar systems by reusing the data collected during the process of previous system developments.

DRACO is significant in that it emphasises the importance of domain analysis and vertical reuse, although Neighbors acknowledges Balzer [89] provided this philosophy. From this analysis a DSL is formulated. Problem specifications are developed using this language. This is an effective method of defining a well-understood problem domain. The reliance on developing a DSL has been viewed as a strength (Taylor et al. [8]) and as a weakness (Singhal [90]). However the process of transforming the specification from one form to the next still requires expert user assistance. This process of refinement is non-trivial and, as documented by Neighbors [86], can involve a large investment of effort in paths of investigation which do not lead to the goal state. This is a barrier to reuse as a programmer is likely to lose confidence in using a system will not definitely provide a solution after a substantial investment.

3.2.3.2 RESOLVE

RESOLVE, Hollingsworth [5], Bucci et al.[91] provides an interface specification language which extends the ADA programming language, along with some programming "principles" or guidelines. The aim is to maximise re-usability by developing components in conjunction with certain standards. Although the work claims to be language independent, a significant quantity is ADA specific.

Of particular interest is the method of introducing component testability. Hollingsworth [5] addresses this need by developing components so that they can be tested in isolation, rather than having to consider the client in conjunction with the component. Although it is an extremely attractive proposition to be able to develop and test the components in isolation, it should be noted that this makes many assumptions about whether one software artefact can affect another software artefact, even indirectly. It is unrealistic to assume that each component can only be affected directly, even if the principles outlined in Hollingsworth [5] were adhered to. Therefore, the system would still need conventional tests at each stage of component composition.

The principles highlight some important concepts. Principle 5, "a component must export sufficient operations such that it's preconditions can be ensured", is often left unconsidered in component design. However, this leaves the problem of accessing a component's internal state through an abstract interface. There must be a standardised way of reporting the external state. This means that even the internal state of the component must be displayed as an abstract state rather than in the way most natural for its implementation. It also means that the specification of the system must be detailed enough to consider this level of detail.

3.2.3.3 GenVoca

The GenVoca paradigm describes the construction of Software System Generators (SSGs). Singhal and Batory [80] describe three main features of GenVoca systems:

- The amount of code which is typically reused is larger than one function or class. They refer to this quantity as a subsystem or a component.
- Components must refer to each other through explicit interfaces.
- Components are composed and customized by parameter only.

The following are examples of the GenVoca paradigm.

- ADAGE – Batory et al. [9] This marks a cross-over between the domains of DSSAs and SSGs. The Domain Analysis is used to develop a model of the domain. This model is used to help describe the reusable components.
- P++ - Singhal and Batory [80], Singhal [90]. P++ is an extension to the C++ language. The extensions are concerned with describing components in terms of the GenVoca model. This research highlights ways of overcoming weaknesses in C++, such as templates, by better dealing with parameterization at differing levels of abstraction. However, as outlined in Singhal [90], these extensions to component specification do not consider conditions in which the component will perform satisfactorily.

3.2.3.4 Automatic programming methods.

There are a number of alternative methods for automatically generating programs. A survey of these is performed in Rich and Waters [85]. Of particular interest for this thesis are the two alternatives of Transformational or Compositional programming methods.

Transformation programming methods, as applied in DRACO, are characterised by requiring user intervention in order to generate code. This is due to the enormous number of possible applications of each transformation. The strengths of transformation systems, as stated in Rich and Waters [85], are that

> *"they provide a very clear representation for certain kinds of programming knowledge, such as Horner's rule."*

As previously stated, DRACO is based on a transformation programming methodology.

Compositional programming methods (described as a kind of procedural method Rich and Waters [85]) are used to compose code from other constructs. Compositional programming usually involves constructs of a larger scale than single programming constructs. Much of the research in the GenVoca project has been compositional in nature, especially their Software System Generators.

3.3 Summary

This chapter has dealt with two separate issues. The first issue was that of software debugging and testing. This is provided to give a brief outline of testing methods and to provide the terminology to be used later in the thesis. The underlying philosophy of this section was to reduce the risk of ambiguity due to clashes in terminology.

The second issue covered by this chapter was alternative methods of achieving the same goals as the more conventionally accepted software reuse methodologies outlined in Chapter 2. These are Software Agents, Domain-Specific languages and Automated Programming. Each of these was described in terms of a brief overview and how they address problems which are similar to the reuse one. The concepts described in the last two chapters are now applied to the thesis, that of applying a DSL-based approach to component composition, error-detection, and fault-prediction.

4. Concept

4.1 Introduction

As discussed in chapter 2, black-box reuse has currently more potential than white-box for inter-company reuse. This is due in part to the lack of protection of intellectual property inherent in white-box reuse. This requires no small amount of trust that the reuser will not reverse engineer the software artefact[6]. A particularly salient field to modern black-box software artefact reuse is software components. As discussed in chapter 2, software components are software artefacts which obey certain stipulations in order to maximise their reusability.

In software component reuse, significant difficulties still exist because there is no clear concept of what information should be recorded and what can be abstracted. It is not reasonable for a software developer to be expected to record all data about a component when the conditions of reuse are not known at development time. This problem can also be viewed from the potential reuser's perspective. There is often no guide for the re-user as to under what conditions software components are designed to function. Even when such a guide is present there is no guarantee that there is sufficient data to make a fully informed decision. A method of reducing this problem is Domain-Specific Software Architectures. By making certain decisions at design time and documenting them extensively there is the potential for the contextual dependencies and design decisions to become standardised across the domain. This explicit information will ease the decision process of whether a component can be reused but at the cost of generality. This cost must be weighed against the cost of documenting and understanding the individual component specification issues at component development time and location and comprehension of the documentation at reuse time.

[6] It should be noted that black-box systems can also be reverse engineered to create a prototype, typically to create a specification of how the system acts. There are also a number of techniques to attempt to recover source code from the object code. However these are both significantly more resource intensive than simply reading the source code.

An important issue in software component reuse is component faults. It is assumed that the software will be developed using good practice in software engineering and that it has been tested to a reasonable degree. Unforeseen behaviour may still result. This may be because the model upon which the software was developed is insufficient to describe the complex interactions (a design error), a compiler fault, an ambiguity in requirements or simply an error in coding.

As previously stated in chapter 2, there is also a large incentive to reduce time taken for the stages of requirement analysis to implementation. Achieving this can create a competitive advantage for companies who are increasingly reliant on computer systems to help provide new services. This has been made possible by using more problem focussed development and programming environments. Because these Rapid Application Development systems are focussed on certain types of common problems they are based upon domain-specific reuse.

The approach outlined in this thesis is also described in Ingham [92].

4.2 Hypothesis

This chapter contains the concepts proposed in this thesis. They are described in terms of a main hypothesis, and where deemed appropriate, sub-hypotheses which describe concepts in more detail. It is hoped that by separating the hypotheses, a separation of concerns can be achieved, hence enabling the separation of certain issues.

The main underlying hypothesis of this research is:

Given a sufficient number of components, described in sufficient detail, it is possible to automatically develop systems in a manner such that human intervention is minimised.

By "a sufficient number of components" it is meant that there must be sufficient components in existence to describe the new software, assuming that they could be assembled by some mechanism. A more vague term in this hypothesis is "in sufficient detail". Before it is known what level of detail is required, the problem description must be formulated. As stated earlier (chapter 2) this is a non-trivial problem for a developer of reusable components who may not be able to foresee which factors are important. More importantly, a potential reuser should know what detail to expect in such a system. In order to develop such a description mechanism, a model of the problem domain is required, or Domain model as in Taylor et al. [8]. Requiring the existence of an unambiguous problem domain model made a DSL a very attractive specification mechanism because it has been used under those conditions in conjunction with DSSAs Batory et al. [9]. By utilising a DSSA where possible, common architectural issues could be abstracted away to the level of the entire system. That is to say every component in the architecture would operate under the defined conditions. This mechanism reduces the number of factors that the reuser needs to consider at component level and hence makes their task easier.

Further investigation into component specification, in particular into how DSLs could be used, yielded the following sub-hypothesis:

A DSL can be applied to specify the behaviour of components. The specification can be used for the purposes of locating components and verifying that their behaviour matches what is expected.

A DSL requires a stable and well-understood domain. If the domain does not fulfil these requirements, then it would not be economically sound to use this method of software development. Sacrificing generality for an unambiguous problem requirement which could be translated into code by automatic means was viewed in this research as an acceptable compromise. Furthermore the requirements can be described in more tailored terminology making it possible to improve confidence that an operation is successfully completed by automatically generating pre/post operation tests. This was described using the following sub-hypothesis:

Given a specification of the problem modelled in sufficient detail it is possible to detect some errors occurring in a system and in some cases find the cause of those errors (i.e. faults).

The question of what constitutes sufficient detail is discussed, in terms of error detection and fault prediction, later in this chapter (sections 4.5 and 4.6). Since the automated programming environment may have redundancy in the system, it could be possible to reconfigure its behaviour to attempt to avoid the faulty component. This is described using the following sub-hypothesis:

Upon detection of an error, the system will attempt to ascertain where the fault lies. Then a new solution will be created if such an alternative is feasible within the system.

To summarise the hypotheses, they state this research will:

- Use a DSL as a component specification mechanism. This mechanism will be used to define behaviour in terms of semantic naming and behavioural pre-post conditions.
- Translate the requirements, which are in terms of the DSL, into an executable program, assuming sufficient components exist.
- Upon detection of an error in the resulting system, ensure the supporting architecture will use available data to predict the location of the fault. The supporting architecture may then attempt to generate an alternative solution.

This chapter discusses the concepts that this thesis is based upon, and to emphasise the original contributions made here. They fall into these categories:

1. Support architecture
2. Domain-Specific languages.
3. Error detection
4. Fault prediction

ORIGINAL IN COLOUR

4.3 Support architecture

As previously stated, this thesis proposes the use of an architecture which supports the underlying cycle of generate executable, detect error, predict fault and then re-generate code. An implementation of this architecture should, where possible, keep any underlying complexity separate from the generated executable. It is important to keep the executable as simple as possible as complexity is often an indicator of faults.

Another design consideration is that the architecture should be robust, that is to say that certain parts of the architecture could change (e.g. different fault prediction strategies) without unnecessary changes being impacted on the rest of the system.

Figure 4:1 shows the data-flow around the architecture:

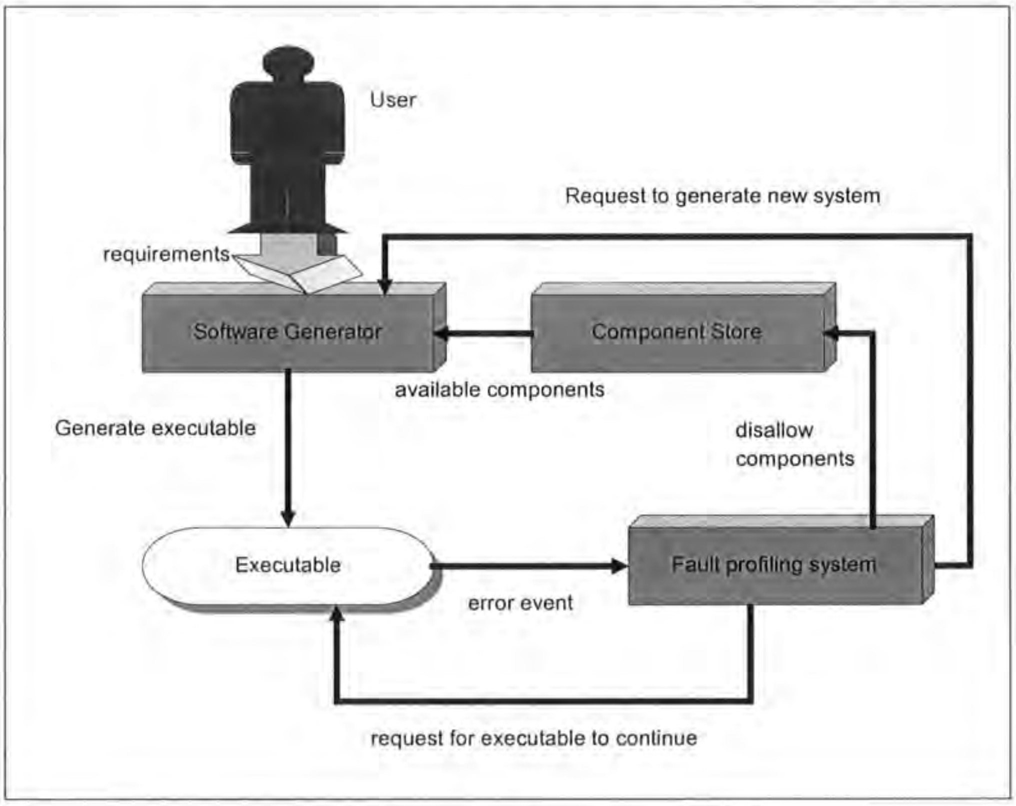


Figure 4:1

Each part of the architecture is now described:

1. Software generation
2. Executable

3. Component store
4. Fault profiling system

4.3.1 Software generation

There are a number of possible alternatives for translating a high level problem definition into an executable. Traditional compiling techniques are quite successful in translating one language into another, typically a lower-level one. However this task involved translating requirements, described in terms of a DSL, into a collection of black-box components, verification code and glue code. Such a system may often be referred to as being an "automated programming" system.

An important requirement, often omitted, in automated programming is reliability. If the user does not have confidence in the executable produced then there is a high likelihood that future executables will be produced by other means. To remain reliable, an automated programming system must control complexity because:

- There is a direct link between complexity and faults.
- Too much complexity will make tracing how the resulting system was produced difficult, which may be required in fault-diagnosis, in cases where a software generation technique is responsible rather than the software being reused.

Data about the structure of the executable is encoded into the executable itself, in order to allow for reflection. This is used in the fault diagnosis process.

Some common automated-programming techniques have been outlined in chapter 3. The compositional programming technique, based upon the DSL requirement definition language approach is most appropriate for this thesis. There were a number of criteria for this decision:

- There had been a number of successes in this field as documented in Batory et al. [9], Rich and Waters [85] and Stichnoth and Gross [93].
- The concept of the code generation process would be well-understood, as it has been applied by many authors (e.g Mueller [94] and Batory [95]).

- The system can scale to any number of components, providing they are all specified in terms of the DSL.
- Compositional model code lends itself well to error-detection and fault-prediction, as the boundaries of components are well defined.

The software generation process could be separated into a number of simple steps, as illustrated in Figure 4:2.

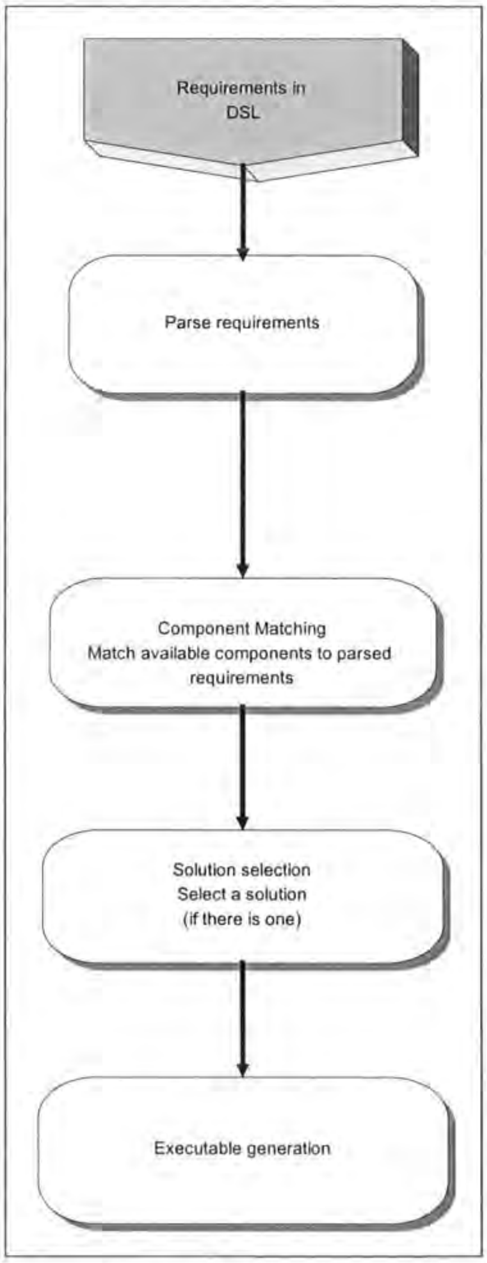


Figure 4:2

This section will now explain each stage of solution generation.

ORIGINAL IN COLOUR

4.3.1.1 Parse Requirements

The requirements are parsed, checking that the type of each parameter obeys those specified in the DSL.

```
A { A.1 A.2 { A.2.I A.2.II} A.3 }
```

Figure 4:3

They are broken down into a tree. For example, the DSL requirement as shown in Figure 4:3, where A, A.x, A.x.y are components such that A.x is a parameter of A and A.x.y is a parameter of A.x and A.1 is the first parameter and A.2 the second parameter of A, will generate a tree as illustrated in Figure 4:4

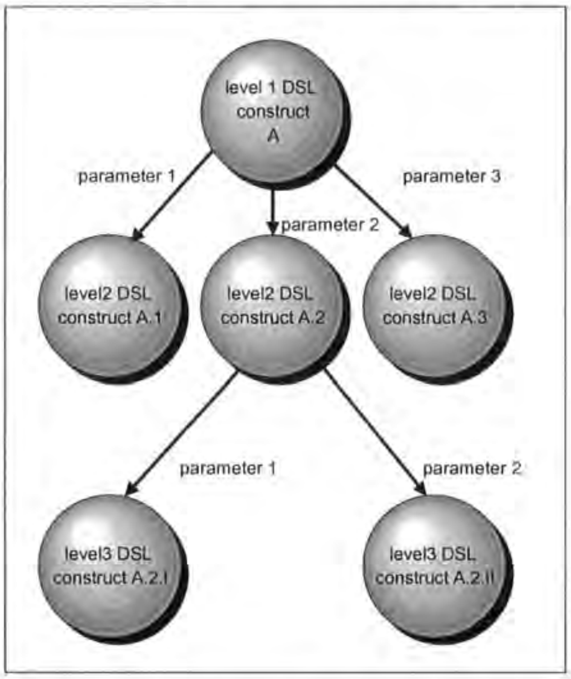


Figure 4:4

4.3.1.2 Component Matching

Each component is described in terms of the DSL constructs. This means that every component can fulfil the behaviour described by one or a collection of DSL constructs. The algorithm which is deemed most appropriate is bottom-up matching of sub-trees, as it avoids infinite recursion, given that the requirements tree and number of components in the system are both finite.

A solution for the component matching is found when the whole tree has been matched to components. Typically there will be more than one way to assemble the components

For the purposes of example in Figure 4:5, there could be three components:

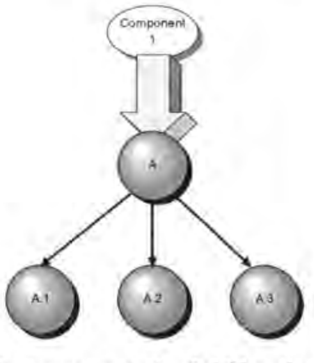
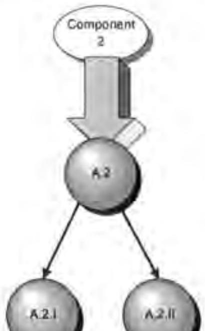
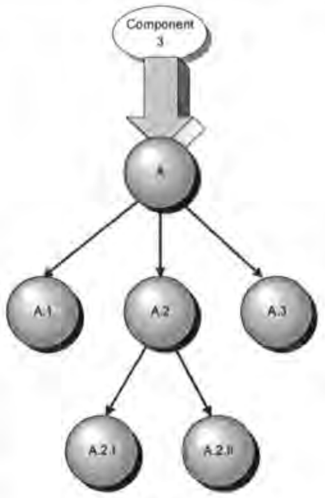
 Component 1 fulfils the requirements of DSL Constructs A, A.1, A.2 and A.3.	 Component 2 fulfils the requirements of DSL Constructs A.2, A.2.I and A.2.II.	 Component 3 fulfils all the DSL requirements

Figure 4:5

In this very simple example two solutions are available. Component 3 fulfils the requirements on its own. Component 1 and component 2 used in conjunction also fulfil the requirement.

If a component's behaviour is suspected of being faulty then the component may be omitted from the pattern matching algorithm.

When the matching algorithm finishes there are a list of complete solutions. Each solution should contain:

- The structure of the solution.
- The rules which were applied to make this solution.
- Any metrics or solution scoring systems which have been applied.

4.3.1.3 Solution selection

After the system has generated alternative component compositions which fulfil the requirement, one solution must be selected which will be used to generate the actual executable. Each component may be scored with reliability metrics which may be used in the decision making process. These metrics may be updated with further run-time information for the system. Alternative approaches for systems with no metrics guidance include:

- selecting the solution with as many components as possible as this is the nearest solution to the DSL granularity level and as such will have more tests based on DSL construct specifications.
- selecting the solution with as few components as possible as if something does not work there are less components which can be suspected.
- Select a solution at random.

4.3.1.4 Executable generation

The selected solution is converted into source code. For each component, the pre-condition, post-condition and WBT tests are inserted automatically, in the target language. This stage is the only one which needs to be language dependent. Alternative languages could be used as implementations by changing this stage only.

Extra data is encoded into the executable. This includes:

- At each component level, a program slice of what has executed before this point. This data is used in the fault-prediction stage.
- Error messages which correspond to those defined in the DSL construct behaviour specification.
- Code which, upon an error event, communicates with the fault-prediction system

4.3.2 Component store

The component store is an abstraction for a, potentially distributed, collection of components which may be stored in many different locations. Any component which is to be used in the architecture must be registered with the component store.

This is not a new concept (it appears in a very similar form as a Object Request Broker (ORB) in the CORBA Orfali and Harkey [96]) and has only been included for completeness. It should be noted that ORBs do not usually use DSL encoded data, although they could easily be made to do so.

4.3.3 Fault profiling system

The fault-profiling system is continuously waiting for the executable to request its services. Upon detection of an error the executable reports the error message, the component in which it was detected and also the structure of the system.

If there is run-time profiling of components (for example to measure performance or reliability) the executable may report this data as well.

The fault-profiling system then acts in a way consistent to the algorithm chosen for it. Possible actions include:

1. Telling the executable to continue.
2. Requesting a new system from the system generator.

4.4 Domain-Specific Languages

The term Domain-Specific language (DSL) is ambiguous (as is described in chapter 3). However, for the task outlined in the hypothesis, it was necessary to extend the notion of what the DSL should define. Guidelines for developing DSLs have been developed.

The approach taken here, and in much research connected with DSSAs is that a DSL should be designed in conjunction with a domain model. Developing a DSL should be a tool for recording the terminology that a certain class of user expects to use.

For this research a DSL should have a target category of user. Each different category of user may require their own DSL although there is scope for combining the interests of sufficiently related user categories. For example a database may have a category of user such as data-entry personnel who may need operations such as add and delete record. An administrator would have a different focus with operations such as back-up database and check integrity. The DSLs could be used as a totally separate system or, as is more usual, integrated into a generic system. There is a compromise between having one DSL covering all the categories of users in the system and a DSL for every different type of user. The former leads to difficulties due to complex grammar and an unfocussed DSL which is not optimally designed for anyone. The latter option means that people whose jobs fall into several categories may have to learn several DSLs. The optimal solution lies somewhere between these two extremes.

All the DSLs which belong to the same category of problem as viewed by different users (i.e. problem domain) should have the same architectural constraints in order to maximise the reuse of components which fulfil user's roles in different categories. This will also alleviate some of the difficulties for users who still need to utilise a number of DSLs, as some degree of familiarity should be maintained.

The DSLs could be used in isolation or, as would be more usual, integrated into a generic system.

A DSL construct is the smallest unit available by which to describe a component. If a component is smaller in granularity than the DSL construct, then before it can be used in the system it must be composed with other components until it reaches the construct scale.

Each DSL construct has the following parts:

1) Semantic specification.
2) Parametric specification.
3) White-box specification.
4) Behavioural specification.

This section will discuss each part of the DSL construct and show how a sample DSL construct may be described (in this case an Add operation for a database system)

4.4.1 Semantic specification

The semantic specification is the name of the construct. Ideally the name should maximise the familiarity of the target user with the concept which is available. However there is an inherent risk that by using familiar terminology, some ambiguity may also be associated with that term. To reduce the risk of such ambiguity and increase comprehension, each DSL term should be accompanied with a detailed description of how this construct is to be used, in natural language (e.g. Figure 4:6).

Operation Name:

Add.

Description:

This operation adds a record to the database. The record must not already exist in the context of the database.

Figure 4:6

4.4.2 Parametric specification

This outlines the arguments which should accompany the DSL construct. Each field has a type. The underlying representation of the type should be considered independently. For example a type could be "surname" and the underlying representation could be "string". There is a distinction to maximise the evolution potential as it may be necessary to alter the underlying representation to encompass more types whilst allowing existing user requirements to remain unchanged. However, there must be a method for providing input for the parameter when specifying a value. The mechanism for data-entry in this system is making all the types in the system support a string interpretation. See Figure 4:7.

e.g. the parametric specification for Add could be:

Surname, Firstname, UserID and return type HasWorked (a Boolean value)

An instantiated request may look like:

Add Surname "Ingham", Firstname "James", UserId "25"

Figure 4:7

4.4.3 White-box test specification.

To define the behaviour of certain components requires more than the concept of input parameters and output parameters. Components may also contain state, meaning that the behaviour may not be purely dependent on input for its next output. In order to verify that such categories of components are behaving correctly, the concept of a white-box test (WBT) specification is proposed. This breaks abstraction in order to provide information that may be necessary to ensure the component is operating as expected. However, by breaking abstraction, this can limit the categories of components which fulfil the DSL requirement. Therefore it is important that the WBT is discussed in terms of abstract state.

By abstract state, it is meant that every foreseeable component or collection of components which implements this requirement must hold sufficient data to be able to calculate these values. See Figure 4:8.

e.g.

For the add component, there is the abstract concept of the number of records.

NoRecords (Type Integer)

This provides a method of ascertaining how many entries have been made independent of how many entries each record is actually stored as.

Figure 4:8

Although it would be attractive to specify the behaviour of the component in a less-abstract manner, this is not necessarily a good idea. For example, the behaviour of the database could be described in terms of the actual records. This is also in terms of abstract state, since the records have to exist in some form for all implementations. However, the performance overhead to perform these checks may be prohibitive. There may be a case for differentiating between WBTs which are designed to be completed every time the component is used and WBTs for detailed testing. For the purposes of this thesis, the former are more appropriate.

In the current system every WBT that is used to specify behaviour at an abstract level for the DSL construct must be defined for every component which implements that construct.

4.4.4 Behavioural specification.

The aim of the behavioural specification section is to provide an abstract representation of the component's behaviour defined in terms of preconditions, post-conditions and WBTs. This specification is based on a model of the component. This is achieved by defining the behavioural specification in terms of a different DSL. To reduce ambiguity the DSL which considers the abstract behaviour of a DSL construct will be referred to as DSLTest. Any reference to DSL is concerning the construct whose behaviour is being specified. The task of DSLTest is to provide a mechanism for translating automatically the description into executable code. DSLTest has the following inbuilt constants (Figure 4:9):

Variable name	**Purpose of variable**
Inp<n>, n is >=1	For a DSL construct, Inp1 takes the value of the first argument, Inp2 the second and so on. This provides the language with a mechanism with which to discuss input parameters. The Inp values are defined before the operation is performed, to allow them to be used in preconditions.
Out	This provides the language with a mechanism with which to discuss return values. Out is only defined after the software component has been executed and hence can only appear in post-conditions.
WBTPre	This variable holds the value of the WBT before the software component is executed. Therefore it may be used in the precondition.
WBTPost	This variable holds the value of the WBT after the software component is executed. It may only be used in the post-condition.

Figure 4:9

In addition to the inbuilt constants DSLTest also contains the following operations (Figure 4:10):

Operation	Description of function
DefineCondition	This is used to define a behavioural rule such as the first input variable must be greater than 5.
CheckCondition	Will use a behavioural rule as per DefineCondition. It is also used to define the error message which is output if the condition is broken.
DefineVariable	This can be used to create a new variable. The variable's type must also be considered when creating.
AssignToVariable	Allows a variables value to be altered
Equal	Comparison check.
NotEqual	Comparison check.
Greater	Comparison check.
Less	Comparison check.
GTOEqual	Comparison check (Greater than or Equal).
LTOEqual	Comparison check (Less than or Equal).

Figure 4:10

If the behaviour of the implementation of the DSL construct (i.e. the component or components) deviates from the behavioural specification, then an error has occurred. Further guidelines on the behavioural specification process is discussed in more detail in the section on error-detection. Figure 4:11 gives an example of the behavioural description in terms of DSL-test.

e.g.

PreCondition of Add.

<none>

Postcondition of Add.

DefineCondition(CheckDataAdded,Equal(WBTPost,WBTPre+1)

CheckCondition(CheckDataAdded, "The number of records in the database did not increase by one")

/* This check verifies that the number of records currently present in the

```
database is one more than before the record was added. */
```

Figure 4:11

4.5 Error detection

The process of detecting errors in systems is non-trivial. In more conventional program systems this is the task of system testers. In automated systems, error detection is sometimes performed by using alternative versions of the system or perhaps encoded data about the task, to provide robust behaviour. In the approach described in this thesis, automated error-detection is achieved by comparing the model of the component (or group of components) behaviour which is encoded at DSL construct level to the actual behaviour as exhibited by the components.

The error-detection system is based on the following principles:

1) The system should only detect errors when faults are present
2) The system may not, in certain cases, detect an error when it occurs.

These principles have a number of implications. By only detecting errors when one has actually occurred, a working system will not be needlessly altered. Hence, the behavioural specification of the component(s) at construct level cannot be stronger than necessary, as it will contravene principle 1. However the behavioural specification can be weaker than necessary, as stated by principle 2. The importance of this principle is that it is not always easy to specify the behaviour of a DSL construct in abstract terms. For example, it is difficult to specify a user interface in abstract terms.

To restate, it may not always be possible to generate tests which have the same strength as the component's requirements. In these cases a test which is weaker than the components requirements should be used.

The error detection scheme is totally reliant on the underlying model of the domain being sufficiently detailed to describe the behaviour of the system. In cases where the model becomes too complex, architectural assumptions can be made which greatly simplify the model. In the Database example, if the Database is concurrently accessed by external clients then the behavioural specification used is too simplistic i.e. the number of records may change arbitrarily while the system is adding a record. This could cause random error events despite no fault being present[7] in the implementation. However if details of concurrent access can be omitted without degrading the functionality of the system then this makes the domain significantly simpler.

Other examples of architectural issues potentially making the model of the problem domain more complex include memory allocation issues, and communication link availability.

This is why the DSL must be developed in conjunction with a domain model which is detailed enough for the system to be predictable. In practice this requires developers who are familiar with the domain and the problems associated with it.

Although this can be viewed as a limiting factor there is evidence that the effort of domain modelling can improve the quality of the resulting system and, if done correctly, will mean that the systems have more predictable development times because there are fewer surprises[8]. Domain models have also applied in the Unified Modelling Language approach (Booch et al. [97]).

Despite the effort of modelling it is still accepted that the system will not be totally predictable. This is due to the inherent complexity of the hardware and software which still acts unexpectedly under certain conditions, which could be viewed as emergent behaviour. As total predictability is not feasible, the improvement in the confidence in the system must be worth the effort expended.

[7] By considering a fault to be an inconsistency between the specification and implementation, the implementation does not contain a fault. The problem lies in the model of the system and hence is resident in the specification.

[8] The example of concurrent access to a database is a very simple issue which in practice would not be overlooked by an experienced programmer.

4.6 Fault prediction

This process is started upon detection of an error. The system has the following information:

- An error which has been detected at a certain point in the program provided by the error-detection scheme.
- A program slice of the components provided by the executable itself and generated at compile time.

With this information the system must predict where the fault has occurred. There are potentially many different fault-prediction strategies which are dependent upon which assumptions can be made.

The following assumptions have been made for all the strategies:

1. The architecture is not directly causing the fault.
2. The fault can be corrected by removing some or all of the components.

In terms of automated reconfiguration, without these assumptions there is no point in finding where the fault is as there is nothing that can be done about it. Assumption 1's meaning is that although the architecture may cause faults by unforeseen interactions with certain components, it is assumed to not simply generate random data between components.

There is an extra level of indirection present in this system which may also be at fault, the component specification mechanism. For example a functioning database component may be accidentally mapped to a user interface DSL construct. Unfortunately there is no way of distinguishing between a faulty component and one which is being used for the wrong task. There needs to be some underlying notion of the behaviour which is trusted or otherwise the system has no frame of reference

However the rules which are an intrinsic part of the component matching algorithm could potentially contain faults. Although the system maintains a list of the rules which were used in system generation there is currently no fault-prediction strategy based on this data.

4.6.1 Fault prediction strategies

This section explicitly lists a number of strategies that can be applied. Chapter 7 contains the detailed evaluation of some of these strategies, considering strengths and weaknesses.

One assumption that can be made to considerably simplify the fault-prediction strategy is that only components which appear in the program slice could be at fault. This reduces the fault-prediction problem to selecting items from a list.

Strategies include:

- Do nothing - The fault may be intermittent.
- Select any one component and remove its use from the system.
- Remove the component in which the error was detected.
- Select any one component but make the component in which the error was detected more statistically likely to be chosen.
- Select any one component but make any components which have been suspected of faults before more statistically likely to be chosen.
- Utilise metrics on the components to help decide which component is at fault. This requires the underlying system to maintain statistics on the number of times a component has been suspected.
- Select multiple components with one of these schemes.

Similar strategies for systems which do not rely on the faulty component being present in the program slice, but are based on the list of all components used. This could be easily compiled from collected data.

- Do nothing - The fault may be intermittent.
- Select any one component which has been used and remove its use from the system.
- Remove the component in which the error was detected.

- Select any one component but make the component in which the error was detected more statistically likely.
- Select any one component but make any components which have been suspected of faults before more statistically likely to be chosen.
- Utilise metrics on the components to help decide which component is at fault. This requires the underlying system to maintain statistics on the number of times a component has been suspected.
- Select multiple components with one of these schemes.

Some of these fault-prediction algorithms are evaluated in chapter 7.

4.7 Summary

The aim of this chapter is to explain the underlying philosophy of the approach taken, to outline the original contributions made and to provide an outline of the parts of the system which are discussed later in the thesis.

The following main hypothesis was proposed in this chapter:

> *Given a sufficient number of components, described in sufficient detail, it is possible to automatically develop systems in a manner such that human intervention is minimalised.*

In addition, the following sub-hypotheses were proposed:

> *A DSL can be applied to specify the behaviour of components. The specification can be used for the purposes of locating components and verifying their behaviour matches what is expected.*

> *Given a specification of the problem modelled in sufficient detail it is possible to detect some errors occurring in a system and in some cases find the cause of those errors (i.e. faults).*

Upon detection of an error, the system will attempt to ascertain where the fault lies. Then a new solution will be created if such an alternative is feasible within the system.

To emphasis the original content, the following are concepts, deliverables, theories or approaches which, to the author's knowledge, did not exist before this thesis:

- The concept of using a DSL for component specification to achieve automatic program generation, automatic error detection, automatic fault prediction and reconfiguration.
- The automatic error-detection theory and approach taken.
- The automated strategies and examples for fault-prediction algorithms for a component combination driven system.
- Automatic error detection and fault prediction based on DSL specification without necessitating additional user interaction.
- The importance of selecting a component combination technique that is explicitly traceable, predictable, avoids recursion and can, assuming sufficient components, produce alternative solutions.
- The concept of a DSL as illustrated in this chapter.

5. Implementation

The aim of this chapter is to describe some of the methods and technologies applied in implementing the architecture as described in the last chapter. In particular, this chapter describes the implementation of the Solution Generation phase, from finding potential matches to generating an executable version. The overall prototype was named Hydra, after the mythical beast which could regenerate when a head was severed.

5.1 Smart Components?

One design decision made quite early in this research was that each part of the system should be as simple as possible. This was of particular importance when considering how the components would work together. There were two obvious alternatives:

- Components which provided functionality only.
- "Smart components" which configured themselves.

It was evident that "smart components" and software agents shared a great deal of common ground, e.g. some degree of autonomy in decision making, self-configuration, decentralised control structures. It was decided that "smart components" were unsuited to this application as a strict control hierarchy and centralised decision making allowed the architecture as a whole and the individual components to deal with their own respective tasks without additional complexity.

5.2 Architecture

The architecture has already been outlined in the last chapter. The run-time interactions are described in Figure 5:1:

ORIGINAL IN COLOUR

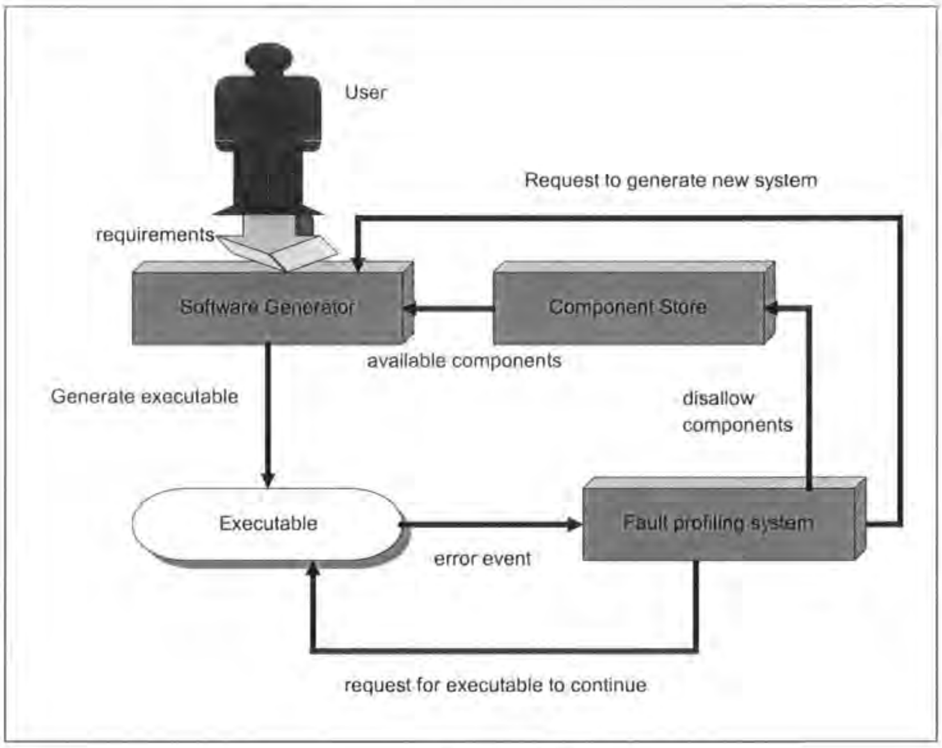


Figure 5:1

The inter-communication between the elements in the run-time architecture has been implemented using CORBA (Common Object Request Broker Architecture). CORBA was selected because it allows distributed objects implemented in a number of supported languages (such as Java™ and C++) to be interconnected with little extra client-side complexity. This is an attractive proposition if one of the stages becomes resource-intensive as it could be placed on a separate server. The intrinsic parts of the executable, i.e. the interconnected components, are also inter-connected using CORBA. This is important as one useful form of redundancy is to have backup hardware which provides similar services to be used in the case of a sub-system failing. This also allows vendors to provide services on their own machines rather than having to distribute executables, of particular use for applications with specific hardware requirements. To summarise, there is a great deal of potential for reuse using CORBA as a means of locating and inter-connecting components. It should be noted that other technologies exist which also implement similar capabilities, specifically Microsoft's DCOM and Sun's RMI. Comparisons between the three technologies are available in Harkey and Orfali in [98] and direct comparisons between CORBA and DCOM are available in Szyperski [99].

5.3 Software generator

As previously stated in the last chapter, Figure 5:2 shows the stages in generating the executable.

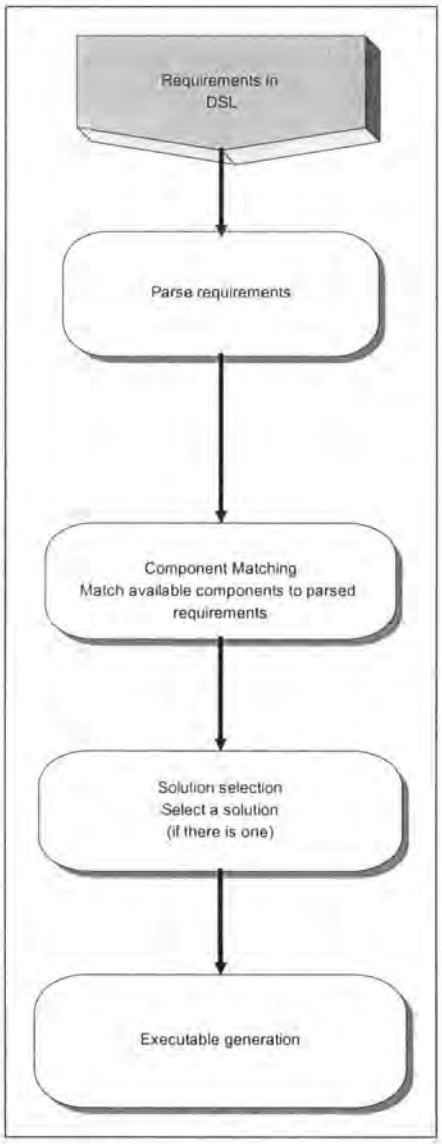


Figure 5:2

5.3.1 Requirements Entry

The requirement entry process was achieved by entering data using a rudimentary GUI implemented in Java, of which screen-shots are shown in chapter 6.

ORIGINAL IN COLOUR

5.3.2 Parsing requirements

The method of parsing the requirements is not conceptually original and hence has not been outlined here. The input is a DSL-requirement and the output is a tree of the DSL constructs as shown in Figure 5:3, also previously stated in chapter 4.

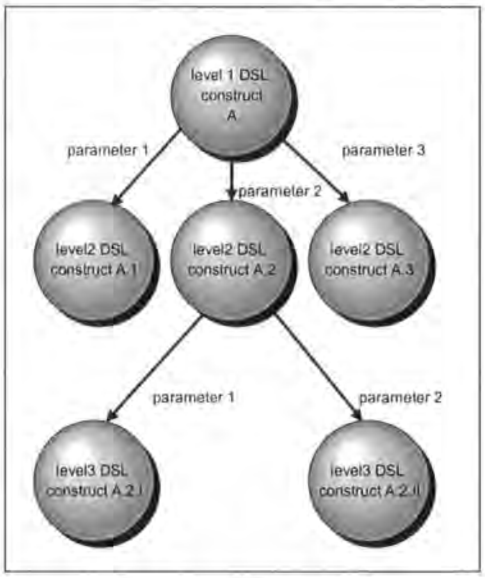


Figure 5:3

5.3.3 Component Matching and Solution Selection

The component matching algorithm was implemented in a Java implementation of CLIPS, called JESS (Java Expert System Shell [100]). This system was easily extendable, as new rules and facts could be added and removed in CLIPS (a fact based programming language) with ease, and also new basic operations in CLIPS could be defined in Java and added to JESS.

The pattern matching algorithm is relatively simple. The tree of data is split into a list of facts in CLIPS, each fact containing four parts, as shown in Figure 5:4. The component rules[9], which are held in the component store as shown in Figure 5:1, are then added to the rule base. A solution is reached when the DSL-construct which is the base node in the tree (i.e. Level 1 - construct A (Figure 5:3) is matched by a component and has no further dependencies unfulfilled. A match signifies that a DSL-construct can be implemented by a component. In this system, only DSL-

[9] Only rules for component which are not disallowed are introduced into the system

construct-to-component mappings are permitted. Upon each match, a new fact is added to the fact-base. The fact contains the following data (see Figure 5:4)

1. Construct number
2. Description of Solution so far, in terms of components
3. Rules applied so far
4. Any dependencies still not fulfilled

Figure 5:4

The construct number is required for knowing where in the tree this current fact belongs. The second field is used to represent the solution as it is created. Without this field, the system would be able to ascertain whether a solution is possible but would not know what the solution was. Field 3 was introduced in order for extra fault-prediction information to be used. This provides a representation of exactly which rules were applied and in what order they occurred. However, it should be stated that no fault-prediction algorithms have been implemented which use this information. The final field is used to see whether this component has been implemented fully, i.e. all of the items upon which it depends have been matched to components.

The scheme for component combination used in this prototype is that after one or more DSL-constructs has been matched to a component A, it can be combined with other components (e.g. B, C,..) that implement constructs upon which component A depends, if all of the components upon which A depends no longer have dependencies of their own. That is to say, the fourth field of each component upon which A depends must be empty for component A and the dependent components (B, C,..) to be combined. This algorithm implements the bottom-up matching of constructs. It should be noted that because there are only a finite number of component rules and once each rule has matched, it cannot match again to the old fact (due to the way CLIPS deals with facts "firing" rules) or continue to match on the new fact (because only DSL-construct-to-component match rules are allowed), the solution generation system will always terminate. There are two ways in which the system can terminate:

1. There are no more rules to match and a solution has not yet been found.

2. A solution has been found.

At this stage, the prototype deviated from the original concept, because, as stated in chapter 4, all solutions should be found and then one selected. In the prototype, the first solution found is selected. This was done because the prototype had no method of distinction between solutions therefore it makes sense to stop at the first one. If no solution can be found then it may be necessary to re-allow components which are suspected of containing faults.

5.3.4 Executable generation

The final executable is generated by creating an intermediate Java and CORBA implementation, from the Software generator, and compiling the source code into byte-code, to be executed on the Java Virtual Machine.

The scheme for code generation is very simple. For every component in the solution, there will be a method call. The base method call will call the next level, which will call the next level and so on. Therefore this scheme is unsuitable for systems which are likely to have a very high number of constructs because it consumes a large amount of stack space. Inside each method, the generated code will contain the error-detection code (pre- and post-conditions) which is generated by transforming the high-level behavioural description (in DSL-test) into Java. Extra data is encoded into each method, such as the purpose of the executable (i.e. the DSL requirement), the structure of the executable (i.e. the order and name of the components), and how the solution was formulated (i.e. the rules applied to generate this solution).

Upon an error-event, Java Exceptions are used to provide a dynamic trace of what has executed, and where the error has occurred in the current component (i.e. pre-conditions or post-conditions). The outer-most calling method (i.e. the method which is called first) contains code which connects to the Fault-Profiling System (FPS), informs it of the error and then exits.

5.4 Component Store

The component store maintains a list of all the components available in the current system. Figure 5:5 shows the data stored about each component in the component store:

1. Component name
2. Component rules
3. Connection Code
4. Actual execution code
5. DSL construct name
6. WBT code (if any)

Figure 5:5

Therefore, in order to introduce a new component to the system, it simply needs the relevant data to be entered in the component store. The component store plays many similar roles to an Object Request Broker (ORB), which is part of CORBA™. However, the Component Store was introduced due to certain ORB services not being available in the CORBA implementation used. For large systems, there would typically be more than one Component Store, and a request for a component could be forwarded to other component servers. A similar scheme exists for ORBs in current CORBA standards.

5.5 Fault Profiling System

This was implemented in Java™ and CORBA™. A number of fault-profiling strategies were applied to this problem, which are detailed in chapter 7.

5.6 Summary

This chapter has provided a brief overview of the implementation of the system, the prototype implementation of which was named Hydra. The majority of the details concerning the underlying architecture have been, where possible, abstracted upon. The conceptual description of these stages appears in chapter 4. However, the strategy for generating new solutions and creating code has been described briefly in order to demonstrate the methods applied and to emphasise the compositional rather than transformational approach taken in this thesis. Examples of the system executing are described in chapter 6 and the data and techniques applied are discussed and critically evaluated in chapter 7.

6. Case studies

6.1 Introduction

This chapter describes a number of sample uses of the Hydra architecture which was developed to verify the ideas proposed in chapter 4. The main aim of this chapter is to highlight the experimental prototypes in sufficient detail in order that the issues can be described in chapter 7.

The components which are used in these systems are artificial in that they were not previously existing code. This means for the purpose of actual scientific discussion, the components were all designed for reuse (DfR) rather than with reuse (DwR). An implication of this is that further investigation is necessary before claims about DwR components can be made.

During the course of the research, three main prototypes were studied. The first one, dealing with sorting mechanisms, never reached implementation. The reasons for this are explained more fully in chapter 7. This first attempt is documented here for a number of reasons

- The failure was very instructive in the learning process of how to apply the DSL.
- To illustrate the limitations which have been found.
- To avoid repetition of this category of mistake.

The second prototype system generates executables using components without any persistent data. That is to say the executable would comprise of stateless components. Persistent components were seen to be the more difficult case and hence dealt with in the next prototype.

A third and final prototype was developed to verify the hypothesis' would work with components with state and components without state. It is based on a cost-accounting domain. This domain was chosen because of its need for persistent data.

This section now describes the three domains. For the sake of brevity, the first two domains are only described in sufficient detail to illustrate the difficulties involved. These are discussed in chapter 7. The third domain is covered in more detail, in order to give the reader an idea of what the DSL systems are like.

6.2 The sorting domain

As previously stated this domain was never implemented. The original concept was to develop a DSL which would describe the sort domain so that a list of data could be sorted. The type of data and the sorting algorithm used would be described by a DSL. The only factor to be considered in this prototype was

- Maintaining type integrity.

The main consideration was that comparisons in the sort routines would be performed by the correct operations for the types of data being sorted.

Many different formations of languages were investigated. The main problem observed was that for the DSL to be expressive enough to describe the problem would require the constructs to be at the same level or smaller than the target components would potentially be. This was aggravated by the fact that each component would have to be so fine grained that it could be implemented in some languages using less that 10 lines of code. Many of the DSL language structures were found to be so near to certain choices of implementation language (such as C++ with Standard Template Library) that the DSL was deemed not worth implementing. As previously stated the underlying decision of what or when to reuse should be based on economy. This was obviously not an efficient usage of resources.

6.3 The mathematical domain

This domain was chosen in order to implement a simple system where the components would not require persistent state. Mathematics seemed an ideal domain.

A DSL was designed with the following considerations, in addition to those described in chapter 4.

- The requirement should be described in as few constructs as possible. This will shorten the time to express the requirement and probably reduce the chance of user-error.
- The DSL should contain as small a number of domain constructs as possible. This means that the DSL-user will have less constructs to remember.

The system would have each DSL construct encoded with what was sufficiently accurate as an answer. In order to do this each construct had an approximation defined in its behavioural specification. However, as stated in chapter 4, the approximation would not be permitted to be too strong. This is a quite different approach to the self-testing mathematics by Rubinfeld [63].

For this scheme, inaccuracy is a common form of error. This is another example of a component behaving in an undesirable manner. The inconsistency occurred because the overall design contained the assumption that a certain degree of accuracy was acceptable. As illustrated in this example these assumptions may not be made by the component itself (i.e. the inaccuracies can be due to the representation of the underlying architecture). In the presence of an unacceptable result the system generated alternatives which in turn self-tested to see if they complied.

One weakness found was that the pattern-matching algorithm could not express "this construct can be implemented using the following combination of DSL constructs". The system could instead express "this construct can be implemented using the following combination of components".

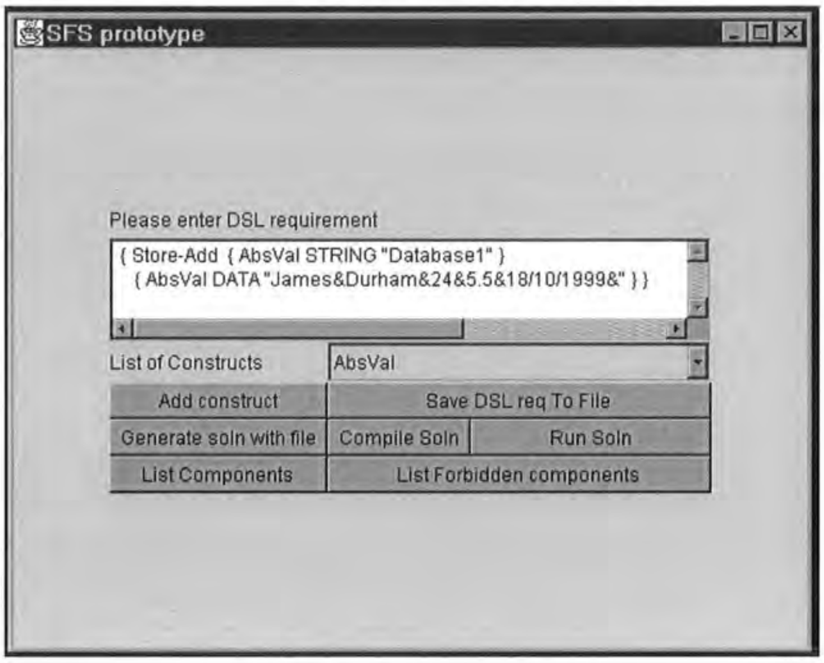
6.4 The accounting domain

This domain has been covered in most detail because it embodies the general theme of the thesis. The problem domain is vertical not horizontal, the components contain persistent data and they require white box tests in addition to the standard interface tests. However, the description in this section will only cover the general use of the system, highlighting the results attained from the system in order to discuss them in later chapters. This section does not provide a user manual, or even a general notion of how the implementation can be used. It is the author's advice that the reader should concentrate on the general issues raised rather than how to use the system. In fact, it should be emphasised that a real implementation system should have a more usable GUI and may have a completely different method of data entry. As this system was an evaluation prototype, the usability aspect was not the focus. A more detailed description of the DSL developed for this domain is described in Appendix A.

This implementation performed well above expectations. For example, inconsistencies between the DSL construct and the component were located in a small number of test-runs. The user could add, remove, query and manipulate data in ways which should be familiar to a user who has experience with accountancy (according to data taken from Weygandt et al. [101])

The system, built upon the Hydra architecture, interacts using a GUI as shown in (Figure 6:1), which is used to help guide the user with entering requirements in the DSL.

ORIGINAL IN COLOUR



```
{ Store-Add { AbsVal STRING "Database1" }
  { AbsVal DATA "James&Durham&24&5.5&18/10/1999&" } }
```

Figure 6:1

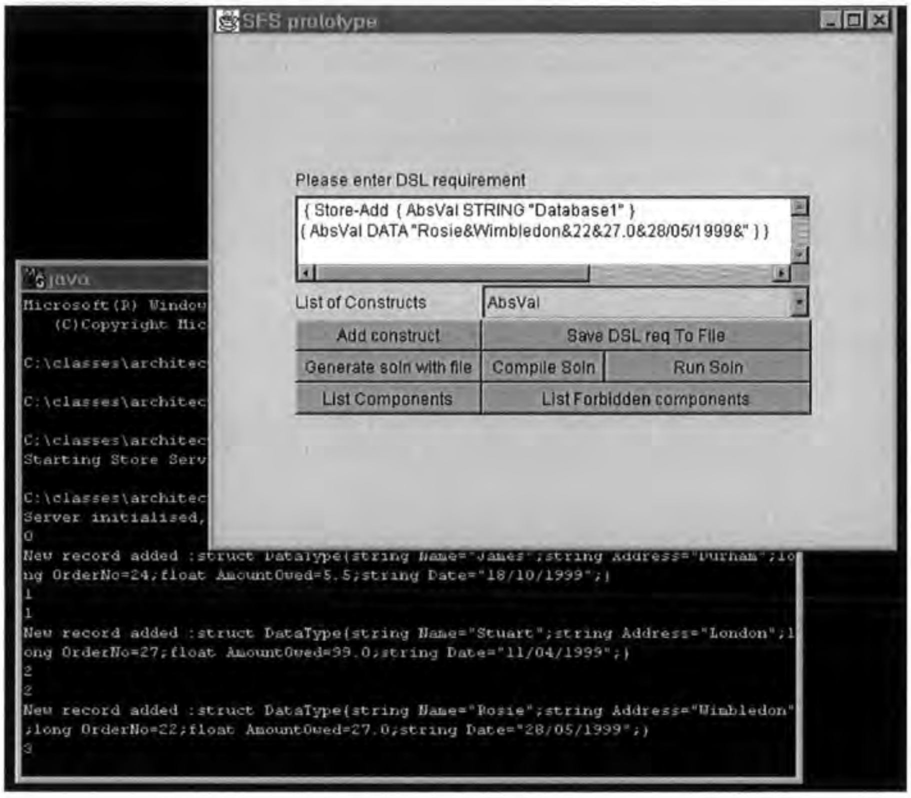
In Figure 6:1, the user has signalled the desire to add a record to the database called "Database1". The record is of type **Data** and is entered via the user interface. The second line of the DSL requirement is a string representation of the record. The user does not typically deal with the raw textual representation but instead manipulates the data through the GUI.

The data in the record to be added is of the form described in Figure 6:2:

Field	**Value**
Name	James
Address	Durham
OrderNo	24
AmountOwed	5.5
Date	18/10/1999

Figure 6:2

This entry is entered and the system attempts to locate components which fulfil this requirement. In this example there is only one component which can implement the DSL construct ("Add"). In this example JAVA code can be generated and executed successfully with no errors. The system automatically verifies that after the operation is completed there is one more record than before it was executed.



```
{ Store-Add { AbsVal STRING "Database1" }
{ AbsVal DATA "Rosie&Wimbledon&22&27.0&28/05/1999&" } }
```

AbsVal

```
Microsoft(R) Windou
   (C)Copyright Mic

C:\classes\architec

C:\classes\architec

C:\classes\architec
Starting Store Serv

C:\classes\architec
Server initialised,
0
New record added :struct DataType(string Name="James";string Address="Durham";lo
ng OrderNo=24;float AmountOwed=5.5;string Date="18/10/1999";)
1
1
New record added :struct DataType(string Name="Stuart";string Address="London";l
ong OrderNo=27;float AmountOwed=99.0;string Date="11/04/1999";)
2
2
New record added :struct DataType(string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";)
3
```

Figure 6:3

This operation is repeated two more times, successfully with records containing the data described in Figure 6:4.

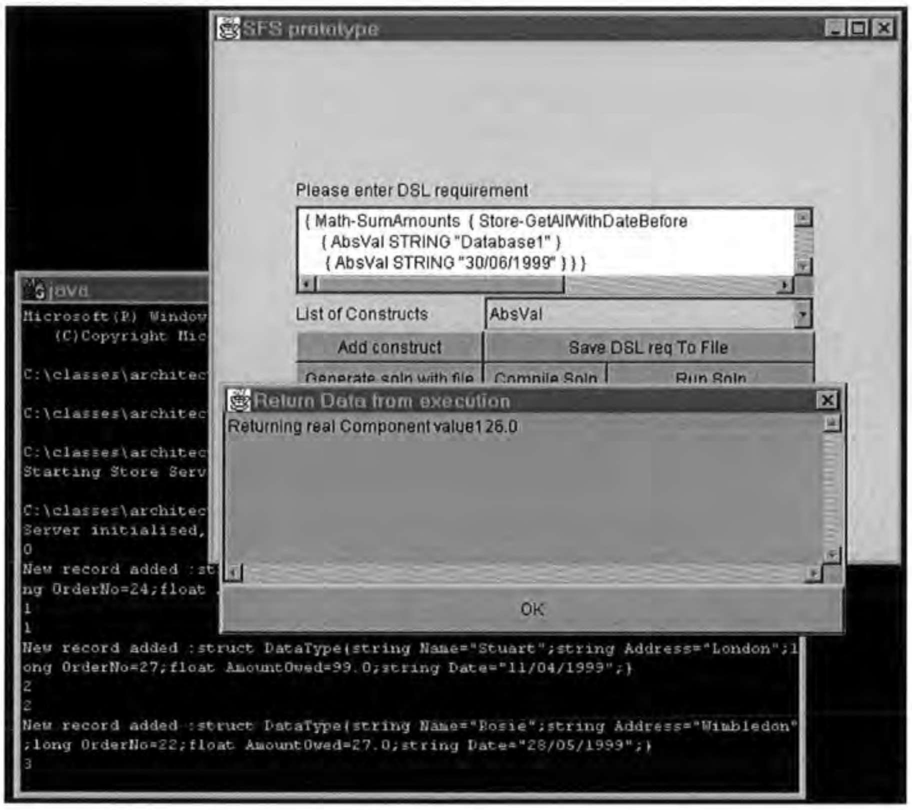
Field	**Value of record 2**	**Value of record 3**
Name	Stuart	Rosie
Address	London	Wimbledon
OrderNo	27	22
AmountOwed	99.0	27.0
Date	18/10/1999	28/05/1999

Figure 6:4

The black DOS window behind the GUI window in Figure 6:3 shows the output supplied by the component implementing the add operation. It displays a string representation of the Record, along with the abstract number of records in the component before and after the execution (i.e. 0 – no records, 1 – 1 record after the operation).

ORIGINAL IN COLOUR

Having populated the component with data the user wishes to query the database. They need to know the sum of the amounts owed in "Database1" where the date occurred before the 30th of July. The system generates a solution that uses two components, "SAB" which provides the implementation to the SumAmount construct and "Add" which implements the GetAllWithDataBefore construct.



```
{ Math-SumAmounts { Store-GetAllWithDateBefore
  { AbsVal STRING "Database1" }
   { AbsVal STRING "30/06/1999" } } }
```

```
Returning real Component value126.0
```

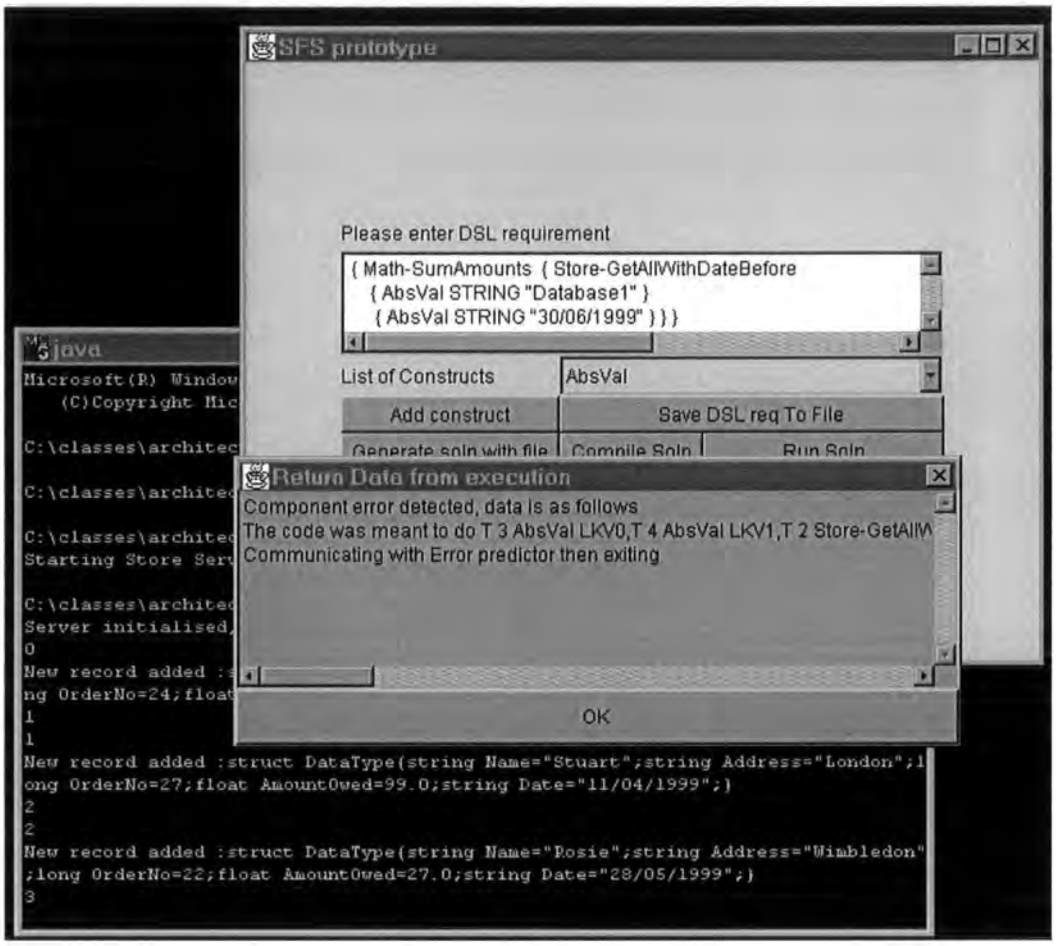
```
Microsoft(R) Window
   (C)Copyright Mic

C:\classes\architec

C:\classes\architec

C:\classes\architec
Starting Store Serv

C:\classes\architec
Server initialised,
0
New record added :st
ng OrderNo=24;float
1
1
New record added :struct DataType{string Name="Stuart";string Address="London";l
ong OrderNo=27;float AmountOwed=99.0;string Date="11/04/1999";}
2
2
New record added :struct DataType{string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";}
3
```

Figure 6:5

As shown in Figure 6:5, the correct answer (126) is produced. However it should be noted that the behaviour of the component implementing the "SumAmounts" construct is not well defined. An incorrect answer could also have been generated and the system may not have detected the error.

Figure 6:6 illustrates the case that same operation is performed later but this time the "SAB" component has failed and is not currently operating.

ORIGINAL IN COLOUR



Please enter DSL requirement

```
{ Math-SumAmounts { Store-GetAllWithDateBefore
   { AbsVal STRING "Database1" }
    { AbsVal STRING "30/06/1999" } } }
```

```
Component error detected, data is as follows
The code was meant to do T 3 AbsVal LKV0,T 4 AbsVal LKV1,T 2 Store-GetAllW
Communicating with Error predictor then exiting
```

```
Microsoft(R) Windov
   (C)Copyright Mic

C:\classes\architec

C:\classes\architec

C:\classes\architec
Starting Store Serv

C:\classes\architec
Server initialised,
0
New record added :s
ng OrderNo=24;float
1
1
New record added :struct DataType{string Name="Stuart";string Address="London";l
ong OrderNo=27;float AmountOwed=99.0;string Date="11/04/1999";}
2
2
New record added :struct DataType{string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";}
3
```

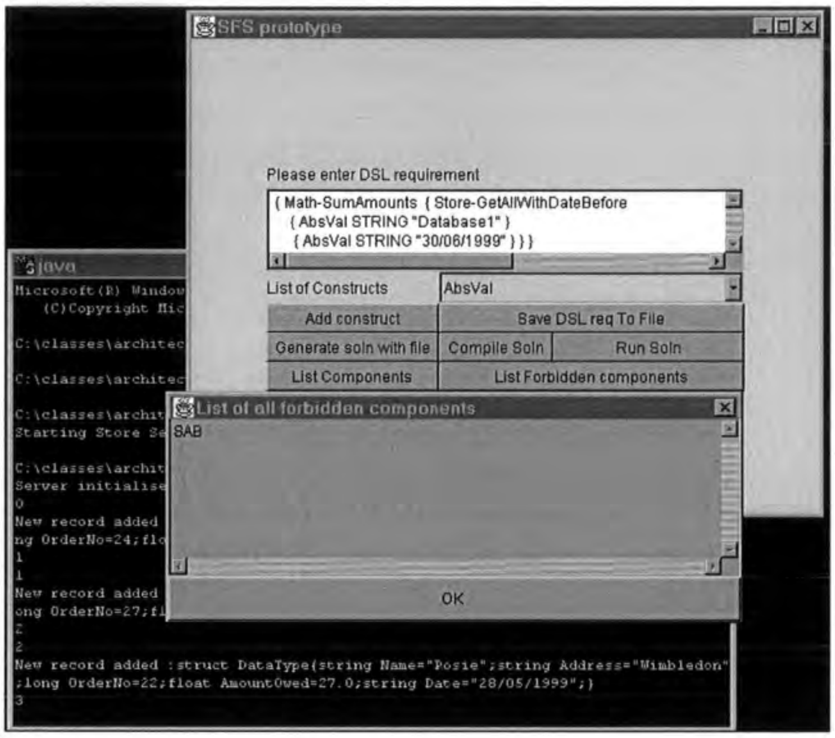
Figure 6:6

The executable detects that an error has occurred and reports the error to the Fault Prediction system. It passes any information about the error that has been detected along with the structural information about the program which has been encoded at Code Generation stage (as described in detail in chapter 4).

The fault-prediction stage then examines the structure and attempts to predict where the fault lies. The algorithm used in this example is to take all the components that have already executed in this example. In this example, a random mechanism is used to decide which component is to blame but the component where the error was detected is three times more likely to be chosen than any other component (described as Algorithm 1 in chapter 7).

In this case the "SAB" component is blamed and forbidden for use in future systems, as shown in the GUI in Figure 6:7.

ORIGINAL IN COLOUR



```
{ Math-SumAmounts { Store-GetAllWithDateBefore
  { AbsVal STRING "Database1" }
   { AbsVal STRING "30/06/1999" } } }
```

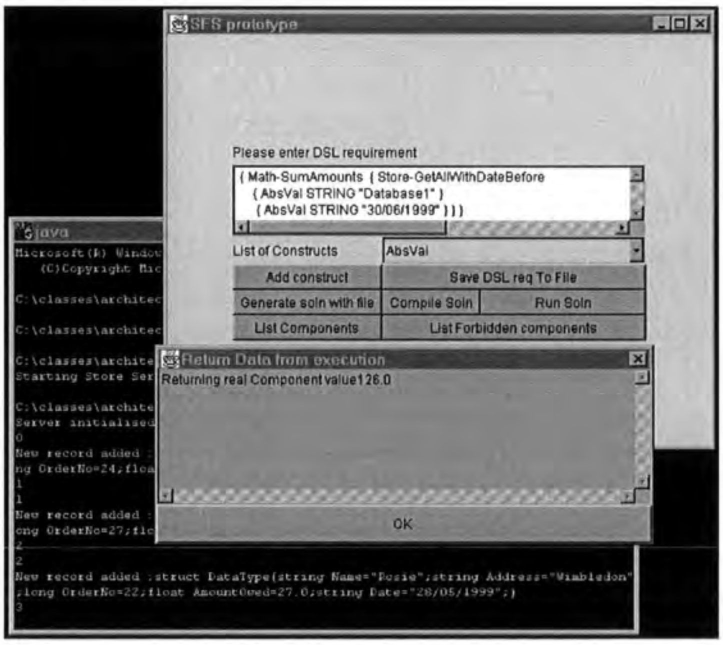
List of Constructs: AbsVal

SAB

```
Microsoft(R) Window
   (C)Copyright Mic

C:\classes\architec

C:\classes\architec

C:\classes\archit
Starting Store Se

C:\classes\archit
Server initialise
0
New record added
ng OrderNo=24;flo
1
1
New record added
ong OrderNo=27;fl
2
2
New record added :struct DataType(string Name="Posie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";)
3
```

Figure 6:7

The system then generates an alternative solution, this time using a second component which implements the SumAmounts construct. The system has recovered from one of the components breaking by using existing redundancy, as shown in Figure 6:8.



```
{ Math-SumAmounts { Store-GetAllWithDateBefore
  { AbsVal STRING "Database1" }
   { AbsVal STRING "30/06/1999" } } }
```

List of Constructs: AbsVal

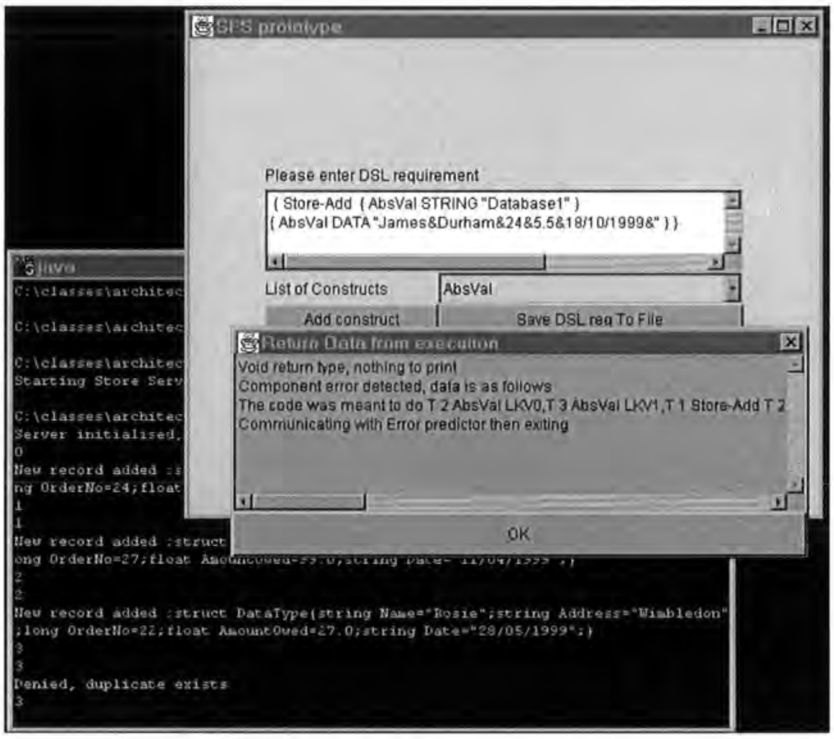
Returning real Component value126.0

```
Microsoft(R) Window
   (C)Copyright Mic

C:\classes\architec

C:\classes\architec

C:\classes\archite
Starting Store Ser

C:\classes\archite
Server initialised
0
New record added :
ng OrderNo=24;floa
1
1
New record added :
ong OrderNo=27;flo
2
2
New record added :struct DataType(string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";)
3
```

Figure 6:8

ORIGINAL IN COLOUR

As stated in chapter 4 the system requires a way of expressing incorrect input. If there is no concept of user input error without component error, problematic side-effects can occur. For example the database has a constraint that the order number must be unique. However it is possible to enter a record with the same order number. As shown in Figure 6:9, the actual component does not allow this to happen. This creates an inconsistency with respect to the abstract behaviour of the construct because the number of records has stayed the same. This is detected by the executable and reported to the Fault-Prediction System. This breaks the underlying philosophy in chapter 4 that an error should only be detected when a fault is present.



```
( Store-Add ( AbsVal STRING "Database1" )
( AbsVal DATA "James&Durham&24&5.5&18/10/1999&" ) )
```

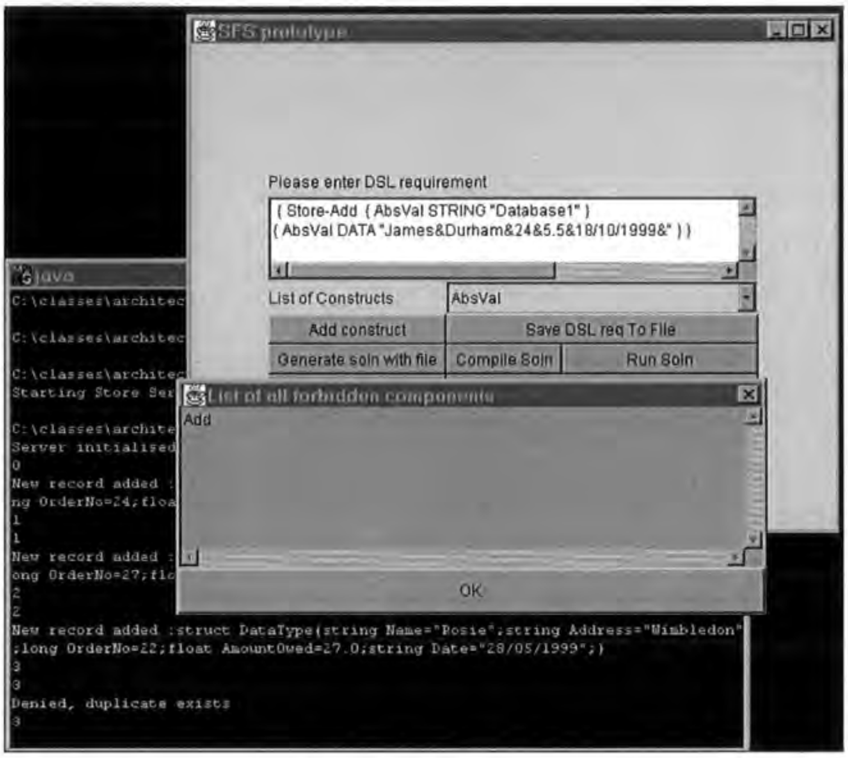
```
Void return type, nothing to print
Component error detected, data is as follows
The code was meant to do T 2 AbsVal LKV0,T 3 AbsVal LKV1 ,T 1 Store-Add T 2
Communicating with Error predictor then exiting
```

```
C:\classes\architec

C:\classes\architec

C:\classes\architec
Starting Store Serv

C:\classes\architec
Server initialised,
0
New record added :s
ng OrderNo=24;float
1
1
New record added :struct
ong OrderNo=27;float Amo
2
2
New record added :struct DataType(string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";)
3
3
Denied, duplicate exists
3
```

Figure 6:9

This case is serious because the component that will be blamed also contains the system data as shown in Figure 6:10. This illustrates another problem with persistent components, that of how to recover the state to a component which is to be eliminated from the system. It is evidently not sufficient to simply ignore the data which is contained in that component.

ORIGINAL IN COLOUR



Please enter DSL requirement

```
( Store-Add ( AbsVal STRING "Database1" )
( AbsVal DATA "James&Durham&24&5.5&18/10/1999&" ) )
```

List of Constructs: AbsVal

Add

```
C:\classes\architec

C:\classes\architec

C:\classes\architec
Starting Store Ser

C:\classes\archite
Server initialised
0
New record added :
ng OrderNo=24;floa
1
1
New record added :
ong OrderNo=27;flo
2
2
New record added :struct DataType(string Name="Rosie";string Address="Wimbledon"
;long OrderNo=22;float AmountOwed=27.0;string Date="28/05/1999";)
3
3
Denied, duplicate exists
3
```

Figure 6:10

6.5 Summary

This chapter has given a number of example implementations of the Hydra architecture which was produced during this research. The level of detail has been kept minimal to aid in the understanding of the process of using the system.

Three main domains were researched. The first example, that of the sort domain was deemed too fine grained to be worthwhile as the components were too small to realistically map to single DSL constructs. The second domain used only stateless components; these can be readily interchanged and are easier to test at interface level. This domain was a success but the architecture could not express one DSL construct being implemented in terms of other DSL constructs, only a DSL construct being implemented in terms of components. This was unsuitable for the mathematics domain where one construct could often be built from others. The third prototype system, which was in the accounting domain contained persistent and stateless components. This prototype was successful to a surprising extent. It highlighted inconsistencies between implementations and specifications even after a very small number of test runs.

Even though the third prototype was successful, a number of issues were found. The primary issue was that a scheme was needed by which the DSL could take into account user-error so that user-related errors such as incorrect input would not cause the system to disallow components. Another important issue was how to recover the state of a persistent component when it was at fault.

7. Evaluation

The purpose of this chapter is to evaluate the concept of applying a DSL based approach to component composition, error-detection and fault-prediction. The underlying concepts involved in this approach has been described in chapter 4. To enable evaluation of this concept some prototype architectures have been developed. These have been described in chapter 6.

The overall aim of this research is to discover whether the thesis proposed is feasible and to create an architecture and guidelines for developing systems. It would have been unrealistic and unwise to attempt to develop a comprehensive system without knowing what was to be made, how it was to be made or even if it should be done this way.

Although statistical profiling has been performed, it will not be used as the main form of discourse. This is because the components have all been designed with reuse by a single developer, for a restricted number of domains and in a restricted number of languages (as stated in chapter 4 these systems have been developed as initial illustrations of the concept proposed). Hence the issue of whether the data is representative could not be resolved without significant further investigation. Therefore the principal means of discourse for this evaluation is natural language.

A fair criticism can be made of the prototype systems, in that they are all unrealistically simple. Unfortunately this was unavoidable given the limited experimental conditions. There was a conscious decision that it was better to learn the way the hypothesis performed in restricted conditions than developing a realistically-scaled system and risking the findings because of external conditions. However this issue needs to be addressed before the approach can confidently be proposed as a realistic way of addressing reuse.

This chapter is in three parts. The first section evaluates the research in terms of the criteria which were considered upon commencing this research, as described in chapter 1. The next section describes the issues which were discovered experimentally, particularly those highlighted in chapter 6. The final section is a summary of this chapter.

7.1 Original evaluation criteria

The evaluation criteria, as appearing in chapter 1, are as follows:

1) The investment required to develop the proposed system in comparison to conventional system.
2) The return on the investment for the proposed system.
3) The types of error which can be detected by the proposed system.
4) The types of fault which the system can handle.
5) How the proposed system can evolve.
6) Feasibility of using this type of system

The aim of this section is to describe the approach in terms of these criteria.

7.1.1 Investment required

As stated in chapter 2, investment is a significant factor when creating a new system. The long-term aim of most software companies is to reduce the required investment in software while maintaining, or improving, quality. As with most methods of reducing long-term costs the approach proposed here requires increased short-term investment. The current methods of creating the DSL construct specification system are tailor-made and hence more resource intensive than is necessary. However, the architecture (Hydra, chapter 4) proposed in this thesis is generic for any DSL with a compatible grammar and therefore it should be possible to implement new DSLs without redeveloping the architecture, assuming the overall syntactic structure of the DSLs remain the same. Even ignoring the extra investment required due to developing and using new methods, specialising Hydra still requires significant additional effort in order to design the language, tailor the generic tools to implement the system, categorise the components in terms of the DSL and help familiarise users with the new methods. An implication of the investment required is that the system must be used a sufficient number of times to make this investment worthwhile. Other methods of categorising software are also susceptible to this problem, for example classification libraries as in chapter 2.

7.1.2 Return on investment

The automation of black-box component reuse is of great commercial interest for most medium to large scale companies, and all software developers. Although the user still needs to program in some form, albeit high-level, the DSL should make this task easier. It should be emphasised that the term "programming" is used here to describe the process of communicating the user-requirements to the system. There is no reason why this would have to be represented in text-form. Some research has already been done into visual programming languages (e.g. Korfhage [102]) but visual programming-environments are already commonly used, for example the modern spreadsheet. There is great scope for applying innovative user-interfaces in this field.

The foreseen return on investment for using the proposed methods and architecture is that the user has a RAD system which should allow them to describe their requirements in well-defined and familiar terminology. This allows the actual programming to be delayed to stages as late as the end-user, rather than being performed by the developer. As stated in Szyperski [3] this can give a considerable competitive advantage to companies.

Another aim of this research is that the return on investment also will manifest itself in increased confidence that the components are behaving as desired hence addressing, in part, the "not-invented here" syndrome. This may help to encourage the external development of software which, as stated in chapter 2, can be viewed as a method of sharing investment risk. Although substantial research remains before this can be claimed to be achieved for real-scale systems, the methods and architecture proposed in this thesis have been shown to achieve these aims for simpler domains.

7.1.3 Error detection capabilities

The ability of the system to detect errors if, and only if, there is a fault is critical to the success of this work. This is dependent upon the model of the system and the domain, acceptable non-functional requirements and the level of error-detection detail desirable. The dependencies of the error-detection model upon the system are described in chapter 4.

When devising error-detecting systems, there is an important issue of non-functional requirements, one of compromise. The system may require an operation to be performed within 'x' seconds or using 'y' units of main storage. Introducing extremely detailed inter-component tests may unnecessarily degrade system performance. Even systems with less demanding performance requirements cannot satisfactorily use arbitrarily large quantities of resources for no observed gain. Conversely there is a restricted category of tests which can be performed while holding to non-functional requirements, an implication of which is that the system will not be able to detect errors optimally.

Two issues requiring consideration for the level of detail in the error-detection model are the conflict of abstraction with verification and the expressiveness of the DSL-test language.

7.1.3.1 Abstraction and verification

The issue of abstraction and verification requires serious consideration when designing a new architecture and associated DSL. A very abstract description at DSL construct level, i.e. one which is tailored to describing generic traits rather than being tailored to specific implementations, will maximise the potential for reusing existing components at the expense of ensuring that they are behaving correctly. An implementation-driven system can maximise the test detail in order to ensure that the operation was performed correctly at the expense of not being able to map to as many components. In chapter 4, the concept of the abstract state of a component was proposed which partially addresses this issue. Each component which implements the abstract concept denoted by the DSL construct must, by definition, have sufficient data to provide the abstract state of a component. That means that the tests which are introduced must be implemented for all instantiations of the DSL construct (i.e. components).

7.1.3.2 Expressiveness of testing language

The last issue, that of the expressiveness of DSL test, was highlighted particularly well in the third prototype(Accountancy Domain). The native types of the system were originally as shown in Figure 7:1

String
Bool
Int
Real

Figure 7:1

They had the following operations defined for them (chapter 4), where any inappropriate operations were marked as undefined (see Figure 7:2)

Equal
NotEqual
Greater
Less
GTOEqual
LTOEqual

Figure 7:2

They were, at design time, satisfactory decisions[10] for the tasks at hand. Figure 7:3 shows the two extra types that were introduced:

Data
List

Figure 7:3

As stated in appendix A, *Data* is a composite record of a number of other elements, which reflects the information which will be stored in the accountancy database. *List* is a representation of a sequence of *Data* records. These types were added without extending the operations.

[10] In retrospect String should have had a "length of string" operation and a "contains" operation.

The operations of the types in DSL–test were originally defined generically in order to reduce the overhead of learning the language. Unfortunately the language became ambiguous and confusing. Therefore, the generic operations approach was discarded in favour of the type-dependent operations. To illustrate this point consider the type *List*, which represents a list of *Data* entities. None of the operations outlined are appropriate for this type. For example, does Equal mean that every element is the same and in the same order or that every element is the same and in any order? The GreaterThan/LessThan comparisons have more ambiguity. Does GreaterThan, when applied to two lists, describe whether there are more elements in the list or whether the totals of the amounts owed is greater? Although the issue of DSL-test construct semantics is of concern to the DSL designers and component developers, confusion will inevitably lead to incorrect implementations and frustration. This issue can be addressed by defining each type's operations independently.

7.1.3.3 Violation of principle 1

It was demonstrated in chapter 6 that it is possible to break the error-detection principle 1.

Principle 1 - The system should only detect errors when faults are present

A scenario that illustrates the difficulties involved is that of adding a record to a database. Records in the database should not contain duplicate order numbers. The behavioural description modelled the add operation in terms of the abstract concept of the number of records in the database. Specifically, there was a check that there was one more record in the database after the operation than before the operation. However this operation fails in terms of the description, without any fault being present. If the user enters a record with an existing order number, the component(s) implementing the DSL construct may refuse to add the new record. Therefore the number of records remains constant and an error is detected. This directly contravenes principle 1 of the error detection philosophy because the user of the system was at fault, rather than the implementing components.

The divergence of behaviour between model and implementation was caused by a weakness in the model of the system. In order to guarantee that an 'Add' operation did increase the number of elements in the database, it would be necessary to check that the data entered was not a duplicate. The cause of this problem was inability to describe whether an element was already in a database. This problem was actually caused by the poor expressiveness of DSL-test.

7.1.3.4 Undoing operations

The issue of how to undo operations is important for considering any component system which contains state. For example, if one part of a requirement (e.g. a requirement which is implemented by multiple components) is executed and then an error is detected, it is important the system should be returned to the state before the execution the requirement. Rollback, as used in the field of databases, is a method of achieving this. The data is not permanently changed until a commit, that is until the operation is declared by all participating entities to be complete. There are a number of potential methods of implementing this. One method is to make all persistent components (i.e. components which map to constructs containing abstract state) implement an undo mechanism which is called when required. However, there is a high likelihood that faults could be present in the undo mechanisms. An alternative method is for the architecture to implement the undo mechanism by copying persistent components before the operation is executed. This could potentially be performed by the architecture at a binary level. As this is a conceptually simpler and more reliable method of providing rollback functionality it seems more appropriate. Some other relevant methods and techniques are used in Transaction Processors and in CORBA (Orfali et al. [96]).

7.1.4 Fault-prediction capabilities

As previously stated, upon detection of an error the system should attempt to find the source of the problem. Unfortunately, current software methods often allow errors to propagate. For example, a missing file name in a user interface may cause an error in a database which verifies the data before storing. Therefore it is insufficient to rely on the fault being present in the system where the error occurred. In fact, there is a complex relationship between faults and errors. If the error always occurred in the faulty component, i.e. the behavioural specification was a perfect representation of the system, then fault-prediction would be trivial.

7.1.4.1 Fault-prediction strategies

Chapter 4 outlined a number of fault-prediction strategies. These are listed below. These strategies can be applied in two ways. The first is to only consider components which have executed , or were executing, at time of error-detection and the second is to consider all components present in the system.

The algorithms described were as follows[11]:

1. Select any one component but make the component in which the error was detected more statistically likely.
2. Select any one component but make any components which have been suspected of faults before more statistically likely to be chosen.
3. Select any one component which has been used and remove its use from the system.
4. Utilise metrics on the components to help decide which component is at fault. This requires the underlying system to maintain statistics on the number of times a components has been suspected.
5. Do nothing - The fault may be intermittent.
6. Remove the component in which the error was detected.
7. Select multiple components with one of these schemes

Investigation was undertaken into each of these algorithms. These can be separated into 2 categories:

- Dynamic algorithms
- Static algorithms

Dynamic algorithms can create different results from execution to execution from the same data. This may be due to a random element in the algorithm or because the algorithm learns and hence adapts over time. Algorithms 1-4 are dynamic

[11] These have been re-ordered, but are the same algorithms.

algorithms. This research investigated algorithms 1-3 in detail, but not 4 due to lack of time.

While dynamic algorithms may provide a different answer using the same data, static algorithms always give the same answer. The static algorithms proposed were 5 and 6. They are highly reliant on the actual fault being what was expected. For example, if the error did not occur in the component where it was detected, then the algorithm will have no probability of getting the correct answer. The advantage of the static algorithms proposed in this thesis is that they are very easy to implement, quick to execute and easy to understand.

Algorithm 7 was not investigated because there was significant complexity for a single component prediction without considering multiple components. This has been proposed as a line of further work.

7.1.4.2 Dynamic algorithm profiling

As previously stated, for the purposes of this research, the first 3 algorithms have been statistically profiled:

1. All components are equally likely to be at fault, apart from the one where the error was detected. The component where the error was detected is more likely to be blamed.
2. Every time a component is selected for blame, the chance that it will be reselected next time is increased. The system can provide feedback to inform the fault-prediction system that the executable did not work.
3. Select any component at random (with equal probability).

It is acknowledged that the behaviour of Algorithm 3, due to its simplicity, is not of interest and has only been explicitly considered in this section as a means of comparing the success of Algorithms 1 and 2.

The behaviours of both strategies 1 and 2 can be altered by changing the emphasis, or "weight" as it is referred to here, for the components. For example, in strategy 1 the weighting of the strategy (the amount by which the component where the error was detected can be made to be more likely to be selected than other components) can be varied. If either algorithm is applied with "weight" 0, then it will result in an equivalent probability distribution as algorithm 3.

Strategies 1 and 2 are particularly interesting because of their diversity of approach. Strategy 1 is based on the assumption that the error-detection model is, in general, strong enough to detect an error in the component where it occurs and Strategy 2 is based on the assumption that a system can provide reliable feedback, which can be used to learn which component is to blame. In addition to the two strategies, a separate issue was raised, that of whether to reduce the component search space. As previously stated, there were two alternatives:

(a) Only consider components which have executed before or are executing when the error is detected.

(b) Consider all components present in the executable.

Reducing the search space, as by using (a), relies on the error occurring during execution of, or after execution of that part of the system (as illustrated in Figure 7:4). Although certain types of fault can escape this form of fault-prediction, for example hidden inter-component interference, it allows the use of the meta-information encoded into the generated executable. The executable provides a dynamic program slice upon detection of an error, which is used to help provide this information.

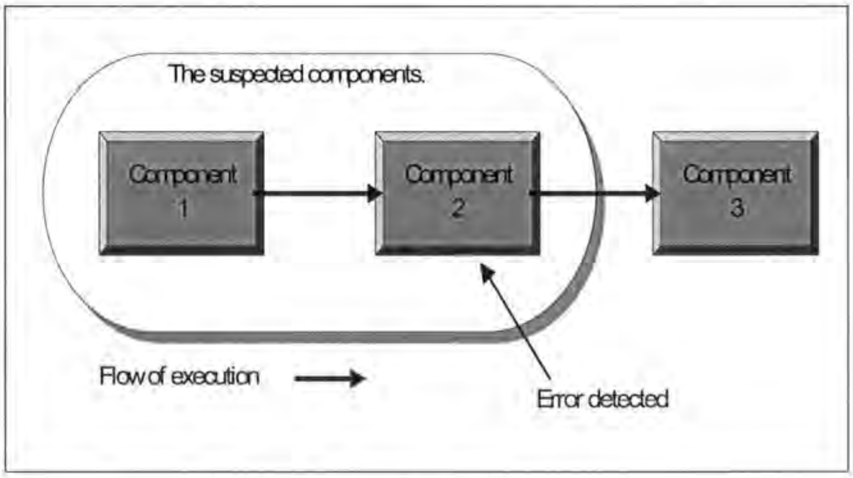


Figure 7:4

7.1.4.3 Test cases

The algorithms were both evaluated using eight test cases:

Test case	Number of components	Error detected at	Component at fault	Feedback
1	4	4	4	No
2	4	4	2	No
3	20	10	10	No
4	20	10	4	No
5	20	10	10	Yes
6	20	10	4	Yes
7	20	10	15	No
8	20	10	15	Yes

Cases (1) and (2) were selected to test how a system would function with a small number of components. Test case (1) illustrates how many times the system is correct when the fault lies in the component where the error was detected, and Test case (2) tests how the system will react when the fault is not present in the component where the fault was detected. The purpose of test cases (3) and (4) is to cover the same cases as (1) and (2) but for an executable with a larger number of components. Test cases (5) and (6) are the same as (3) and (4) but the system provides feedback, that is to say if the prediction is incorrect, the algorithm is informed of the fact. Test case (7) is designed to illustrate how the system will react if the error is caused by a component which has not been executed yet and test case (8) is the same as (7) but with feedback.

7.1.4.4 Algorithm 1

If x is the component where the fault was located, probability of x being blamed P(x) is equal to:

$$P(x) = (w+1) / (n+w)$$

Where w= weight, and n= number of components being considered.

ORIGINAL IN COLOUR

If x is not the component where the fault was located, probability of x being blamed P(x) is equal to:

P(x) = 1 / (n+w)

Figure 7:5 illustrates how algorithm 1 performed after 1,000,000 test runs when pruning has occurred and using the pruning strategy(a). It is evident that when the algorithm is correct, as in Test cases (1), (3) and (5)[12], satisfactory results can be achieved and the more weight added to the algorithm, the better the results in comparison to Algorithm 3 (or weight 0) . When the algorithm is incorrect, i.e. the component where the error was detected is not the component containing the fault, as per test cases (2), (4) and (6), the results deteriorate with weight. The algorithm also becomes consistently less effective with more components. Test cases (7) and (8) give no correct answers as only components which have been executed at time of error-detection are considered, when using the pruning strategy(a).

Note that Algorithm 3 is equivalent in behaviour to either Algorithm 1 or 2 with a "weight" of 0, and has only been included as a basis for comparison.

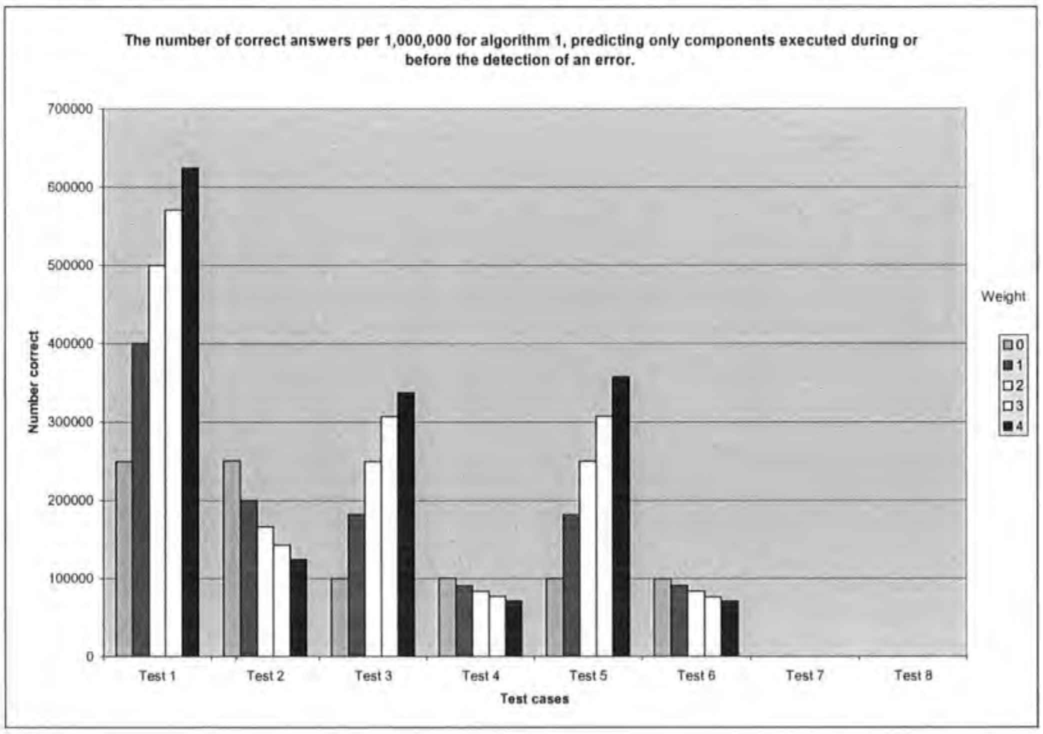


Figure 7:5

[12] Algorithm 1 does not use previous data to learn. Therefore the pairs of tests (3) and (5); (4) and (6); and (7) and (8) are effectively the same tests.

ORIGINAL IN COLOUR

The same algorithm has been applied whilst considering any component which is a part of the executable as a potential suspect, as shown in Figure 7:6. In an executable with a small number of components, as in (1) and (2), the results are very similar. However, Algorithm 1 does not perform well when applied to an executable with a larger number of components for tests (3), (4), (5) and (6) as the same algorithm with pruning (using strategy (a)). In contrast, test cases (7) and (8) are improved, as shown in Figure 7:6.

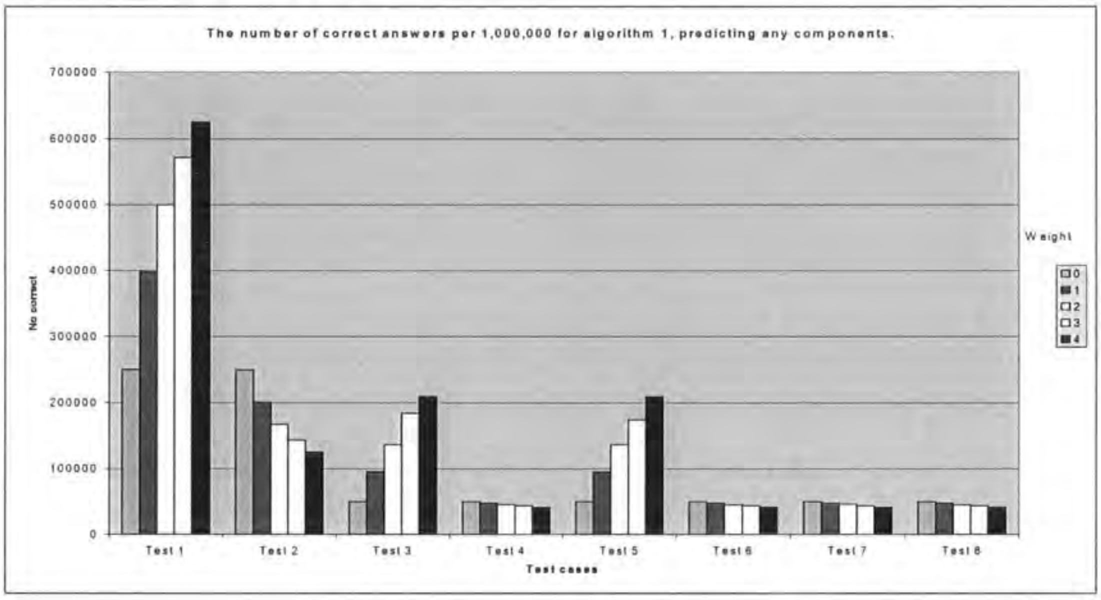


Figure 7:6

These results suggest that Algorithm 1 is applicable for fault-prediction in executables with a small number of components and a detailed error-detection model.

7.1.4.5 Algorithm 2

The probability of a component 'x' being selected, 'P(x)' is defined as:

$$P(x) = (1 + yw) / (n+tw)$$

where component 'x' has been selected for blame 'y' times previously, 'n' is the number of components considered in the system, 't' is the number of times the algorithm has been executed and 'w' is the algorithm weighting.

The same tests (1-8) were applied to Algorithm 2 for 1,000,000 iterations, whilst only considering components which had been executed. Test cases (1) to (4) were

ORIGINAL IN COLOUR

found to generate random results. This is because the algorithm is not supplied feedback. Test cases (7) and (8) both result in no correct answers because components which have not been executed when the error occurs are not considered.

The results are shown for a in Figure 7:7.

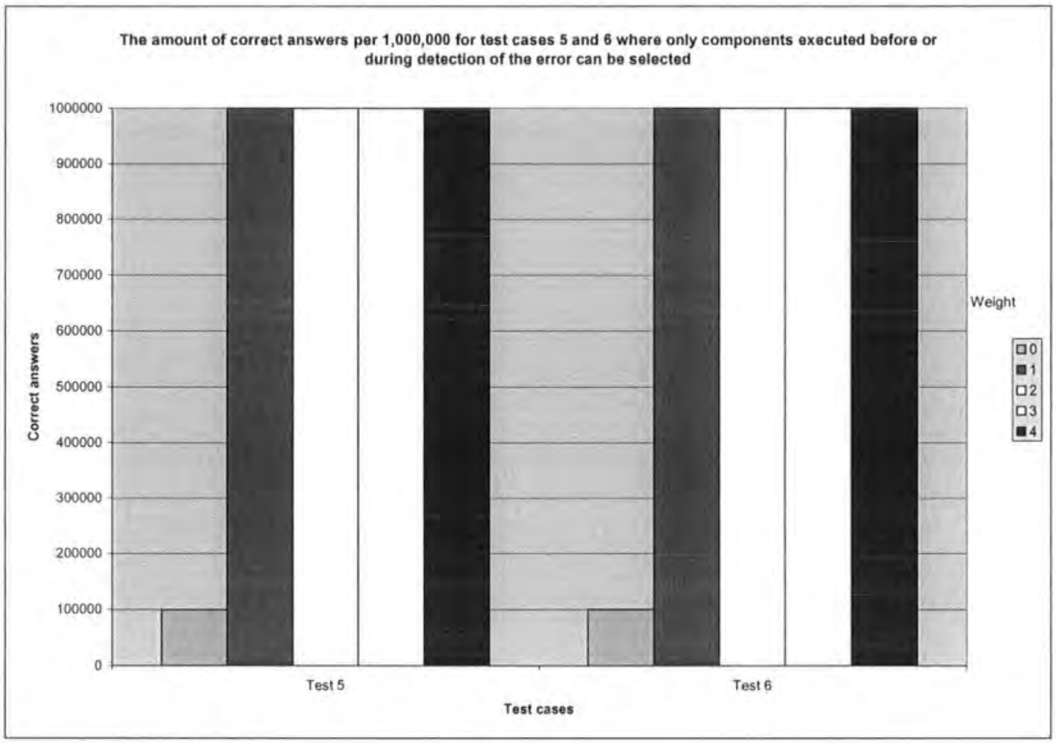


Figure 7:7

The results for the same experiment without the application of pruning strategy (a) are shown in Figure 7:8.

ORIGINAL IN COLOUR

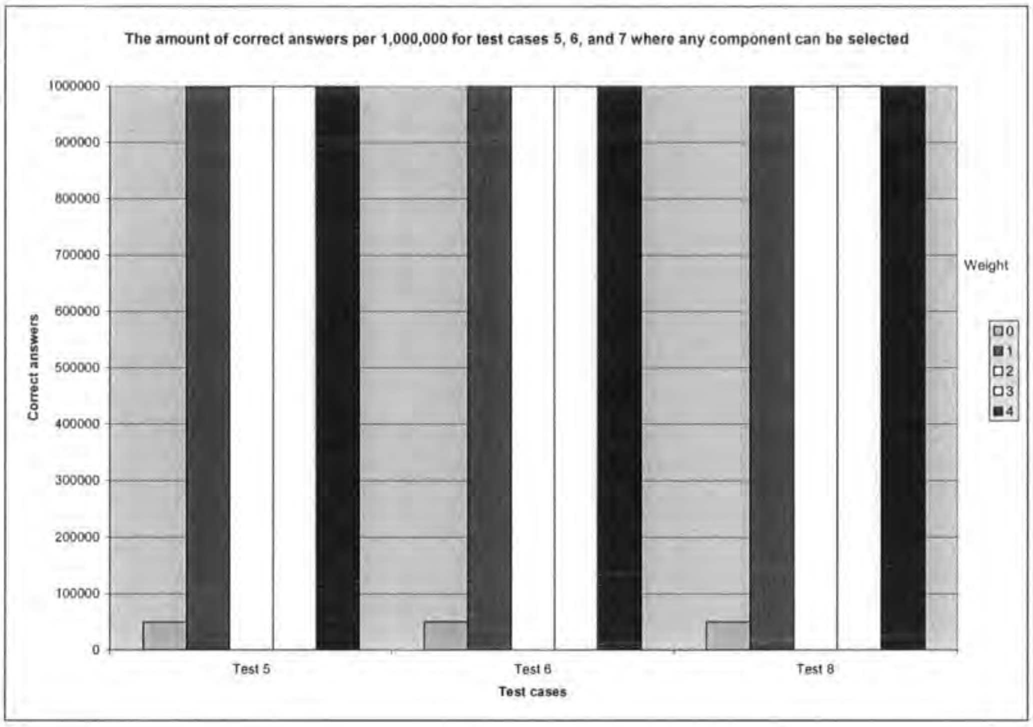


Figure 7:8

As can be seen, the results are very near to perfect for all variants of the algorithm, independent of the weight applied (assuming weight is >=1). There are three reasons why this is so. The first is that the feedback is 100% accurate. In practice this is not likely. Experimentation has been performed on less than perfect feedback but no conclusions can be drawn from these due to complex relationships. A second reason for the extreme accuracy in the predictions is that the profiling system had to consider one type of experiment only. In real systems, data from different examples may have been applied so trends may not be as obvious as documented here. The third reason is that the experiment was run 1,000,000 times which provided the algorithm with plenty of learning time. Therefore Algorithm 2 was also profiled to see how it performed with a smaller number of learning opportunities. Figure 7:9 shows data from test case 5, indicating how the algorithm learns over time, while only considering components which had executed before the error was detected. Figure 7:10 shows the data for the same test case while considering all components. Test case 6 has been omitted due to the similarity of results between 6 and 5.

ORIGINAL IN COLOUR

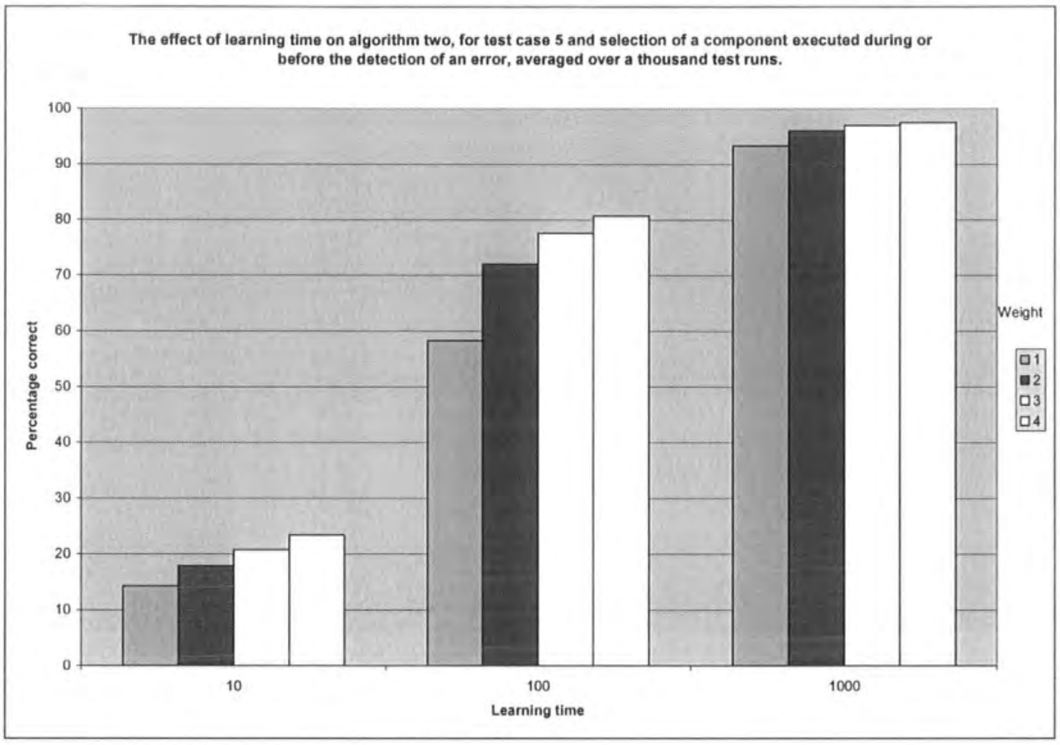


Figure 7:9

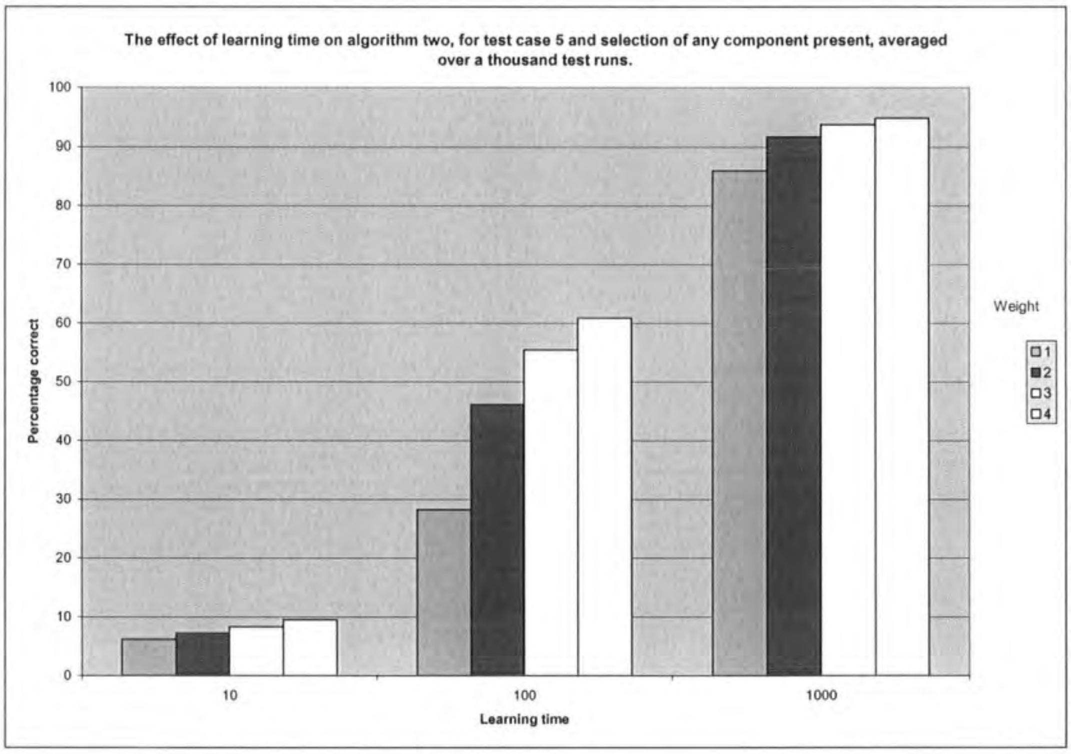


Figure 7:10

This data suggests that Algorithm 2 is applicable for systems where accurate feedback is possible, and where DSL requirements are likely to be called more than once. Some difficulty remains as to how similar two DSL requirements have to be before the data from one of them is relevant to the other. In this example the cases are all identical. In practice, the assumption that there will be sufficient identical cases may be too strong.

To re-iterate, the aim of this section was to illustrate two methods of fault prediction, and to contrast them with each other and to a purely random selection method. This has been done in order to demonstrate their use and show their strengths and weaknesses. This data cannot be used "as is" to see which algorithm is the best on average. This would require a detailed study into which cases of use, as modelled by the test-cases, are more typical.

7.1.4.6 Further Applications of Fault profiling

As previously described, information on the dynamic behaviour of a system[13] should be stored and can be used when making future fault predictions. This profile may include, for example, measures of reliability including the number of times the component has been present in an executable where an error has occurred and what the composition of the executable was at that point. The profiling data could also be used to provide feedback to component developers. It was found experimentally that, upon presentation of test-data demonstrating strange behaviour, a component developer is often able to correct the fault relatively easily. An example of such an occurrence is a data-base not storing certain characters correctly. Therefore such a system could support the efforts of the component developers as they could be periodically supplied with the profile data.

7.1.4.7 Fault-prediction problems

In addition to the difficulties already shown in fault prediction, this method has the implicit side-effect that sometimes components may be blamed for malfunctioning when they are not responsible. This statement can also be extended to include components which behave acceptably for the majority of the time but, in very specific conditions, malfunction. It therefore seems undesirable to ban components from a system completely. This leads to the issue of component re-introduction. As with fault-prediction, there are a number of different cases for re-introducing components to a system. For example if the system is re-configured and the error reoccurs in the next executable, then that component may be re-introduced because it was not necessarily at fault[14]. As none of these were implemented there is no further discussion on component re-introduction.

7.1.4.8 Non-component faults.

There is an additional possibility, for any system which has a separation between implementation and specification, that some of the components will be associated with an inconsistent abstract behaviour description (i.e. a component can be mapped to the wrong construct, in the wrong way or will not function in the target environment). A component which never detects it is at fault is acceptable, as stated in chapter 4, error-detection principle 2, that:

"The system may not, in certain cases, detect an error when it occurs."

It is also possible, and in fact probable, for a component's White-Box Test (WBT) operations to be incorrectly implemented and generate errors when none are present. However since the component and its description are, in this system, inseparable this is not of particular concern. A component causing a false error is, in itself a fault and hence does not contravene principle 1 of the error detection.

"The system should only detect errors when faults are present"

The current system encodes the rules which were applied to create the executable into the executable itself, in order to potentially identify rules which contained faults. However, no fault-prediction algorithms have been developed which utilise this data.

This thesis and all the subsequent underlying research, is based on the assumption that there is a flawless implementation of the underlying architecture and types. In practice this is unlikely. In order to address this issue, the architecture has been kept as simple as possible and utilises only well-understood methodologies and

[13] Particularly which components have been banned because they are suspected of containing faults and whether the fault continued afterwards
[14] Fault-prediction where only one component can be blamed at a time may use a different mechanism to achieve this for example the architecture may "forget" that a component is banned after a certain amount of time.

tools. However, it must be acknowledged that a typical architecture and associated types and operations would contain faults which the system could not correct.

7.1.5 Evolution

It is important to consider how a new system can change with time. As the implemented architecture and associated DSL is based on an underlying domain model, the domain model can be used to help describe the sorts of changes possible.

The changes identified were as follows:

1. A new construct needs to be added.
2. An existing construct needs to be re-described.
3. The domain model needs to be changed significantly.
4. An underlying architectural assumption must be changed.

In the case of adding a new construct, the architecture allows new constructs to be added. The accountancy prototype will allow this to be performed run-time.

Care should be taken when considering altering an existing construct. If the changes make usage of the altered construct more restrictive than the original then existing requirements could be incompatible, creating a legacy problem. The same will occur if the actual function of the construct changes. If the construct is to change in either of these ways then either creating an explicitly new version of the language or using a different construct name should be considered as courses of action.

Significant changes to the domain model are problematic in that they could involve changes in a number of constructs. A sensible approach is to create a new version of the language. This category of changes may also be addressed by the approach of developing the DSLs in hierarchies (as shown in Figure 7:11), an idea originally proposed for generic DSLs in Jarvis [103]. A lower level DSL could be developed during the domain analysis process. Then different DSLs, focussing on different users needs, could be developed in terms of the lower level constructs. This method should be more resistant to this type of change, assuming the concepts identified at lower levels do not need to change in a way that would make the hierarchy inconsistent.

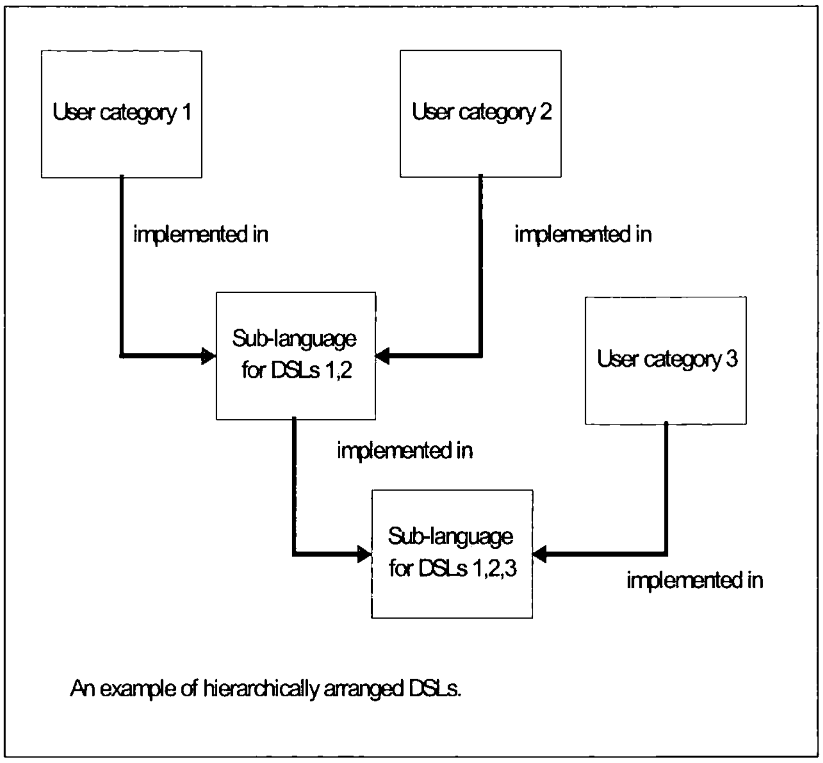


Figure 7:11

Similar mechanisms often exist in hierarchical programming methods. They often have the top level as user-oriented and the lower levels as more implementation specific data. The top levels typically call the lower level functionality.

The final category of change considered for the DSL system is that the target environment changes so that the architectural assumptions are significantly different to the previous system. This category can be potentially require a large quantity of changes. If the components are only documented in terms of the original architecture, i.e. they work in the original architecture, then each component must be re-evaluated individually to see if it will work in the new environment. If components belong to a number of collections, each with its own architectural assumptions, then some of the components may not need to be re-evaluated. For example, an architecture underlying a particular DSL system has a certain collection of architectural support facilities (in this case automated tidy-up and a certain structure of event handling). If these support facilities conditions are too weak (or incompatible) then components with more demanding (or conflicting) architectural requirements will not be applicable. Even worse, it may not be clear which components will be appropriate.

7.1.6 Feasibility of using this type of system

One of the aims of this research was to evaluate whether it is feasible to use the proposed architecture and methods in real world systems. Although it is not possible to claim that real-world systems have been described here, the results from the case studies are very encouraging. Therefore, although it would not be realistic to claim the approach outlined in this thesis will definitely address real-world systems, a less strong claim can be made. There has been no reason found why this approach would not work, within the limitations described, in real-world systems. That is to say the overall method of applying a DSL and supporting architecture was found to be an effective method of communicating information about the domain. After the prototypes were developed, new requirements could be used to generate code with no additional human interaction, assuming sufficient components exist. However, considerable additional investment in effort was required in order to develop the domain model, write the components and provide data for the program generator to use. Therefore this approach would only be feasible for systems with a high degree of reuse. This restricts the approach to domains which are stable. It may also be utilised if a domain model is created during the software development process.

The process of self-reconfiguration was found to be successful if the error-detection guidelines were followed. If the error-detection model is weak then the number of components suspected was found to generally significantly greater than the number suspected using a more precisely specified construct. Two main types of fault prediction models were investigated and their behaviour evaluated in a number of cases. It was found that both performed well under certain conditions.

7.2 Additional issues

During the course of the research other issues were raised which needed further consideration. The aim of this section is to describe these issues in detail, explain their significance and to suggest methods of improvement on the original solutions.

ORIGINAL IN COLOUR

7.2.1 Method of combination

The current prototype uses the following model of software generation, as shown in Figure 7:12.

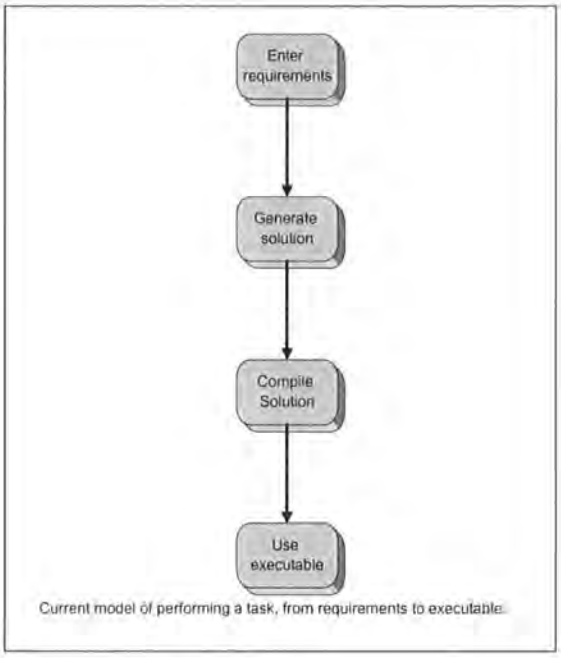

Enter requirements
Generate solution
Compile Solution
Use executable
Current model of performing a task, from requirements to executable.

Figure 7:12

Although a compiler approach was sufficient, in terms of evaluating the hypothesis, it is not appropriate for use in an actual DSL system where different user requirements could be submitted frequently. For this case a requirements interpreter would be more appropriate. Background execution could periodically generate partial solutions so that the pattern matching phase could be made more efficient.

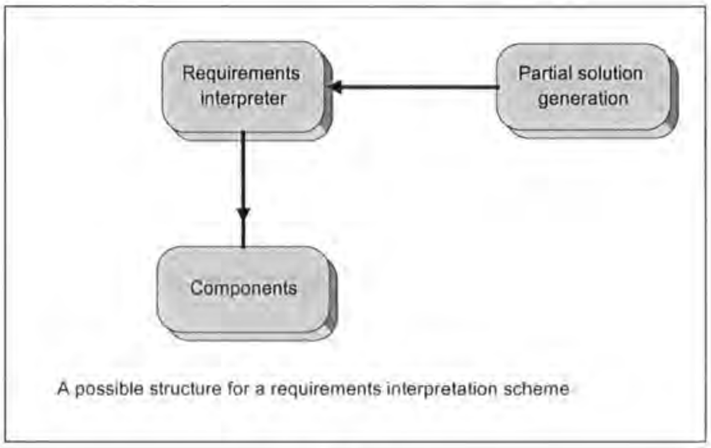

Requirements interpreter
Partial solution generation
Components
A possible structure for a requirements interpretation scheme

Figure 7:13

Although the lifecycle of requirements to executable would be required to change for a new system, an equivalent interpreter system, i.e. Figure 7:13, should be easily achievable, and many of the findings in this thesis are still applicable to such a system.

7.2.2 Introduction of transformations

As previously stated in chapter 5, it is useful for certain domains to be able to describe a DSL-construct in terms of other DSL-constructs. The current proposed architecture does not support this. The reason for the absence of this category of transformation is due to the problem of direct and indirect recursion of these rules as shown in Figure 7:14.

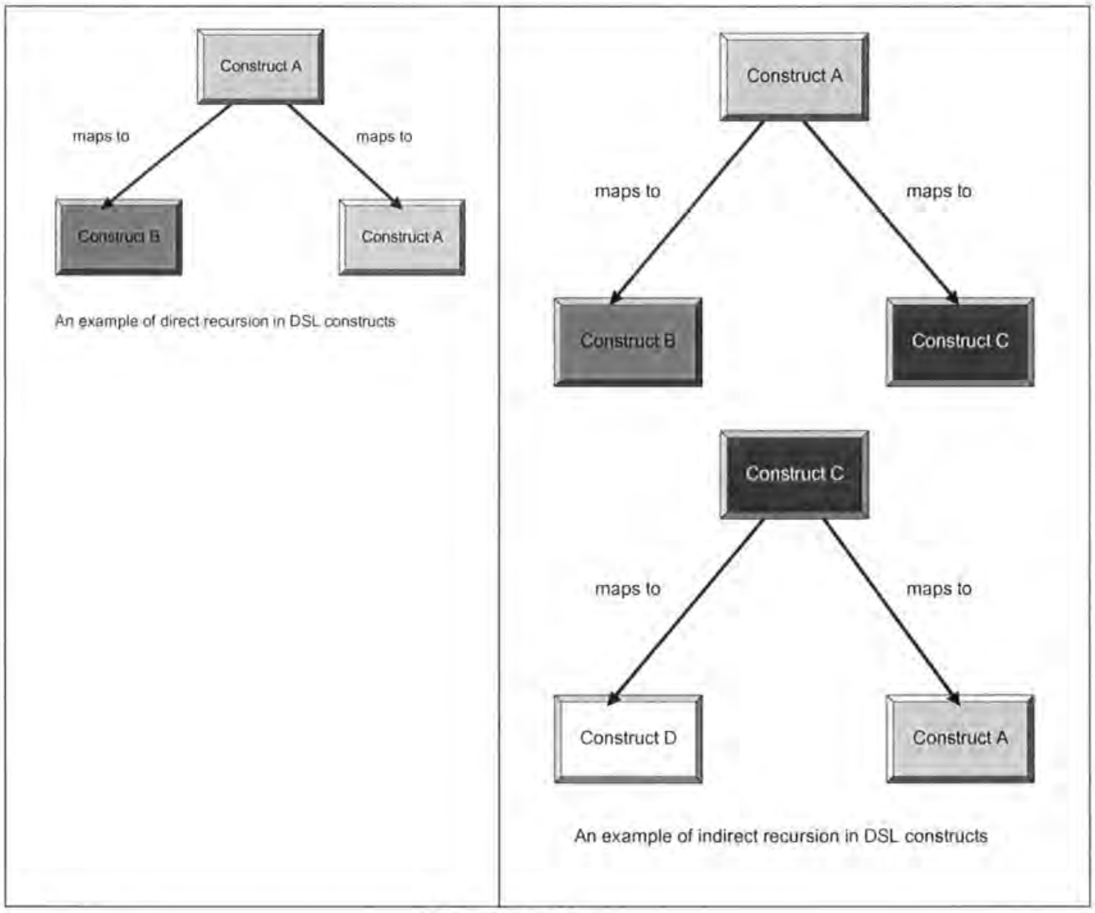


Figure 7:14

The existing system does not require human intervention to find a solution because, due to DSL constructs only mapping to components, the rules can only match a finite number of times. If DSL-constructs are permitted to refer to other DSL-constructs then infinite recursion becomes possible. This contravenes one of the aims of trying to reduce human interaction, because human interaction would be necessitated in order to prevent infinite recursion.

A hybrid scheme could permit a DSL-construct (A) to map to other DSL-constructs (B,C) if there are no routes (direct or indirect) from B or C back to A. In this compromise, the ability to describe arbitrary DSL-construct to DSL-construct mappings has been sacrificed in order to remove the need for human intervention at solution generation phase.

7.2.3 Granularity

The issue of granularity of components and DSL-constructs was raised, as outlined in chapter 6, and particularly well illustrated by prototype 1.

To recap, a DSL for a sort domain was investigated and found to be an inefficient solution for this domain. One of the reasons for this was the granularity of the constructs. The sort domain could be described in a variety of abstractions but these constructs were of similar granularity to actual programming constructs in generic programming languages. As this research is concerned with mapping DSL constructs to components, not lines of code, it was evident that this was unsatisfactory. That is to say, the domain's constructs should ideally be significantly larger than can be implemented by a few lines of code. If the constructs all map to very small components then a different composition technique would also be required.

It should be emphasised that the payoff for a DSL system which maps to very fine-grained components would typically be less than for a larger grained system, in terms of effort saved in composition. Therefore the economic feasibility of this approach is less strong for a very-fine grained component system than for a coarser grained one.

7.2.4 Type of domain

A mistake which was made in the first two domains (sort and mathematical) approached was that, in order to make the systems as generic as possible, the domains selected could be viewed as more horizontal (or cross-application) domains[15] than domain-specific. That is to say the sort domain and mathematical domains do not reflect an application family. They are focussed on forming solutions for a very-wide range of problems. This mistake was corrected in the accountancy domain. The payoff for implementing vertical rather than horizontal domains is threefold:

- The DSL can help document the problem domain.
- There is typically a larger payoff for vertical domain reuse partly because generic programming languages have not implemented as much of the necessary functionality as for horizontal domains.
- The underlying architectural constraints can be more restrictive and hence reduce the need for exhaustive component testing and documentation.

It is therefore suggested that any further research should mainly focus on vertical domains.

7.2.5 DSL systems (DwR or DfR)

There are a number of ways of developing a domain model. However, for a DSL system which maps constructs to components, there is an important issue of populating the component repositories. The two extreme approaches are :

(1) Design the domain model entirely around existing components, so there is at least one instance of each construct implemented at the start (analogous to DwR – chapter 2),

(2) Design the domain model completely independently of existing components then populate the component repository (analogous to DfR– chapter 2).

In practice, a compromise between these two extremes would be preferable, but this presupposes the existence of relevant components. Similar techniques have been described for frameworks, as described in chapter 2.

[15] It is acknowledged that there is sometimes no clear distinction between a horizontal and vertical domain.

7.3 Summary

This chapter has evaluated the methods and architecture in terms of the issues described in chapter 1 and in terms of issues whose importance became evident during the course of this research.

In evaluating the research with respect to the criteria identified in chapter 1, it was found that the proposed methods required additional short-term investment of effort. However, the preliminary findings suggest that if a system is used frequently enough to repay the initial investment then the methods provided an effective method of rapid application development, allowing the user to define their requirements in well-defined and familiar terminology. It also allows some of the programming to be delayed to later stages in development. The preliminary findings also suggested that there was increased confidence that the components were behaving as desired hence addressing, in part, the not-invented here syndrome. The aim of this was to encourage the external development of software which, as stated in chapter 2, can be viewed as a method of sharing investment risk. Although substantial research remains before these aims can be claimed to be achieved for real-scale systems, the methods and architecture proposed in this thesis have been shown to achieve these aims for simpler domains and no reason was found why this approach would not be appropriate for real systems.

The process of self-reconfiguration was found to be successful if the error-detection guidelines were followed. If the error-detection model was too weak then the number of components suspected was found, on average, to be higher than the number suspected using a more precisely specified construct. Two main types of fault prediction models were investigated and their behaviour evaluated in a number of cases. It was found that both performed well under different circumstances.

The methods and architecture were also evaluated in terms of other issues which were raised during the course of this research. It was found that the method of execution, which is currently based on a compile-then-execute cycle, was needlessly inefficient and an alternative, interpreter based, method of implementation was proposed. The current system's inability to define DSL construct implementations in terms of other DSL constructs was outlined as a weakness and a scheme for introducing construct-to-construct mappings was introduced. This was found to be a non-trivial task as the ability to describe arbitrary mappings between constructs made infinite recursion a possibility and hence necessitating human intervention at the solution generation stage. As this was contrary to one of the aims of this research (to minimise human intervention), construct-to-construct mappings are only permitted in a limited form, i.e. no direct or in-direct recursion is permitted. The issue of DSL construct granularity was identified as being problematic. Specifically, it was found that if the granularity was too fine then the pay-off for using the DSL-system was too small. Another important issue was the type of domain to be implemented. It was found that two of the three domains were actually horizontal domains and as such had a tendency to duplicate common general purpose programming language functionality. This was found to be unsatisfactory. The final issue outlined was that of how to actually instigate a DSL-to-component system (as described in this thesis), whether to develop the language and then populate the component repositories or to develop the language in order to reuse existing components. A compromise between the two approaches seemed most appropriate.

8. Conclusion

This thesis has described the process of taking a Domain-Specific Language based approach to component composition error-detection, and fault prediction. This has taken the form of an introduction chapter, two chapters of literature review, a chapter discussing the concepts proposed in this thesis, a chapter describing the implementation issues, a chapter describing three case studies which illustrate the ideas proposed here, and an evaluation chapter which discusses relevant issues. The aim of this chapter is to summarise the thesis, in terms of an overview of each chapter, and in terms of the research which has been undertaken in this thesis. The final section of this chapter describes areas that require further research.

8.1 Summary of research

Initially, a number of modern software reuse issues were researched. Originally, software reuse was considered without preconceptions of organisation or structure, and therefore the term **software artefact** was used to refer to the entity to be reused. The different categories of reuse were found to be:

- Types of code reuse (e.g. Black box, White box)
- The domain of code to be reused (e.g. Horizontal, Vertical domain)
- The origin of the code to be reused (DfR or DwR)

These issues were described in detail in chapter 2. It was found that for inter-company software reuse, only black-box and grey-box reuse are feasible approaches. Reuse for software in vertical domains was found to have a larger potential saving for reuse than horizontal domains. Whether the code was designed for reuse (DfR)) or previously existed before the decision to reuse was made (design with reuse (DwR)) was discussed and it was found that DfR artefacts are more generally applicable, but more restricted in what implementation assumptions they are allowed to make than DwR artefacts.

A number of factors which need to be addressed in order for successful reuse were found. They were:

- The quality of the software.

- The behaviour of the software
- Hidden design assumptions in the software
- The code requiring changes before reuse
- The identification of relevant code

These factors were described in detail in chapter 2.

A number of existing methods of addressing software reuse were identified:

- Software Components
- Software Libraries
- Software Frameworks
- Software Architectures (and DSSAs)

These methods were discussed with respect to the factors outlined previously to see how they addressed the factors. Domain-Specific Software Architectures (DSSAs) were found to be a particularly effective method of providing black-box reuse.

Chapter 3 evaluated other methods of software development which addressed similar issues to software reuse approaches and defined terminology for existing testing methods. The testing section was introduced to provide common terminology with which to discuss issues to be discussed later in the thesis. The alternative approaches to software reuse were:

- Software agents
- Domain-Specific languages (DSLs)
- Automated programming

Each was defined and described in terms of the reuse factors outlined in chapter 2.

The research hypotheses, taking the form of a main hypothesis and sub-hypotheses, were discussed in Chapter 4. The main hypothesis was:

> *Given a sufficient number of components described in sufficient detail it is possible to automatically develop systems in a manner such that human intervention is minimalised.*

This hypothesis was extended in a number of ways. Primarily, DSLs were proposed as the component description mechanism

A DSL can be applied to specify the behaviour of components. The specification can be used for the purposes of locating components and verifying their behaviour matches what is expected.

Each DSL construct, i.e. each element in the language, would provide two important services. Primarily, it would be used as a task description mechanism. That is to say the DSL construct name would be chosen to define a role which the implementing component(s) would fulfil. In addition to this, the DSL construct would also have an abstract concept of how implementing components would behave. This may be purely in terms of input and output specification, as with a construct which does not have a concept of abstract state, or may also include extra checks to see if the abstract state has changed. This is achieved by making any component which implements a construct containing abstract state also implement a verification method, referred to here as a White-Box Test (WBT). These tests are used to detect whether there is a deviation between the expected behaviour, which is defined in a second DSL (referred to in this thesis as DSL-test), and the observed behaviour, obtained from the actual values in the interfaces or the WBT. Black-box reuse with additional capabilities to describe certain aspects of internal state can be viewed as very similar to grey-box reuse. Hence the following sub-hypothesis was proposed:

Given a specification of the problem modelled in sufficient detail it is possible to detect some errors occurring in a system and in some cases find the cause of those errors (i.e. faults).

The mapping between requirements described in terms of the DSL and the implementation would not always be 1:1. That is to say, alternative solutions for the same result may be possible. Therefore, upon detection of an error, the architecture will attempt to prescribe blame. A number of fault-prediction strategies have been proposed. A component which is blamed for a fault may be removed from future executables. That is to say:

Upon detection of an error, the system will attempt to ascertain where the fault lies. Then a new solution would be created if such an alternative is feasible within the system.

Chapter 4 then described the concepts of the support architecture (Hydra), DSLs, software generation techniques, error-detection and fault-prediction methods. Of particular interest are two principles of error-detection which have been proposed to aid in the automatic detection of errors and fault-prediction.

(1) The system should only detect errors when faults are present

(2) The system may not, in certain cases, detect an error when it occurs.

The implementation issues pertaining to developing the prototype supporting architecture, named Hydra, were discussed in chapter 5.

In Chapter 6, a number of simple problem domains were identified, in order to illustrate the methods and to highlight issues which were discussed later, in chapter 7. They were as follows:

1. Sort domain
2. Mathematical domain
3. Accountancy domain

The first domain suffered from a problem of DSL constructs duplicating functionality readily available in generic programming languages such as Java and C++. The second, mathematical, domain had different issues. It is highly likely that one mathematical function can be implemented in terms of others which exist in the system. For example, Square(x) and Multiply(x,x). The DSL system did not allow one DSL construct to be implemented by other DSL constructs. This caused unnecessary duplication of rules. The third, accountancy, domain was the first to use DSL constructs which contained abstract state. This prototype worked as planned, that is to say upon detection of an error it could predict which component was to blame and exclude its use in the next executable. However, a weakness in proposed scheme was found. Error-detection principle 1 could be broken by erroneous user input. This was caused by an error in the domain model.

Chapter 7 compared the methods and architecture with the evaluation criteria set out in chapter 1 and additional criteria whose importance became evident during the course of the research. A number of fault prediction algorithms were highlighted in chapter 4 and some of these have been evaluated in chapter 7. Two algorithms[16] were selected for profiling, to evaluate under what situations they performed best. It was found that both algorithms performed well under certain conditions.

Other aspects of using this method were described in chapter 7, including methods of improving the component combination mechanism, how and when to introduce DSL-construct-to-DSL-construct mappings, examining the issue of component and construct granularity, emphasising the importance of using vertical domains and issues of whether to design components around the DSL-constructs or to design DSL-constructs around components.

8.2 Further work

One of the primary aims of this thesis was to provide an initial investigation into the methods and theory required for dynamically re-configurable systems based on DSL constructs being mapped to components. This research has been very promising in addressing issues which should improve the confidence of software reusers, the rate of implementing new functionality and in empowering external software-developers by providing a common understanding of the problem domain. However, a number of issues still need to be researched, as described in the remainder of this section.

8.2.1 Integrating DSL systems with other systems

If these systems are to be of genuine application to real world problems, it is important to consider how they will be integrated with existing programming technologies and other DSL systems. It is not economical to duplicate the functionality of existing systems, especially generic programming languages, which currently provide equivalent functionality in a perfectly acceptable way. Hence, how will the proposed architecture be integrated into other programming environments? The existing architecture partially addresses this issue by using a CORBA architecture as a backbone. Any programming language or environment which can communicate with a CORBA system can communicate with the implemented system[17]. However, there are still issues of how best to integrate generic programming languages with the architecture. Of particular importance is the question of how to deal with different communication mechanisms, for example exceptions and event-driven mechanisms rather than simple call-returns. In addition to this, the method of integrating the DSL with existing systems, for example whether existing generic programming languages should be extended to support the generic DSL-construct systems, or to rely on CORBA to explicitly communicate between the two systems.

8.2.2 Optimising solutions

With a system that is able to generate a number of alternative solutions, it would be advantageous to select the "best" one. As the definition of what is "best" is usually domain-dependent, it seems conceptually sound to extend the domain model to reflect preferable characteristics of solutions. It may not be immediately obvious what factors are required to be present in the model in order to control certain characteristics. For example, in a time-dependent system it is fairly certain that operation-completion time would be an issue that would require modelling, but memory consumption may also be require modelling as this can indirectly cause the time characteristic to degrade due to the executing environment being forced to use swap space.

One method for achieving optimising behaviour whilst using the existing system is by using the existing architecture and defining an error to include events of undesirable performance characteristics. In this case it is also sensible to model the effect of generating a new solution in addition to its predicted performance. However, before the goal of using such a system for optimisation can be achieved, the methods and types of modelling required should be investigated to see if this

[16] There were actually three algorithms, but one was included purely as a basis for comparisons.
[17] This makes a number of assumptions about inter-compatibility of CORBA implementations. However, ignoring certain implementation differences, distinct CORBA implementations are predominantly compatible.

approach is genuinely feasible and to highlight issues which need to be considered at design time.

8.2.3 Advanced error-detection

The current techniques and guidelines for the error-detection systems are sub-optimal. In particular, the support for types requires an improved scheme, perhaps by considering type elements as first class objects in the DSL languages. In addition to this, more detailed consideration of how to use the data is needed.

Further investigation is required to see if categorisation of error-types will improve the error-detection process. That is to say, certain categories of errors, for example network failures, could be identified as being caused by factors external to the component and hence treated differently to a component which contains the fault. Different categories of generic and domain-specific errors may improve the feedback of the system and hence prevent scenarios such as network instabilities causing a component to be permanently excluded from future executables.

8.2.4 Advanced fault-prediction

A number of fault-prediction strategies have already been proposed. Two algorithms have been investigated in detail to evaluate their behaviour. However, it is evident that there is scope for improvement in this field. Also, the concept of components which have been incorrectly blamed still needs to be addressed In order to achieve this aim, there needs to be a mechanism for comparing similar occurrences. Then, if on situation 'x' component 'c1' was forbidden and a new executable is generated which does not use 'c1'. If this new executable still exhibits the problematic behaviour, then 'c1' could not be the only component containing a fault in the executable. If errors always occur with a certain parameter value, then this information could be utilised in the decision of when to use the component. There is also scope for fault-prediction based on composition rules.

8.2.5 Different DSL architectures

The current research has only considered a strict call hierarchy. The components are called by each other, the first calling the later ones. Although this architecture is quite commonly found in existing software systems, it is not the only architecture in existence. Other call hierarchies include pipe and filter models (Shaw and Garlan [46]) as shown in Figure 8:1, which can be viewed as a special case of the hierarchy used here, but all later calls have one argument.

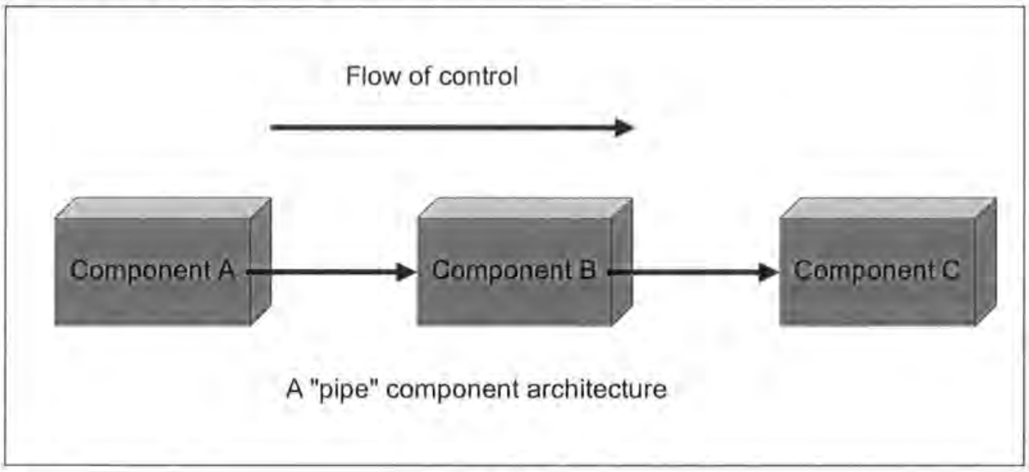


Figure 8:1

More complex architectures are often more appropriate for certain types of application. For example, user interfaces are often modelled using event-driven hierarchies to allow for more dynamic behaviour. It is unclear how, or if, DSLs should be applied to this category of problem and in particular what data should be encoded into the executables to aid in error-detection and fault-prediction.

8.2.6 Application as a Metric evaluation system.

If a large scale, well-designed architecture is developed in accordance with the guidelines in this thesis, there are potentially interesting alternative applications, to simply generating executables. Assuming a DSL, supporting architecture and components are created, metrics on the reliability of components could be applied. As previously stated, the architecture is highly dependent upon the data provided by the error detection model. If a metric is accurate, i.e. if it selects components which are less error-prone, then the system should[18] be less error-prone too. Therefore there is a potential application of such an architecture being appropriate for measuring the success of reliability metrics. If a system can be created which has a predictable performance model then it may be possible to evaluate performance metrics as well.

[18] Assuming random use-cases are selected and a sufficiently large number of test-cases.

8.3 Summary

This chapter has evaluated the methods and architecture in terms of the issues described in chapter 1 and in terms of issues whose importance became evident during the course of this research.

In evaluating the research with respect to the criteria identified in chapter 1, it was found that the proposed methods required additional short-term investment of effort. However, the preliminary findings suggest that if a system is used frequently enough to repay the initial investment then the methods provided an effective method of rapid application development, allowing the user to define their requirements in well-defined and familiar terminology. It also allows some of the programming to be delayed to later stages in development. The preliminary findings also suggested that there was increased confidence that the components were behaving as desired hence addressing, in part, the "not-invented here syndrome". The aim of this was to encourage the external development of software which, as stated in chapter 2, can be viewed as a method of sharing investment risk. Although substantial research remains before these aims can be claimed to be achieved for real-scale systems, the methods and architecture proposed in this thesis have been shown to achieve these aims for simpler domains and no reason was found that these methods would not be appropriate for actual systems.

The process of self-reconfiguration was found to be successful if the error-detection guidelines were followed. If the error-detection model was too weak then the number of components suspected was found to be on average higher than the number suspected using a more precisely specified construct. Two main types of fault prediction models were investigated and their behaviour evaluated in a number of cases. It was found that both performed well under different circumstances.

The methods and architecture was also evaluated in terms of other issues which were raised during the course of this research. It was found that the method of execution, which is currently based on a compile-then-execute cycle, was needlessly inefficient and an alternative, interpreter-based, method of implementation was proposed. The current system's inability to define DSL construct implementations in terms of other DSL constructs was outlined as a weakness and a scheme for introducing construct-to-construct mappings was introduced. This was found to be a non-trivial task as the ability to describe arbitrary mappings between constructs made infinite recursion a possibility and hence introduction of these rules would necessitate human intervention at the solution generation stage. As this was contrary to one of the aims of this research (to minimise human intervention), construct-to-construct mappings are only permitted in a limited form, i.e. no direct or in-direct recursion is permitted. The issue of DSL construct granularity was identified as being problematic. Specifically, it was found that if the granularity was too fine then the pay-off for using the DSL-system was too small. Another important issue was the type of domain to be implemented. It was found that two of the three domains were actually horizontal domains and as such had a tendency to duplicate common general purpose programming language functionality. This was found to be unsatisfactory. The final issue outlined was that of how to develop a DSL to component mapped system, whether to develop the language and then populate the component repositories or to develop the language in order to reuse existing components. A compromise between the two approaches seemed most appropriate.

9. References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*: Addison-Wesley Professional Computing Series, 1995.

[2] M. Buichi and W. Weck, "A plea for Grey-Box components" *Workshop on Component oriented programming, ECOOP*, 1997.

[3] C. Szyperski, *Component software: beyond object-oriented programming.* Harlow: Addison-Wesley, 1998.

[4] G. Kiczales, "Why black boxes are so hard to reuse : a new approach to abstraction for the engineering of software" : Stanford, CA : University Video Communications, c1994, 1994.

[5] J. E. Hollingsworth, "Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada", PhD. thesis, *Graduate School of the Ohio State University*: The Ohio State University, 1992.

[6] Honeywell, "What are the Benefits of Using a DSSA?", vol. 1998: Honeywell Technology Center, http://www.htc.honeywell.com/projects/dssa/ - DSSA page, 1999.

[7] J. S. Poulin, "Software Architectures, Product Lines, and DSSAs : Choosing the Appropriate Level of Abstraction" *Annual Workshops on Institutionalizing Software Reuse*, 1998.

[8] R. N. Taylor, W. Tracz, and L. Coglianese, "Software Development Using Domain-Specific Software Architectures: CDRL A011A curriculum Module in the SEI Style" *SIGSOFT Softw. Eng. Notes*, vol. 20, pp. 27-37, 1995.

[9] D. Batory, L. Coglianese, S. Shafer, and W. Tracz, "The ADAGE Avionics Reference Architecture" presented at AIAA Computing in Aerospace, San Antonio, 1995.

[10] D. Batory, D. McAllester, L. Coglianese, and W. Tracz, "Domain Modeling in Engineering Computer-Based Systems" presented at Internation Symposium and Workshop on Systems Engineering of Computer Based Systems, Arizona, 1995.

[11] B. Stroustrup, *The C++ programming language: Second edition*, Addison Wesley, 1999.

[12] M. Becker and J. L. Diaz-Herrera, “Creating Domain Specific Libraries: a methodology and design guidelines” presented at Third international conferences on Software Reuse: Advances in Software Reusability, Rio de Janeiro, Brazil, 1994.

[13] L. Burd and M. Munro, “Object Recovery” RISE, Computer Science, University of Durham, Durham, Technical Report 04/99, 1999.

[14] I. Sommerville, *Software Engineering*, 4th ed: Addison-Wesley, 1992.

[15] W. Tracz, *Confessions of a used program salesman : Institutionalising Software reuse*: Addison-Wesley, 1995.

[16] C. Brooke, M. Ramage, and N. Gold, “From legacy system to business asset: a model to support organisational and technological change” presented at Fifth Annual International Conference on Advances in Management, Lincoln, 1998.

[17] B. Robben, F. Matthijs, W. Joosen, B. Vanhaute, P. Verbaeten, and K. U. Leuven, “Components for non-functional requirements” presented at ECOOP 98:. Workshop on component oriented programming, Brussels, Belgium, 1998.

[18] D. d. Judicibus, “Reuse: A cultural change,” presented at International workshop on systematic reuse, Liverpool, 1995.

[19] N. E. Fenton, *Software metrics : a rigorous approach.* London: Chapman & Hall, 1991.

[20] J. S. Poulin, “Software Reuse: Been There, Done That,” *Communications of the ACM*, vol. 42, pp. 98-100, 1999.

[21] J. S. Poulin, *Measuring Software Reuse: Principles, Practices and Economic Models".* Reading: Addison Wesley, 1997.

[22] J. S. Poulin, “Measuring software reusability” presented at Third International conference on Software Reuse: Advances in Software Reusability, Rio de Janeiro, Brazil, 1995.

[23] SEI, “Capability Maturity Model (SW-CMM) for Software”: SEI, 2000.

[24] D. Lightfoot, *Formal specification using Z*: Macmillan, 1991.

[25] J. P. Bowen, *Formal specification and documentation using Z: a case study approach*: International Thomson Computer press, 1996.

[26] J. A. Bergstra, J. Heering, and P. Klint, *Algebraic specifications*: Addison-Wesley, 1989.

[27] I. V. Horebreek, *Algebraic specifications in software engineering*: Springer, 1989.

[28] J. Samentinger, *Software Engineering with Reusable components*: Springer Verlag, 1997.

[29] J. Bosch, "Superimposition: A component Adaptation Technique" *WCOP 97*, 1997.

[30] J. Neighbors, "An assessment of reuse technology after ten years" presented at Third International conference on Software Reuse:Advances in Software Reusability, Rio de Janeiro, Brazil, 1995.

[31] A. Brown and K. Wallnau, "The current state of CBSE," *IEEE Software*. September/October, 1998.

[32] W. Tracz, "Domain-Specific Software Architecture (DSSA) Frequently Asked Questions (FAQ)" Version 1.2 - ADAGE-IBM-93-12B ed: Lockheed Martin, http://www.owego.com/dssa/faq/faq.html, 1995.

[33] V. Traas, "Software component reuse survey", http://vtraas.cjb.net/, 1999.

[34] V. Baggiolini and J. Harms, "Toward automatic, run-time fault management for component-based applications", presented at WCOP 97, Finland, 1997.

[35] P. Henderson, "Modelling architectures for dynamic systems", http://www.ecs.soton.ac.uk/~ph/papers, 1999.

[36] E. J. Ostertag, "A classification system for software reuse", PhD thesis, *Computer Science*: University of Maryland, 1992.

[37] R. Prieto-Diaz, "Implementing faceted classification for software reuse", *Communications of the ACM*, vol. 34, pp. 88-97, 1991.

[38] H. Mili, E. Ah-Ki, R. Godin, and H. Mcheick, "Another Nail to the Coffin of Faceted Controlled-Vocabulary Component Classification and Retrieval", presented at 1997 ACM Symposium on Software Reusability (SSR'97), Boston, MA, 1997.

[39] C. Nelson, "A forum for fitting the task," *IEEE computer*, vol. 27, 1994.

[40] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented software engineering*, Addison-Wesley, 1996.

[41] W. Pree, *Design patterns for object-oriented software development*, Addison Wesley, 1995.

[42] R. Mauth, "A better foundation", Byte, September, 1996.

[43] Taligent, "Building Object oriented frameworks", IBM:Taligent / IBM, http://www.taligent.com/resources-list.html, 1998.

[44] M. Fayad and D. C. Schmidt, "Object-Oriented frameworks", *Communcations of the ACM*, vol. 40, 1997.

[45] R. Johnson and B. Foote, "Designing reusable classes", *Object-oriented programming (SIGS)*, Vol. 1, pp. 22-35, 1988.

[46] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an emerging discipline,* Prentice-Hall, 1996.

[47] M. Shaw and D. Garlan, "An Introduction to Software Architecture", *Advances In Software Engineering*, Vol. 1, 1993.

[48] C. Hofmann, E. Horn, W. Keller, K. Renzel, and M. Schmidt, "The field of Software Architecture", Technische Universitat Munchen, Munchen TUM-I9641, November 7. 1997.

[49] P. Clements and L. Northrop, "Software Architecture: An executive overview", Report on the reuse and product line working group of WISR8, CMU CMU/SEI-96-TR-003, 1996.

[50] R. Richardson, "Components battling components", Byte, November, 1997.

[51] E. Mettala and M. H. Graham, "The Domain-Specific Software architecture program", CMU CMU/SEI-92-SR-9, 1992.

[52] L. Coglianese, R. Smith, and W. Tracz, "DSSA Case Study: Navigation, Guidance, and Flight Director Design and Development", presented at 1992 IEEE Symposium on Computer Aided Control System Design, Napa CA, 1992.

[53] L. Martin, "Lockheed Martin STARS Reuse papers", http://www.asset.com/stars/lm0tds/Papers/ReusePapers.html , 1999.

[54] L. Martin, "Lockheed Martin ADAGE Documents", http://www.owego.com/dssa/ox-docs/ox-docs.html, 1999.

[55] P. Clements, "Report on the reuse and product line working group of WISR8", CMU CMU/SEI-97-SR-010, 1997.

[56] K. Czarnecki, "Leveraging Reuse Through Domain-Specific Software Architectures", presented at Eighth Annual Workshop on Institutionalizing Software Reuse, Ohio State University, 1997.

[57] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[58] IEEE, *Software engineering (AKA Software engineering standards)*, New York, 1993.

[59] M. Grossman, "Component Testing", presented at ECOOP 98: Workshop on component oriented programming, Brussels, Belgium, 1998.

[60] W. H. Deason, D. B. Brown, K.-H. Chang, and J. H. C. II, "A Rule-Based Software Test Data Generator", *IEEE Transactions on knowledge and data engineering*, vol. 3, pp. 108 - 117, 1991.

[61] Aonix, "Validator/Req", http://www.aonix.com, 2000.

[62] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, vol. (10)7, pp. 352-357, 1984.

[63] R. Rubinfeld, "Robust Functional Equations with Applications to Self-Testing/Correcting", TR 94-1435, July 1994.

[64] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing/correcting with applications to numerical problems", presented at 21st Annual ACM Symposium on Theory of Computing, 1990.

[65] M. Wooldridge and N. R. Jennings, "Agent Theories, Architectures and Languages: A Survey", presented at ECAI-94: Intelligent Agents: Workshop on Agent Theories, Architectures and Languages, Amsterdam - The Netherlands, 1994.

[66] S. Franklin and A. Graesser, "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", presented at Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, 1996.

[67] M. Wooldridge and N. R. Jennings, "Intelligent Agents: Theory and Practice", *Knowledge Engineering Review*, vol. 10, 1995.

[68] F. Farhoodi and I. Graham, "A Practical Approach To Designing and Building Intelligent Software Agents", presented at The Practical Application of Intelligent Agents and Multi-Agent Technology, London, 1996.

[69] AgentsMailingList, "The agent's mailing list", edited by Tim Finin & Yannis Labrou, http://www.csee.umbc.edu/agentslist/, 1998.

[70] FIPA, "Federation for intelligent physical agents", http://www.fipa.org/, 2000.

[71] A. Caglayan, M. Snorrason, J. Jacoby, J. Mazzu, R. Jones, and C. R. Analytics, "Lessons from Open Sesame!, a User Interface Learning Agent", presented at The Practical Application of Intelligent Agents and Multi-Agent Technology, London, 1996.

[72] A. Chavez and P. Maes, "Kasbah: An Agent Marketplace for Buying and Selling Goods", presented at The Practical Application of Intelligent Agents and Multi-Agent Technology, London, 1996.

[73] K. Nygren, I.-M. Jonsson, and O. Carlvik, "An Agent System for Media on Demand Services", presented at The Practical Application of Intelligent Agents and Multi-Agent Technology, London, 1996.

[74] C. Krogh, "The Rights of agents", *Intelligent agents II, Lecture Notes in AI Volume 1037*, M. Wooldridge, J. P. Mueller, and M. Tambe, Eds.: Springer-Verlag, 1995.

[75] Y. Shoham, "Agent-oriented programming", *Artificial Intelligence*, vol. 60, 1993.

[76] UMBC, "UMBC Agent Web", http://agents.umbc.edu/, 2000.

[77] J. Ingham, "What is an Agent?", Research Institute for Software Evolution., Durham, UK #6/99, published 1999, (written 1997).

[78] Y. Labrou, "Semantics for an agent communication language", PhD thesis, *Computer science and electrical engineering*. Baltimore: University of Maryland, 1996.

[79] S. Pearson, "Agent communication languages",
http://www-uk.hpl.hp.com/projects/viceroy/language.html, 2000.

[80] V. Singhal and D. Batory, "P++: A Language for Software System Generators", University of Texas at Austin, Austin, Technical report tr-93-16, 1993.

[81] J. Bentley, "Little Languages", *Communications of the ACM*, vol. 29, 1986.

[82] M. P. Ward, "Language-Oriented Programming", *Software-Concepts and Tools*, vol. 15, pp. 147-161, 1994.

[83] A. v. Deursen and P. Klint, "Little Languages: Little Maintenance?", CWI, Amsterdam December 16 1996.

[84] D. Spinellis and V. Guruprasad, "Lightweight languages as software engineering tools", presented at Conference on Domain-Specific languages, 1997.

[85] C. Rich and R. C. Waters, "Approaches to Automatic programming", *Advances in computers*, vol. 37, pp. 1-57, 1993.

[86] J. M. Neighbors, "Software construction using components", in *Information and computer science*. Irvine: California, 1980, pp. 82.

[87] J. A. Feldman, "Automatic Programming", Stanford University STAN-CS-72-255, February 1972.

[88] J. M. Neighbors, "Draco user manual", University of California, Irvine TR-156, 1908.

[89] R. M. Balzer, "A Global view of automatic programming", presented at Third Joint conference on artificial intelligence, 1973.

[90] V. P. Singhal, "A programming language for writing Domain-Specific software system generators", PhD thesis, *Computer Science*. Austin: University of Texas, 1996.

[91] P. Bucci, S. Edwards, J. Hollingsworth, J. Krone, T. Long, W. Ogden, M. Sitaraman, S. Sreeratna, B. Weide, S. Zhupanov, and S. Zweben, "Special Feature : Component-based software using RESOLVE", *Software engineering notes*, vol. 19, 1994.

[92] J. Ingham and M. Munro, "Applying a domain-specific language approach to component oriented programming", presented at Workshop on component oriented programming: ECOOP, Brussels, 1998.

[93] J. M. Stichnoth and T. Gross, "Code composition as an implementation language for compilers", presented at Conference on Domain-Specific Languages, 1997.

[94] R. A. Mueller, "Automated Microprogram Synthesis", in *Computer Scien*ce, Boulder: Colorado State University, 1980.

[95] D. Batory and B. J. Geraci, "Validating component compositions in software system generators", presented at ICSR, Orlando, Florida, 1996.

[96] R. Orfali, D. Harkey, and J. Edwards, *Instant CORBA*: John Wiley and Sons, 1997.

[97] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modelling Langauge User Guide* Addison-Wesley, 1998.

[98] R. Orfali and D. Harkey, *Client/server programming with Java and CORBA*. New York: Wiley Computer Pub, 1997.

[99] C. Szyperski, "Emerging Component Software Technologies: A Strategic Comparison", *Software Concepts and Tools*, vol. 19, pp. 2-10, 1998.

[100] E. Friedman-Hill, "Java Expert Systems Shell (JESS)", vol. 1997: Sandia National Laboratories, Livermore, 1997.

[101] J. J. Weygandt, D. E. Kieso, and W. G. Kell, *Accounting Principles*, 2nd ed: John Wiley and Sons, 1990.

[102] R. Korfhage, "References on Visual Languages", vol. 2000: University of Pittsburgh, 2000.

[103] M. H. Jarvis, "Foundation Logics: Reuse of languages," University Of Durham, Durham (UK), Draft October 8th 1997.

ORIGINAL IN COLOUR

10. Appendix A

This section describes the DSL and underlying architectural design assumptions which were created in order to develop the Accounting Domain DSL. This domain contains abstract constructs with and without state. The level of detail is aimed at describing the DSL and issues in sufficient detail to allow further discussion in chapter 7.

The DSL is structured on a compositional parameterised model. Component compositions can be a single component call, a sequence of component calls or a hierarchy of component calls, as illustrated in Figure 10:1

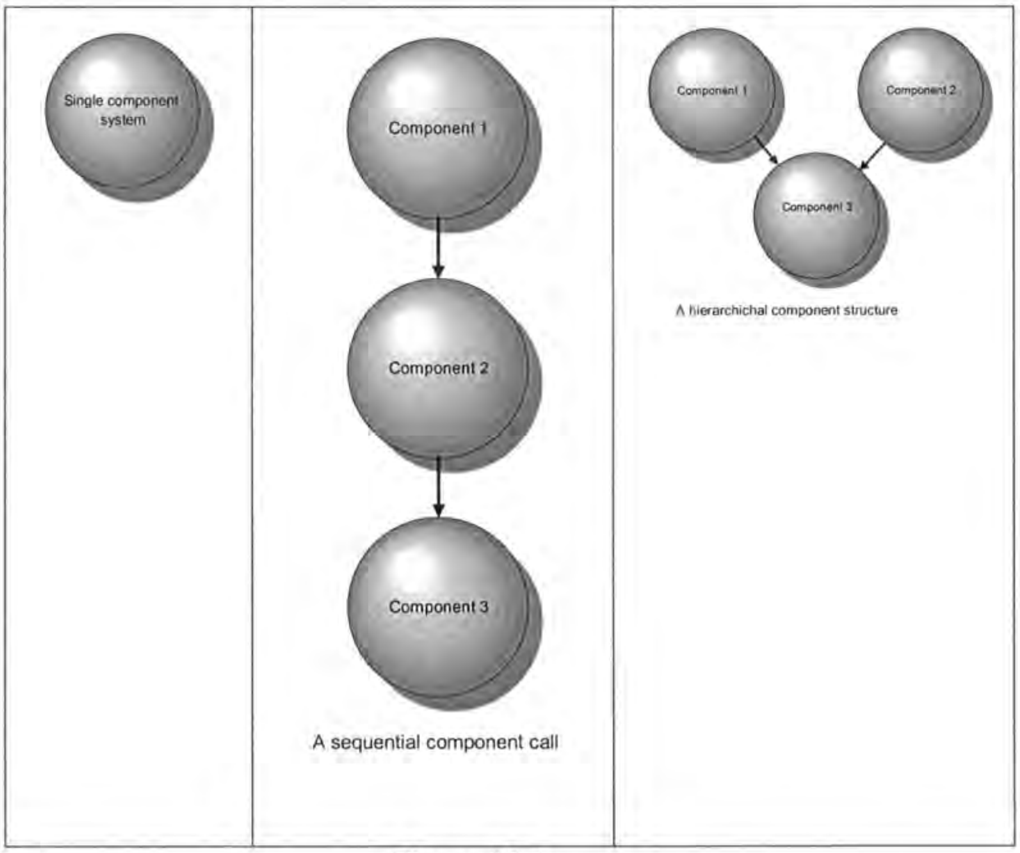


Figure 10:1

The DSL was designed in order to express the needs of its target user in simple terminology. However the user would not be usually expected to write their requirements in the native representation format. This should be used for system-level communication. Typically, the user manipulates the constructs using a GUI. A rudimentary DSL GUI was developed during this research for this DSL. Some screen-shots of the GUI are present in chapter 6.

As previously stated, the DSL is developed with the notion of the sort of jobs the user needs to perform and the perspective the user is likely to view the problem from. The target user for this DSL is someone whose responsibilities are as follows:

- To add new records
- To delete records
- To alter records
- To obtain lists of records, manipulate them and provide totals

To re-iterate the problem domain for this DSL is kept artificially simple for experimental and pedagogical purposes.

The location of the data stored is of particular importance for this DSL. That is to say when something is stored or retrieved, there is a concept of where the data is stored.

10.1 Types in the system

10.1.1 String

Based on the underlying representation in CORBA. The string is a text field which can be used to represent a number of different items of data (e.g. name, address). The string has no maximum length.

10.1.2 Bool

A True/False value, based upon the underlying representation of boolean in CORBA.

10.1.3 Int

A numerical integer value, based upon the underlying representation of long in CORBA.

10.1.4 Real

A numerical value, based upon the underlying representation of float in CORBA.

10.1.5 Data

A record of the information stored in the accountancy database.

10.1.5.1 CompanyName

Abstract type **CompanyNameType** (currently represented as **String**).

A representation of the name of the company about whom the information has been stored.

10.1.5.2 CompanyAddress

Abstract type **CompanyAddressType** (currently represented as **String**).

A representation of the address of the company about whom the information has been stored.

10.1.5.3 OrderNo

Abstract Type **OrderNoType** (currently represented as **Int**).

A representation of the Order Number by which this entry account is to be considered. **OrderNo** must be unique.

10.1.5.4 AmountOwed

Abstract Type **AmountOwedType** (currently represented as a **Real**).

A representation of the amount owed by the company.

10.1.5.5 Date

Abstract Type **DateType** (currently represented as **String**).

A representation of the data when the order was made.

10.1.6 List

The list is the representation of a sequence of **Data** entries.

In addition to the types described, the following operations are defined for this domain:

10.2 Operations in the system

In the prototype implementation of this DSL only one WBT operation is permitted per construct.

10.2.1 Add

10.2.1.1 Informal description

Adds a new entry to the database.

The OrderNo of the data must not already be present in the Database.

10.2.1.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 Data to be added(Data) – The data which is to be added to the database

10.2.1.3 Returns

Initially there was no return type associated with this operation. This was changed to a **Bool** return value as there must be a way of expressing not being able to add a component without the underlying component implementation being blamed.

10.2.1.4 WBT

This returns the abstract number of records in the database. That is to say if there have been 8 successful Add operations then the WBT should return 8. This is represented as **Int.**

10.2.2 UpdateAmountOwedOnOrderNo

10.2.2.1 Informal description

Updates the Amount owed field of the database entry with the same **OrderNo** as passed as an argument in this construct.

10.2.2.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.
#2 OrderNo (Int) – The order number of the account to be updated.
#3 Amount (Real) – The amount by which the value should be changed.

10.2.2.3 Returns

Bool return value to express the concept of whether the record has been updated, i.e. is the data in the database and hence can it be updated.

10.2.2.4 WBT

This returns the amount owed in the record which contains the correct order number.
The WBT can return any value if the record does not exist.

10.2.3 UpdateNameOnOrderNo

10.2.3.1 Informal description

Updates the Company name field of the database entry with the same **OrderNo** as passed as an argument in this construct.

10.2.3.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.
#2 OrderNo (Int) – The order number of the account to be updated.
#3 CompanyName (String) – The new company name of the account to be updated.

10.2.3.3 Returns

Bool return value to express the concept of whether the record has been updated, i.e. is the data in the database and hence can it be updated.

10.2.3.4 WBT

This returns the company name in the record which contains the correct order number.

The WBT can return any value if the record does not exist.

10.2.4 UpdateAddressOnOrderNo

10.2.4.1 Informal description

Updates the Company address field of the database entry with the same **OrderNo** as passed as an argument in this construct.

10.2.4.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 OrderNo (Int) – The order number of the account to be updated.

#3 CompanyAddress (String) – The new company address of the account to be updated.

10.2.4.3 Returns

Bool return value to express the concept of whether the record has been updated, i.e. is the data in the database and hence can it be updated.

10.2.4.4 WBT

This returns the company address in the record which contains the correct order number.

The WBT can return any value if the record does not exist.

10.2.5 UpdateDateOnOrderNo

10.2.5.1 Informal description

Updates the Company address field of the database entry with the same **OrderNo** as passed as an argument in this construct.

10.2.5.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 OrderNo (Int) – The order number of the account to be updated.

#3 Date (String) – The new date of the account to be updated.

10.2.5.3 Returns

Bool return value to express the concept of whether the record has been updated, i.e. is the data in the database and hence can it be updated.

10.2.5.4 WBT

This returns the date in the record which contains the correct order number.

The WBT can return any value if the record does not exist.

10.2.6 RemoveOnOrderNo

10.2.6.1 Informal description

Removes the database entry with the same **OrderNo** as passed as an argument in this construct.

10.2.6.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 OrderNo (Int) – The order number of the account to be removed.

10.2.6.3 Returns

Bool return value to express the concept of whether the record has been updated, i.e. is the data in the database and hence can it be updated.

10.2.6.4 WBT

This returns the abstract number of records in the database. That is to say if there have been 8 successful Add operations and 3 successful remove operations then the WBT should return 5. This is represented as **Int.**

10.2.7 GetAllWithDate

10.2.7.1 Informal description

This returns a list of all the data entries which have the same date as passed as a parameter

10.2.7.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 Date (String) – The date of the accounts to be obtained.

10.2.7.3 Returns

List return value to express the records which fulfil the criteria.

10.2.7.4 WBT

There is no WBT for this construct.

10.2.8 GetAllWithDateBefore

10.2.8.1 Informal description

This returns a list of all the data entries whose date is before the date passed as a parameter

10.2.8.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 Date (String) – The date of the accounts to be obtained.

10.2.8.3 Returns

List return value to express the records which fulfil the criteria.

10.2.8.4 WBT

There is no WBT for this construct.

10.2.9 GetAllWithDateAfter

10.2.9.1 Informal description

This returns a list of all the data entries whose date is after the date passed as a parameter

10.2.9.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 Date (String) – The date of the accounts to be obtained.

10.2.9.3 Returns

List return value to express the records which fulfil the criteria.

10.2.9.4 WBT

There is no WBT for this construct.

10.2.10 GetAllWithCompanyName

10.2.10.1 Informal description

This returns a list of all the data entries who have the same company name as the name passed as a parameter

10.2.10.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 CompanyName (String) – the name of the company by which to retrieve data.

10.2.10.3 Returns

List return value to express the records which fulfil the criteria.

10.2.10.4 WBT

There is no WBT for this construct.

10.2.11 GetAllWithCompanyAddress

10.2.11.1 Informal description

This returns a list of all the data entries who have the same company address as the name passed as a parameter

10.2.11.2 Parameters

#1 Storename (String) - The name of the location the component is to be stored.

#2 CompanyAddress(String) – the address of the company by which to retrieve data.

10.2.11.3 Returns

List return value to express the records which fulfil the criteria.

10.2.11.4 WBT

There is no WBT for this construct.

10.2.12 SumLists

10.2.12.1 Informal description

This operation sums the amount owed by each element of data in the list.

10.2.12.2 Parameters

#1 theList (of type **List**)– The list of data to be summed

10.2.12.3 Returns

Real – The sum of the amount owed field of every record in the list.

10.2.12.4 WBT

There is no WBT for this construct.

10.2.13 AddLists

10.2.13.1 Informal description

This operation joins two lists of data.

Duplicate entries or OrderNos are permissible in the resulting list.

sums the amount owed by each element of data in the list.

10.2.13.2 Parameters

#1 ListA (of type **List**)– a list of data

#2 ListB (of type **List**) – a list of data

10.2.13.3 Returns

List – A list containing all the records in ListA and ListB. If ListA and ListB share common elements, then they will appear twice.

10.2.13.4 WBT

There is no WBT for this construct.

10.2.14 SubtractList2FromList1

10.2.14.1 Informal description

All of the elements which appear in the returned list must appear in List1.

If an element appears in List1 and List2 then it should not appear in the returned list.

10.2.14.2 Parameters

#1 ListA (of type **List**)– a list of data

#2 ListB (of type **List**) – a list of data

10.2.14.3 Returns

List – A list containing all the records in ListA unless they appear in ListB.

10.2.14.4 WBT

There is no WBT for this construct.