Improving Interoperability to Facilitate Reverse Engineering Tool Adoption

By

David Michael Zwiers
B.Sc., University of Victoria, 2001

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© David Michael Zwiers, 2004

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

Supervisor: Dr. H. A. Müller

Abstract

Although we cannot show a direct link between tool adoption and tool interoperability in this thesis, we have completed the first step by increasing our understanding of interoperability. This thesis shows how to use existing technology such as XML, SOAP and GXL to improve interoperability. Although the ideas behind XML are not new, XML has been used to increase interoperability between systems. While the goal is to improve interoperability, we also keep in mind other software engineering design concerns, such as ease of maintenance and scalability.

To evaluate our ideas about improving interoperability, we completed a prototype, which allows us to compare our approach to other existing systems. Our prototype is a reverse engineering tool for which existing systems and requirements are readily available. Some of the more relevant requirements include tool customization, persistence, tool deployment and interoperability. These requirements were combined with the reverse engineering requirements in the design stages of development in the hope of creating a more cohesive system.

In our quest to improve interoperability of reverse engineering tools, we considered three types of integration. Data integration refers to the extent to which applications can share or use each other's data. Control integration is the ability of one system to request another system to perform some action. Process integration is similar to other forms of integration in so far as it looks at how to easily move between two user processes or actions.

In this thesis we compare our prototype, the ACRE Engine, with the Rigi system. The comparison focused on our understanding of interoperability. We found that the Rigi system has many data integration features—most of which stem from its proprietary data format, Rigi

Standard Format (RSF). Rigi's ability to integrate control between applications is restricted to file system messages. We did find the Rigi system could complete process integration tasks effectively. In this thesis we show that the ACRE System is at least as good, and in most cases better than the existing Rigi system with respect to the three forms of interoperability mentioned above.

Contents

Contents iv

List of Tables viii

List of Figures ix

Acknowledgments x

1. Introduction 1

1.1 Problem 1

1.2 Motivation 1

1.3 Approach 2

1.4 Solution 2

1.5 Thesis Outline 3

2. Foundations and Background 5

2.1 Introduction to Architecture 5

2.2 Introduction to Maintenance 6

2.3 Introduction to Interoperability 8

2.3.1 Why are Interoperability mechanisms important? 9

2.3.2 When is a pair of components interoperable? 9

2.3.3 What is good interoperability? 10

2.4 Exchange Formats 10

2.4.1 XML 11

2.4.2 GXL 12

2.4.3 SOAP 12

2.4.4 RSF 13

2.4.5 SVG 13

2.5 Summary 14

3. Problem Definition 15

3.1 Introduction 15

3.2 Tool Tasks 15

3.3 Tool Customization 17

3.4 Tool Persistence 18

3.5 Tool Deployment 19

3.6 Tool Interoperability from a User's Perspective 20

3.7 Summary 20

4. ACRE Engine Prototype 22

4.1 ACRE Engine Prototype Overview 22

4.2 High Level Architecture 23

4.3 ACRE Engine 27

4.4 ACRE Engine's User Gateway Module 29

4.5 ACRE Engine's Data Gateway Module 32

4.6 ACRE Engine's Memory Module 33

4.7 ACRE Engine's Scripting Module 34

4.8 Summary 36

5. ACRE Interoperability 38

5.1 Introduction 38

5.2 Data Integration 38

5.3 Control Integration 40

5.3.1 Web Integration 40

5.3.2 Service Integration 42

5.4 Process integration 43

5.5 Summary 44

6. Prototype Evaluation 45

6.1 Introduction 45

6.2 Data Integration 45

6.3 Control Integration 46

6.3.1 Web Integration 46

6.3.2 Service Integration 47

6.4 Process Integration 48

6.5 Summary 49

7. Related Work 50

7.1 JGraphPad 50

7.2 Creole 50

7.3 TXL 51

7.4 PBS 52

7.5 BOOST 52

7.6 IPSEN 52

7.7 Summary 53

8. Conclusions 54

8.1 Summary 54

8.2 Contributions 55

8.3 Future Work 56

Bibliography 58

A. XML Sample Document and Schema 62

B. WSDL 66

C. Microsoft SOAP Connection 77

D. Tcl Extensions 79

E. SVG Communication Scripts 81

F. Memory Model Graph Class 85

List of Tables

2.1 Lehman's Laws of Software Evolution [1] 7

D.1 Tcl API extension for the ACRE Engine 80

List of Figures

2.1 Sample Interoperability Component Interaction 8

4.1 ACRE Prototype System Architecture 26

4.2 ACRE Engine Architecture 30

4.3 ACRE Engine Memory Model for a Graph 35

5.1 SVG Client Application 41

Acknowledgments

The Adoption Centric Reverse Engineering (ACRE) project is a team effort. It takes many different types of skills and personalities to create such a kind of system. As with any group, we needed leadership. For that I would like to thank Hausi Müller, who gave me the opportunity to work on this research project; it has been a great learning experience. His energy and guidance have helped me greatly through the course of my work—and for that I am grateful. I would also like to thank the other group members, who have each contributed in their own unique ways, Holger Kienle, Jun Ma, Fang Yang, Fei Zhang, Piotr Kaminski, Grace Gui, and Qin Zhu. I would also like to add a special thank-you to Will Kastelic for his help in developing the original prototype, but also for sharing all of my frustrations. My last set of acknowledgements goes to my friends and family, who have put up with me. Thanks Mom, Dad, James, Ian and all of my friends, of which there are too many of you to mention.

To My Parents

Chapter 1

Introduction

1.1 Problem

This thesis attempts to shed light on selected aspects of the software engineering tool adoption problem—why tools, which are apparently useful, are not used regularly. In this thesis we investigate tool interoperability, providing the building blocks for future work into the relationship between tool adoption and interoperability.

Throughout this thesis we are investigating interoperability and possible factors, which could affect interoperability. This thesis compares types of interoperability between systems by an in-depth comparison of various forms of interoperability. This is intended to allow the reader to form recommendations based on these comparisons.

1.2 Motivation

Although reverse engineering tools have become more powerful over the past decade, industry has not experienced a significant increase in reverse engineering tool use. Two of the many possible causes are the lack of advertisement of this genre of product or the tools are too difficult to use for an average software engineer to see the potential benefit of reverse engineering tools. Since reverse engineering is a well-known discipline, the hypothesis is that tool adoption has hampered the widespread use of reverse engineering tools. This assumption led to the Adoption Centric Reverse Engineering (ACRE) project [2]. As part of this project, we investigated the importance of interoperability with respect to tool adoption in reverse engineering environments.

1.3 Approach

The ACRE project has a series of interoperability goals, in particular tool development, customization and deployment. Therefore, we embarked on building a prototype and investigated methods to evaluate the potential improvements in interoperability. Although we only intend to evaluate system interoperability, other project goals, such as providing persistence between user sessions, influenced some of our decisions. In terms of a single user application, this is relatively straight-forward however, the user was also required to access the data through multiple lightweight platforms or applications, which complicated the plan.

This provided a new challenge as we needed a medium where all applications could be accessed equally well, without reducing the functionality requirements. The solution was to create a client server interface where the clients all communicate with the server in a similar fashion. This permitted the storage of data in a central repository and allowed users to access their data from multiple platforms or applications. This was important as most of our non-reverse engineering requirements were central to persistence and interoperability among multiple different applications on multiple platforms. The only remaining problem was providing persistence inside a single user session. This was solved through the use of server-side *servlet* technology [3], which includes methods of providing short term persistence. *Servlet* technology is a Java web technology allowing systems to provide a wide range of data and services. Servlets can provide responses to the client in both binary and textual formats, which are intended for both human and machine clients.

1.4 Solution

The ACRE project hypothesizes that it is possible to investigate how the human user interface, interoperability and cognitive support independently affect tool adoption. In this thesis, we focus on interoperability and build the portion of the project pertaining to interoperability. Our goal in this thesis is to show methods in which interoperability can be both

improved, and measured. Although it would greatly help our project, this thesis does not intend to show that there is a relation between improved interoperability and improved tool adoption. Rather this thesis intends to take the first steps towards understanding, evaluating and improving interoperability for reverse engineering tools.

The proposed solution for the ACRE project was to create multiple modularized interoperable components. Some of the components would include user interfaces, repositories, and an ACRE server. The ACRE server was coupled with the interoperability protocols to help meet our goals for improving and evaluating interoperability. Through this thesis, we focus on the interoperability between the user interface and the ACRE Engine which is vital and central to the ACRE system.

1.5 Thesis Outline

This thesis begins with some background information on a variety of topics which are used later in the thesis. Chapter 2 contains information on high-level topics such as architecture, then drills down to lower level topics, and ends with a discussion of selected new technologies. In Chapter 3, we define our problem and start to see how all of the high and low level topics from the preceding chapter work together to develop a solution. This chapter gives a more in-depth explanation of each of the four main issues this thesis investigates.

Chapters 4, 5 and 6 present a solution to the problem and a comparison to an existing system. These chapters show how the background materials are applied to our problem to produce a solution on multiple levels of abstraction. Chapter 4 provides an in-depth look into our prototype, the ACRE Engine, and many of its features. Chapter 5 introduces various forms of interoperability, and shows how the ACRE Engine supports all of these forms. This chapter is setting the bar to which we compare the Rigi reverse engineering system in Chapter 6.

Some conclusions and observations are finally presented in Chapter 8 after considering

related projects in Chapter 7. Some of the other approaches to reverse engineering systems are considered, as well as other trends towards improving integration. Finally, we present some avenues for future research.

Chapter 2

Foundations and Background

2.1 Introduction to Architecture

Software Architecture is an evolving term, which is generally accepted to mean *the abstraction of a software system*. Shaw defines the architecture of a software system in terms of computational components and interactions among these components [4]. Although this definition is very broad, in practice software architectures are exactly that, a set of components and their interactions. The components, or modules, are abstracted portions of a system, and may represent any component, whether human, machine, code, or any combination. We should also realize that every system has an architecture. The one question about each system is whether the architecture is documented using one or more views.

Software architectures take time to document, thus possibly increasing the cost of evolving a system. Today developers create software architecture methodically and re-document it when a system is modified. When development teams use this practice, the results often include a better cost benefit ratio [1] [4].

Logically developed software architectures—those that are created with intent—often result in cost-saving benefits, such as added system structure, system understandability, cheaper maintenance, extensibility, and development. Other benefits include the ability to split the system into sub-systems, allowing a team to be broken into smaller sub-teams, thus creating experts in each sub-system. The additional knowledge results in faster maintenance and development as the code, which needs to be edited, is significantly smaller, reducing the required learning for some individuals when attempting to understand the sub-system. Well documented architectures also allow for simpler extensibility as the potential user would

be capable of viewing the system to understand where the extensions would be best suited.

Some typical architecture styles include: pipes and filters, object-oriented paradigms, layers of indirection, client-server, three tiers, and others [4]. These commonly found patterns all show many similarities beyond the obvious intent to structure the system. All of these architecture patterns organize the interactions between components into well-defined interfaces. They also reduce the number of different types of interactions coming and going from a module to minimize the effect of changing that particular module. All of these architectures organize the system into sub-systems, which reduces duplication and complexity—leading to a more cost effective system.

2.2 Introduction to Maintenance

Maintenance is defined as "any activity intended to keep equipment, programs or a database in satisfactory working condition" [5]. This does not preclude the ability of maintenance to have negative long-term affects. Most authors in the field agree that there are three types of maintenance: corrective, perfective and adaptive [5] [1] [6]. Although we are looking at maintenance as a concept and are not distinguishing between the three reasons for performing maintenance, we are interested in how maintenance can be affected. To this end, we should understand the motivations behind both performing maintenance tasks, and studying maintenance.

Corrective maintenance is performed when an error is discovered and rectified. This includes all types of errors, from simple code errors to requirement and design errors. Often errors may be relatively inexpensive to fix, but a requirement or design error can be very costly when discovered after the system has been implemented. Requirement and design errors are often represented as feed back loops in software engineering development models.

Perfective maintenance is more frequently performed with legacy systems. This is usually

the result of a new requirement for an existing system. This is not considered corrective because the new requirement change does not represent an error in the previous requirements. One example could be to change a tax system within an automated banking process. This does not imply the old requirements were incorrect, but simply that a new requirement has been added to allow continued use of a valuable code base.

Adaptive maintenance involves applying the current requirements from one platform to another platform. This usually is the result of new hardware or software packages, which need to interact with the current system. This is common in legacy systems, usually when a new protocol for interaction is required or hardware replacement is imminent. Two examples include the introduction of CORBA into legacy systems and the transition from one operating system to another.

Table 2.1: Lehman's Laws of Software Evolution [1]

Law	Description
Continuing change	A program that is used in a real world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program's lifetime, it's rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.

Understanding what types of maintenance occur is only half the story, Lehman's laws (Table 2.2) tell the rest. Lehman developed these rules to explain his observations on software evolution. The rules which have the largest effect on software maintenance are the first,

second and fourth laws. Essentially, Lehman states that programs will be forever changing, and while we may want to change the costs and complexity of a system, eventually the maintainers will be limited by the original design.

Every phase in the software cycle has costs associated with it. However, maintenance tends to be much more costly than the other phases, often producing most of the cost for a software system over its lifetime. Some estimates indicate that maintenance could account for half of the cost of a system [1] [6]. Other authors quote as high as 75% of the cost of the system during its lifetime is maintenance [7].

2.3 Introduction to Interoperability

Interoperability is the ability of two or more systems or components to exchange and use information and use each others operations [5] [8] [9]. In this thesis, we refer to systems as *components*. This will not affect the conclusions because both systems and components have the same properties with respect to this thesis. Interoperability must then have at least two components that communicate or exchange data [5]. For this to occur, a set of conventions or protocols must be defined to govern the interaction of components [5] as depicted in Figure 2.1.

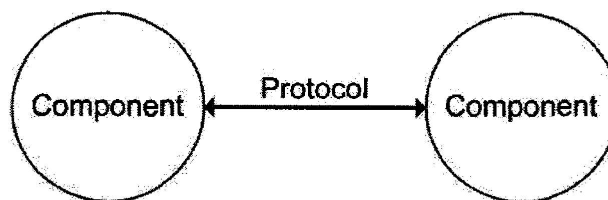

Figure 2.1: Sample Interoperability Component Interaction

A protocol between two components is represented by the incoming and outgoing requirements of each individual component. This provides an insight into techniques for evaluating the interoperability of two components. This does not conflict with the idea of components calling each other within an existing program as the function calls are also defined

protocols.

2.3.1 Why are Interoperability mechanisms important?

Interoperability mechanisms allow systems to communicate many types of information efficiently and reliably between each other. This ability to communicate allows multiple types of information to be passed, including control flow directives or data between sources and users. This allows larger systems to be created by joining smaller systems together resulting in a simpler solution. The solution is simpler because the systems can use portions of each other to complete difficult tasks, reducing the amount of effort required to develop the overall system. Some popular implementation protocols of interoperability include plug-in architectures, application programmer interfaces, and XML (cf. Section 2.4.1). One common implementation technique for interoperability is the use of scripting interfaces to interface with various forms of data, control, and presentation integration.

2.3.2 When is a pair of components interoperable?

Interoperability requires that a component has the ability to exchange with other components and use information from other components [5]. In this thesis, we do not distinguish between levels of abstraction with respect to interoperability because interoperability follows the same patterns at different levels of abstraction. For example both code and module level interoperability abstractions have protocols which are defined explicitly by a compiler, an interpreter, or by a human user. We also might ask what constitutes “good” interoperability. This is important as it is necessary to be able to compare the interoperability of individual components and their protocols. This affords the opportunity to compare interoperability.

2.3.3 What is good interoperability?

Components can be considered more interoperable when they can easily communicate with a greater number of components or if there are protocols to communicate directly between components. When an intermediate node, or *ontology translator*, is required for two components to communicate their intensions, the interoperability between these nodes is reduced.

Generally the measure of interoperability between two components is inversely proportional to the effort required for two components to work together. This effort may be measured with respect to execution time, development time or the time required by an end user to use the interoperability provided. In many cases all three measures are combined to determine the quality of the interoperability between two systems or components.

Protocols can also be evaluated in a similar manner. A protocol, which promotes interoperability, is used to communicate between multiple components. Statistical measures, such as the average, can be used to compare protocols by computing a measure of the effort required for a protocol to communicate between any two components.

Effort is quantified in terms of the requirements for two components to communicate with each other. This includes the code required in each component, the additional execution time when doing the communication, and the maintenance needed to maintain the interoperability. In its simplest form, it can be stated as better interoperability requires less effort.

2.4 Exchange Formats

The notion of a mark-up language is not a new concept. The idea was first thought of at IBM in the late 60's as a carrier to enable interoperability between word processors. At the time, the solution was known as the Standardized General Mark-up Language (SGML)

[10]. The idea behind SGML has led to many of the current information technologies in the world, including PostScript, LaTex, HTML, PDF, SVG and XML. These are all *mark-up* languages, languages that allow the writer to add instructions to modify the document, which is stored in plain text. Some of these document extensions include functionality or formatting information [11]. Mark-up languages contain both data and instructions to operate on the data.

2.4.1 XML

XML, eXtensible Mark-up Language, is an SGML derivative. XML contains a data component and a mark-up component. The mark-up component is a collection of tag constructs defined by the user. As a result, these languages are very extensible and easily adapted to multiple variants. The variants, or namespaces as they are called in XML, are also user defined. With both a namespace and a document, we can verify that the document only contains valid mark-up tags.

XML documents can be verified in many ways, but one common method utilizes an *XML Schema*. These are used to specify data domains and document organization. The domains can be as flexible or inflexible as the creator requires. XML Schemas are written in XML and provide a framework to represent data. Since the mapping between Schemas and Documents is stored in the Data document, XML parsers can check the format and data constraints specified by an XML Schema for a particular XML data file when parsing the XML data. Appendix A has an example of an XML document and Schema.

A large portion of the data discussed in this thesis is comprised of two parts—a domain and a data component. We use the GXL schema as one domain both by itself and within the SOAP domain. The third data domain we use is the RSF (Rigi Standard Format) domain. As discussed in Section 2.4.4, RSF is similar to XML.

2.4.2 GXL

GXL (Graph eXchange Language) is an extension of XML through a schema definition. This variant of XML has a strict set of rules as to how the data should be marked up to allow both human and machine readers to understand the data. The exact definition of this language in terms of a document type definition, one of XML's methods of specifying the mark-up, can be found at the GXL website at the University of Koblenz-Landau [12]. GXL represents a typed directed graph which can be augmented with node and edge attributes. GXL also allows for layering through the inclusion of sub-graphs. One of the possible drawbacks of GXL is the lack of attribute definition. This could allow any form of data to be included possibly leading to a problem when application specific data is included in the data.

2.4.3 SOAP

SOAP (Simple Object Access Protocol) is an XML variant which specifies a data transfer protocol for communicating between two applications [13]. SOAP is also an Interface Description Language (IDL) [14]. This is a class of languages which facilitates data transfer between applications. SOAP follows the IDL definition perfectly [14], as it uses a reader, writer, an intermediate format, and is able to be represented using text. The SOAP protocol uses a specific XML format, or namespace, for formulating the message to be sent. This can be seen as the intermediate format, and can also be viewed as text. XML readers or writers could be employed, but most major languages have open source libraries available, which add a layer of abstraction and allows the developer to access a SOAP reader and a SOAP writer. Originally the SOAP protocol also specified that these messages should be sent via HTTP (HyperText Transfer Protocol) [15], but now SOAP messages are transported using a variety of transpher protocols.

SOAP is a technology that enables architects to develop MOM (Message Oriented Middleware) applications. This technology is most often used in business-to-business situations,

but can also be found in client-to-business applications. SOAP is based on the need for client/server or server/server architectures.

Three tier architectures are often used to create web applications. These architectures are best described as having presentation, application, and data repository tiers [16]. MOM applications use the presentation tier to communicate between two application tiers using a protocol such as SOAP. The data is presented in a form which can be easily communicated to another application to use. Each application layer has an underlying layer of data representation. Alternatively, the presentation could also produce a presentation which is human friendly, such as a web interface in a web browser. This architecture is the foundation for e-business, MOM applications and also RPC (Remote Procedure Call) servers.

2.4.4 RSF

Rigi Standard Format (RSF) was designed for the Rigi system before XML became popular. RSF is comprised of two portions—a data domain and a data set. Although the data set does not appear to be similar to GXL, RSF's tuples contain much of the same information that GXL data sets do. RSF is also similar to XML in that it is stored as ASCII text. This allows both formats to be edited manually with a text editor.

2.4.5 SVG

Scaleable Vector Graphics (SVG) is also an extension to XML [17]. SVG uses an XML Schema to clearly define the data content. This allowed multiple players, including Adobe [18] and Mozilla [19] to create SVG rendering engines. Like many other SGML derivates, SVG can be edited either manually or with computer assistance. Although most SVG clients support compressed data, SVG clients are still substantially slower to render images when compared to raster images, but most SVG clients include support for dynamic interaction with the user. These types of interactions, from a users perspective, are similar to an HTML web page.

2.5 Summary

There exists a wide variety of levels of abstraction in the discussion on various issues which have an effect on our prototype. We introduced the notion of high level architecture along with maintenance and interoperability—two of the many areas which could be affected by a system's architecture. Interoperability is introduced as the main theme of this thesis. Maintenance is introduced as many of the choices made for interoperability have similar affects for maintenance. In the upcoming chapters, we investigate how data interchange formats can affect system integration.

Chapter 3

Problem Definition

3.1 Introduction

Reverse engineering has progressed significantly since the first tools were developed in the mid 1980's; we now have very capable tools to help software engineers go about their daily work. So we have to ask, why does a software engineer's tool set not include reverse engineering tool suites? Of the many possibilities, this project is most interested in the relationship between tool adoption and interoperability. Understanding this relationship requires a set of basic requirements for the reverse engineering tool and the tool's interaction with its environment. Therefore our goal is to improve the ease with which tools can enter and exit a user's daily working environment.

Entering and exiting a user's environment is more than simple installation and removal of an application; it involves integration into the user's working environment. We should endeavor to allow for easy application state storage and restoration. Users may also want to customize the tool for their particular needs, while linking the tool to other related tools for a multitude of tasks. This should all occur without losing the rich set of reverse engineering tasks that our tools currently support and help us carry out.

3.2 Tool Tasks

When trying to increase the user base of a set of tools, it is fundamental not to alienate the existing user base in the attempt to create a new user base. Therefore, we should continue to provide tools which perform the current fundamental tool tasks—in our case reverse engineering tasks. Some of the most common requirements for a reverse engineering tool

include the ability to abstract a system's design, organization, and the software patterns used [20]. Other requirements that help the user include consistent, interactive data representations of multiple types [20].

The first requirement affects the data structure—the need to represent software artifacts and the relationships between the software engineering artifacts. A software artifact may be a piece of documentation, a physical structure, or an abstract structure. Therefore, it is possible to relate the documentation for a system to the source files they document. These source files may contain code abstractions, such as objects, methods or data structures. The Rigi model [21] was designed to represent these software artifacts and their relationships. The Rigi model can be easily represented as a set of typed nodes and arcs—or a typed digraph. The various node types represent software artifact types, while arcs represent the typed relationships between the nodes. This can easily be expanded to represent any grouping or abstraction that the user may wish to represent through the inclusion of node and arc types.

This means that our client mental model should resemble a graph. Complications may arise when trying to communicate with other components as these graphs are complex graph encodings. This traversal would represent the storage equivalent, which would allow a direct translation from memory to encoding.

The second requirement affects the design; we should use the *Observer* pattern [22] for our client architecture. The Observer pattern, which is also known as the Model View Controller (MVC) pattern [22] fits particularly well because it "not only separates the application data from the user interface, but allows multiple user interfaces to the same data" [23]. This means that our developers could easily have multiple views of the data, with multiple user interfaces, satisfying the requirements for multiple consistent views of varying types. The Observer pattern also allows for interactive views, as it recommends passing messages to and from the model asking for changes and receiving update notices.

This is important for many reasons, but to us the critical issue is the propagation of these

requirements and how they could affect interoperability. The underlying data model and component architectures could reduce the ability to interoperate cleanly among components if their internal data models and interfaces do not closely resemble each other. This would lead us to believe that time should be spent defining the protocols between the components in the Observer pattern.

3.3 Tool Customization

Tool customization is important because it can potentially improve the interaction between the user and the user interface. Michaud in his M.Sc. Thesis states that tool customization need be capable of personalization and behavior modification of the user environment [24]. These customizations can be completed in a number of ways, including source code modifications to style selections or an end-user programmable interface.

Code modification provides the most flexibility in so far as to what can be customized, but also provides the least amount of flexibility after the product has been delivered to the end user. At the other end of the spectrum, the software system could include a scripting interface which allows the user to modify the behavior or the presentation of the software through script executions. Although scripting interfaces require some degree of expertise with programming, the language used to complete the scripting can be modified for the ease of the end user, making the option more available to them [24]. A pre-defined set of preferences may not be as flexible as scripted interface changes, but they have the advantage that the user does not require scripting skills.

For the past few years, Microsoft has provided both features, thus amplifying the ability of each user to customize their environment. Microsoft Excel is a good example where a scripting interface was designed to simplify the task for the users [24]. As the target audiences for our prototype are software developers, maintainers, and documenters, it is fair to assume a competent level of scripting on the part of the user. Therefore everything other than the basic look and feel of the application could be accessed through a scripting

layer. However, this does not imply that it cannot also be found in the applications' GUI. This is better than creating a new scripting interface because the users will already have an understanding of the interface. Therefore the learning time required would be reduced.

Providing a scripting layer provides some added benefits, such as the ability to build other tools into the existing application, a scripting layer would allow us to "exploit, integrate, and deliver diverse software analysis technologies" [20]. One example of this can be seen in JCosmo [25]. Van Emden in her M.Sc. Thesis extended Rigi to display refactoring artifacts extracted using the JCosmo Java Parser. Rigi also imported JCosmo scripts to execute application specific tasks, such as finding more complex code smells, or generalities in code, which could be refactored [26]. Another example is a prototype of Rigi built using MS Office XP products [27]. In this case, an existing tool is ported to reside within a well-known application, including MS Excel, PowerPoint, and Visio.

3.4 Tool Persistence

Persistence is defined as being "firm or obstinate continuance in a course in spite of opposition" [28], or as in the case of our prototype, continuing execution from the same state as last left off. This is an important ability of any tool, because tools should "support the incremental development of software" [20]. Users have come to expect persistence over the years. This is typically done through information storage in a file system, be it directly or indirectly through another component, such as a database.

Since the concern is with interoperability, and we want to provide persistence to the users, it is necessary to evaluate whether persistence will be provided at a central location, or at the GUI, where the user interacts with the system and could impact tool adoption. Some advantages of a central repository are further explored in the next section, including the ability of two different clients to communicate through the central repository, thus simplifying the interoperability process. The maintenance for either option is about the same.

Maintenance, as introduced in Section 2.2 is generally evaluated either by the number of components or by the amount of code. The assumption being made is that there is a separate module for persistence. This means that there exists a library included and is used through an interface either on one of the clients or on one of the central locations (servers). In either case, there is one library per platform, implying that the only difference is the glue code. Thus, either there is a small amount of code on the platform's server implementation and glue code to communicate with the server, or there is glue code on each client to use the persistency module. In either case they could equate to the same amount of code if written using a high level language. Therefore, both a distributed and a central repository are approximately equally beneficial from a persistence view point.

3.5 Tool Deployment

We should attempt to understand how deployment affects the end users and try to minimize the negative effects of such deployments. For this discussion, it is assumed that the user's machine is in a controlled network, as many work place environments are, and evaluate the two scenarios of a stand-alone application and a client/server suite of applications. We are interested in controlled networks because this would imply that a system administrator exists who is required to conduct application installations on client machines, and most certainly performs the server installations. It is then a fair assumption that installation time does not grossly affect the end user, and subsequently tool adoption, so long as the installation process is well defined. In cases where the user performs their own installations, one goal is to minimize the negative installation effects, such as duration of installation. This favors the client/server architecture as the user would not be expected to install the server application, and presumably the GUI built inside an existing commercial product would be simpler to install than a complete stand-alone application, such as the version of the Rigi tool built using MS OfficeXP [27]. In either case the goal should be to complete the installation process on the client's machine relatively quickly, because "if a user can't use the program in 15 minutes it's useless" [29].

3.6 Tool Interoperability from a User's Perspective

As computer applications become more involved, users want to pass results and data between applications and want the applications to interoperate [5]. As an end user, this allows data to pass between platforms and permits data to be viewed using different methods. It may be desirable to use the extensibility of one tool, such as Rigi, to perform some difficult fact extraction duties before viewing the results in a different viewer, such as SHriMP [30]. This allows users a greater degree of freedom when the user can select a suite of tools rather than a single tool to complete a task.

As a developer or maintainer, interoperability can be used to leverage or reuse existing modules. For example, there is no need to re-create a graph manipulation engine, as there already exists one in Rigi. However, it is necessary to interact with this module. This means that developers can create one component and use it in multiple places, thereby creating tools that appear richer to the end user with smaller development and maintenance costs.

Interoperability also allows for both forms of persistence to be used, either central or local repositories (cf. Section 3.4). Tool customization and extensions would be easier as well, as any two extensions that use the same component to build on would find it easier to communicate. It would then be possible to create better interoperability between these new components.

3.7 Summary

If the problem were to be stated in two words, they would be "component interoperability." There is a multitude of issues when designing software for reverse engineering tasks, and in each case, interoperability is a critical factor. When there is a better understanding of how to improve and evaluate interoperability, then there will be better insight into tool adoption. The reverse is also true—tool adoption can have a great affect on component

interoperability. To understand how to enter and exit the user's working environment, it is necessary to understand how the user interoperates with the elements that are part of the working environment.

Chapter 4

ACRE Engine Prototype

4.1 ACRE Engine Prototype Overview

The lack of reverse engineering tools being adopted is a cause for concern. Some people speculate that tool adoption can be affected by interoperability [2]. While it is difficult to evaluate whether such a relation exists, it is possible to create a foundation on which further investigations can proceed. The question this thesis is most interested in is whether the interoperability of reverse engineering tools can be significantly improved. After careful deliberation over the actions of human users with respect to customization, persistence and deployment, along with the effects that interoperability may have, we believe that the system architecture and the associated design choices affect the interoperability of a system (cf. Chapter 3).

The investigation begins by recognizing there are multiple possible architectures from which a solution can be extracted. It is recognized that system architectures are often a collection of well documented architectural patterns with the addition of some ad hoc reasoning. It is also realized that software architectures are designed to enhance one or more software qualities such as maintainability or persistence. With the advent of the World Wide Web, interoperability is becoming a more important software quality, as evidenced in several recent technologies (e.g., MS .NET, Java Jini, IBM Websphere or CORBA). Therefore, a system was designed which had interoperability as the primary requirement, while including some other functional and non-functional secondary software requirements.

4.2 High Level Architecture

In the early stages of compiling a list of requirements for this study, there were four main goals: cognitive support, minimal user installation, interoperability, and persistence. The ACRE project's hypothesis is that it is possible to leverage cognitive support, while minimizing user installation if we build applications on existing COTS (Commercial Off The Shelf) products. Building on existing extensible applications is possible through the use of their scripting layers and extension hooks, allowing us to provide a series of reverse engineering applications with little or no installation cost, while leveraging the cognitive support already provided to the user through their favorite application.

However, we encountered two problems while developing the ACRE Engine. Each COTS component needed a data source and method of storing the data. Moreover, these components also needed to communicate their data between each other. This implied that either each component had a large number of parsers for importing multiple data types or the data had to be stored in a non-proprietary data format. As it is impractical to include multiple parsers for code and proprietary data formats in each component, we aimed for an alternative solution.

The investigation began by studying how to satisfy the interoperability requirement. This requirement had two options: either all the data is in one format, or there are multiple formats. If there is only one format then only one parser is required per component, but this would require all the components to agree on one non-proprietary format. Each component would then have to implement both a pretty printer and a parser.

The alternative was to implement one parser per format for use by each component. Although some code could be reused, ultimately the code required for each parser would be comparable between parsers and pretty printers because both are translating information from one format to another format. As a result, if there are more than two data formats, each component would have more than a minimal number of parsers. Our system had in excess of four components (i.e., Rigi, SVG, Excel, Visio and Lotus Notes), which meant that

when proprietary formats were used, then there would be a maintenance nightmare with in excess of four parsers for each component, and a high probability of needing more parsers in the future to fully interoperate. This left only one viable solution if the maintenance goals were to be satisfied, that of a single non-proprietary format.

The other problem was how to provide a persistent environment for our client applications. Our situation matched the Attribute-Based Architectural Style (ABAS) [31] [32] for an Abstract Data Repository [33]. ABASs are "architectural styles accompanied by explicit analysis reasoning frameworks [31]. They are frequently used to share design information between system architects to allow for design reuse, providing more predictable system results. We used the *abstract data repository* ABAS. This ABAS is an extension on the *Data Indirection* ABAS [33] where the protocols or interfaces to the data repository have an additional level of indirection between the producers/consumers and the repository. This was particularly relevant for our prototype as we had multiple producers (parsers) and consumers (user interfaces).

We should also note that our system's architecture is similar to that of an Open Hypermedia System (OHS) [34]. An Open Hypermedia System provides hypermedia services in various formats to their clients. Unlike our system, one of the primary jobs for the hypermedia system is to manage the connections with the clients [34]. Ongoing work towards improving integration and interoperability in this community has also found that a similar architecture proved useful in their prototype.

Since we are going to store our data remotely and we wish to have a persistent system, then we must allow producers to remotely load data. This implies that when we store data remotely, any other application that has access to the data, and can interpret the data, can use the data. This is the first step to interoperability, having some client(s) passing data. When we factor in that, all of our clients use the same non-proprietary data format due to our level of indirection from the *abstract data repository*, we have a system in which all the clients can use data stored by any of the other clients in the system.

We now have a remote repository for our data and a single non-proprietary data format. At this time we should revisit our requirements to ensure we have met or exceeded them. Our persistence requirement is somewhat incomplete as we are assuming that short term persistence can be provided by the client application. Unfortunately, this is not an appropriate assumption because some clients operate in stateless environments, such as web browsers. Therefore, a new solution need be found. The other unsatisfied requirement is interoperability. Although we have the foundations for data interoperability there are other types of interoperability which have not been addressed, in particular data, control, and process interoperability.

To provide control interoperability between client applications, we can either perform arbitrary tasks for them or can formulate a method for them to communicate requests between each other. Although the second option is possible, it would require implementing a significant interpreter extension for the requests on each client. If we assume that requests are at least similar to a function call, then the interpreter would be relatively simple, but would still result in code duplication. The ideal situation is to request arbitrary functionality which would require a scripting language or some other advanced form of control representation. This implies an interpreter with complex proprietary extensions, which is not feasible for all our client applications. This solution would result in multiple user libraries which complete the same functionality. For example we would require a JavaScript extension for SVG and a VBScript extension for Excel. Therefore, we decided to provide a service with which all the clients can communicate and perform arbitrary tasks. Just like data interoperability, a communication common format will be required. Since we are performing arbitrary tasks remotely, we can also provide persistence requirements at the same time.

We now have two external components in our client application, both of which have data concerns. We chose to represent our architecture as a series of layers. The first layer is the consumers (user interfaces) and producers (parsers). The second layer is the layer of indirection between the clients and the repository, this is our ACRE Engine. The ACRE Engine also performs data manipulations to increase the interoperability between all the

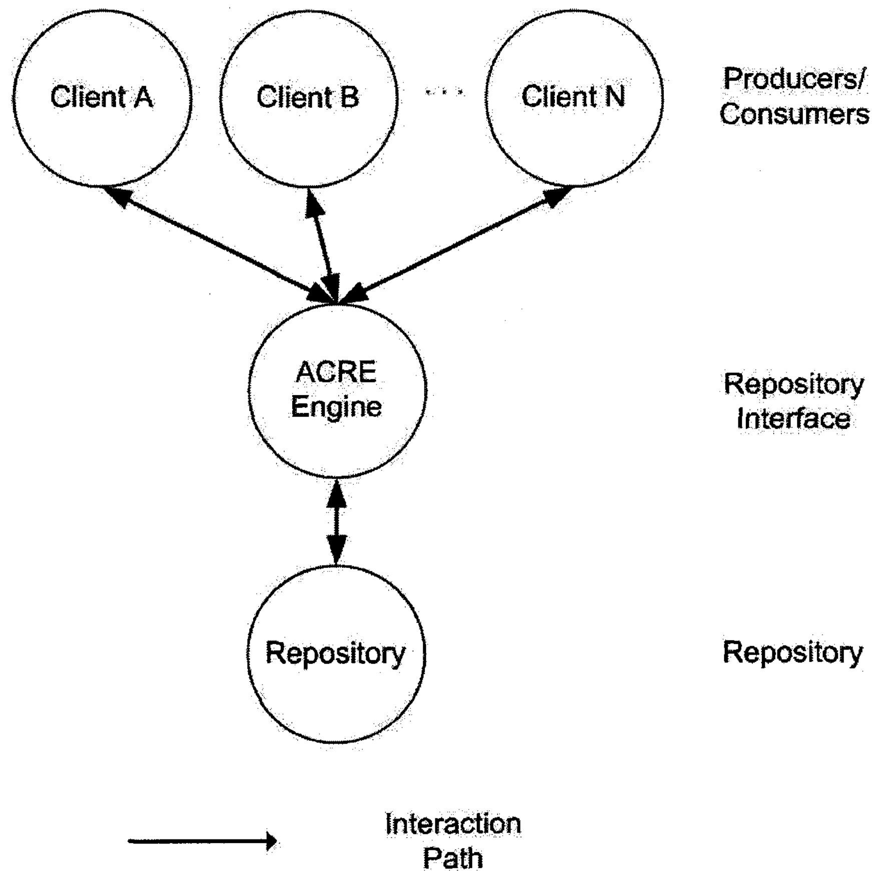

Figure 4.1: ACRE Prototype System Architecture

components in the system. The third and last layer is the data repository. This resulting three-tier architecture, is shown in Figure 4.1.

Our repository options have been limited to a single location with a uniform data format which is shared with the ACRE Engine. We have many options of how to implement our repository, but the simplest solution is to use flat files. As we are most interested in the interoperability with the prototype, we selected a simple storage format. In the future, a faster, more robust form of persistence, such as a database, can be implemented to replace the flat file system. The data repository was further simplified through the reduction in request sources, such as other reverse engineering application, to only accept requests from the second tier.

This architecture has also the potential to improve interoperability if we allow the middle tier to be customized, giving the client applications an opportunity to add functionality, while sharing this functionality with the other client applications. An example would be a short routine which removed unwanted nodes which followed a particular pattern, such as nodes representing files. This would improve the control interoperability of the system through functionality reuse and communication of sets of action requests.

The last large gap remaining is figuring out how the clients will interact with the second tier? We have multiple platform independent client applications which need to communicate with one application that is not necessarily on the same machine. This requires some network protocol to pass data manipulation requests and data. To keep maintenance costs to a minimum, we began looking at existing technologies which could be employed to perform the required transfers. Of all the technologies we investigated, the one which worked best with web applications, yet still had adequate support for other COTS tools such as MS Excel was the Simple Object Access Protocol (SOAP) (cf. Section 2.4.3). As SOAP is defined using XML as an XML Dialect, it was easy to use GXL (cf. Section 2.4.2) as the data transfer protocol. Although there are other text based data formats, the main attraction to an XML based format was the easy acquisition of parsers, which again significantly reduced the work required as well as a suite of existing data transformation facilities such as the eXtensible Stylesheet Language Transformations (XSLT) [35].

4.3 ACRE Engine

The architecture for the whole system has three tiers: the client application, the data manipulation application, and the data repository. The data manipulation portion (ACRE Engine) is required to satisfy several requirements including:

- Communicate control data among all the clients
- Provide short term and long term persistence

- Provide an interoperable environment
- Complete basic reverse engineering tasks such as searching, sorting, transforming, and modifying data
- Extensible to allow the client applications to easily complete customizations for their needs
- Share newly added functionality with the other applications by means of scripts

The ACRE Engine was built as a web service inside IBM Web Sphere because Web Sphere was the most prominent COTS product which supported SOAP. We considered two other viable systems, Microsoft .NET and Apache. We did not select Microsoft .NET because when we were first began our development, documentation for MS .NET was not yet available, and it was not clear when it would become available. In Microsoft's defense, at that time the product was new and we were using a trial version. While Apache's software was well documented, it was overly complex to use. We found that while Apache and Web Sphere both provided the same services, Web Sphere's interface was much easier to use, and development was faster because of the automated features included, such as the Java classes generated from the Web Services Description Language (WSDL) file. Therefore, we created the ACRE Engine using Web Sphere. We should also note that cost was not factored into the equation because all three manufactures provided affordable scholastic versions to our research group.

When we began the design of the ACRE Engine, all of the engine's actions were classified into four distinct groups, communication with the clients, persistence, data representation, and data manipulation. This classification resulted in the four major portions of the ACRE Engine, as shown in Figure 4.2. We should note that the modules communicate with most of the other modules creating a tightly knit solution. This required that our solution had well-defined interfaces among the modules to create a level of indirection between the modules. This will hopefully prove invaluable for future maintainers of the ACRE Engine, as it is expected to reduce maintenance costs.

The user gateway module communicates with the clients and acts as the main control module for the ACRE Engine. This module interprets all incoming requests from the client applications to the ACRE Engine, buffers the data received, and passes execution on to the appropriate module based on the request. This module also formulates the responses back to the client applications. The system's data representation is completed in the memory module. This module provides an in memory representation of the data to be transferred, stored, or manipulated. This data structure and its associated access routines are used throughout the ACRE Engine. When the Engine is requested to store data to provide long term persistence, requests are passed on to the data gateway module. This module acts as an interface to the storage repository. Data manipulations performed by the ACRE Engine are conducted within the scripting module. The scripting module can execute scripts as short as a single manipulation command, or as long as a complete program. The script interpreter was extended to include the data manipulation actions to maximize system interoperability. This allows all actions to be scripted, providing the ability to log the actions leading to a particular solution.

4.4 ACRE Engine's User Gateway Module

The User gateway is the module which controls the communication into and out of the ACRE Engine. This module is responsible for the protocols used to interoperate with the client applications and control the flow of execution through the ACRE Engine. When a message is received this module interprets the request and forwards the request on to the appropriate module.

The first method to access the ACRE Engine was the SOAP interface, which allowed many applications to communicate with the ACRE Engine. Unfortunately, some clients such as the SVG (cf. Section 2.4.5) client, could not communicate using SOAP due to transmission limitations. Communication from the SVG client is limited to sending messages using the *GET* and *POST* parts of the HTTP [15] protocol, excluding direct SOAP communication.

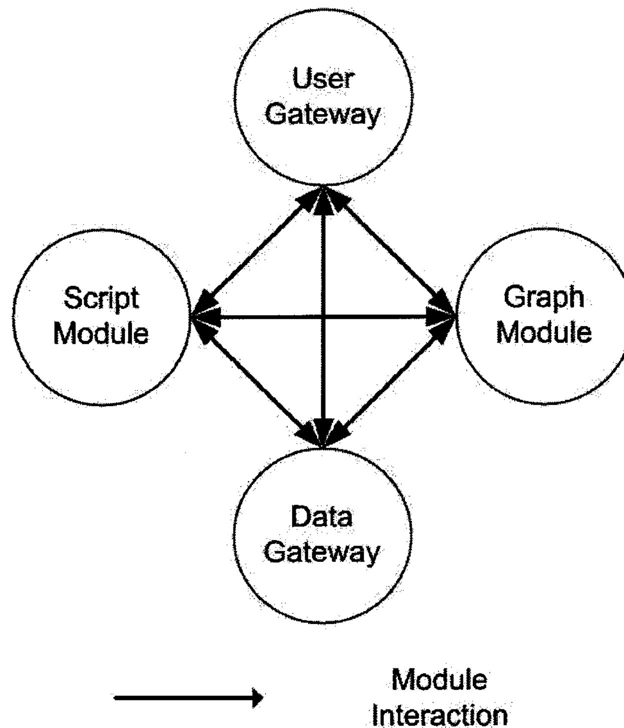

Figure 4.2: ACRE Engine Architecture

Our solution for the SVG client involved creating a second, nearly identical, communication protocol. The only significant difference between the protocol used for the SVG client and SOAP is the method in which the data transfer mechanism. We used the ability of SVG to communicate through the HTTP protocol to pass data to the server, and the response was caught by the SVG client allowing for data and requests to be communicated. To preserve a notion of one logical interface, we simply embedded SOAP messages inside the form submissions and replies. Although this required some extra code at the client application, the maintenance costs were minimized, requiring little extra code in the ACRE Engine.

Many SOAP web services currently available are listed in a Web Service Definition Language (WSDL) file (Appendix B). Web Sphere takes an instance of a WSDL file and generates all the code from the point where the interface hooks to the developer's application start to the point where the messages come and go to the Internet. From a developer's

point of view, defining a SOAP web service is the same as defining an interface in Java. Each action has an output type defined, along with the input parameters and their types. The resulting hooks are in the form of a Java class where it is only necessary to add the body to each method, where each method pertains to one action. The WSDL file developed for the ACRE Engine is listed in Appendix B.

After the WSDL file is completed with the assistance of Web Sphere, Java code is generated by Web Sphere. The next step involves the developer linking Java code to the requests so as to complete the actions requested. These mappings and the WSDL definitions are the two main portions of this module, creating all the links and directives for the actions requested.

The SOAP interface defined by this module provides the option of importing, exporting and listing all existing graph, script, or blob data. The graph data requires a domain to be specified, and the request is completed if the domain is known. Script data must be written in Tcl [36] and have an ID associated with it. Although we considered other scripting languages, including JavaScript, we selected to use Tcl as our scripting language. The reasoning associated with this choice is explained in Section 4.7.

Blob data is all other data which may be proprietary and may be shared with scripts or client applications using similar COTS components which can use the proprietary format of the particular data blob. There is also an option to execute script commands which allows the client application to communicate their customizations. Some of the persistence requirements are satisfied by providing functionality for the client application to read or write temporary state information. These functions store the information for the duration of the network connection. We should also note that all the operations mentioned can be completed within a script, so there is more than one method of completing tasks after a request is received from the client application.

4.5 ACRE Engine's Data Gateway Module

The data gateway is similar to a driver, or interface to which data can be sent and requested. This module connects to the data repository. In this case the data is stored in a file repository. The data gateway accesses the data files in the repository when a file needs to be read and written. This module cannot send requests to any of the other modules and does not accept requests from the scripting module. This extra level of indirection forces the user to explicitly read or write data avoiding accidental loss of data through implicit document replacement.

Although at first glance a file based repository did not appear to be the best solution, we eventually chose this solution because of its simplicity. While better solutions exist for data repositories, our focus was on the ACRE Engine. The only noticeable effect this choice would have on the system would be slower performance when storing or loading a graph.

We investigated methods of representing the data in the file repository. The options were limited, falling into two basic categories—ASCII [37] and binary. Binary storage would have been simple to implement, storing the memory version to the file using the java.io libraries. Unfortunately, this option would not work for all data types. While blobs are well suited as binary data, scripts and graph data are text based. For these two data formats, our view was that textual data storage would have valuable debugging benefits. In particular a second viewer would not be required for the system administrator to browse the raw data.

ASCII or Unicode data is about as broad a data format as binary, and some thought must also be given to how to structure the data in the repository. The logical solution is to continue to use XML as our base for data representation. There are other good alternatives for representing graph data including RSF (cf. Section 2.4.4). Fortunately, we had already constructed routines to read and write the graph data as GXL (cf. Section 2.4.2). Therefore, for reasons of simplicity and maintenance, GXL was used as our primary storage format.

One issue was where to place the presentation data compared to the model data. This problem arose when we realized that each of our clients would be using the same type of graph model, such as nodes and edges, but they may not all need presentation data, such as node location on the screen. For our prototype, the presentation data was left with the model because a large portion of the model data is required to reproduce the representation data. By choosing to maintain the presentation data with the model data, we were introducing the possibility of reducing the interoperability within the system, but we were also avoiding a large amount of duplicate data. We should also note that should a user only want the model data, the model can be easily extracted from the data stored. The additional presentation information fits within the GXL domain and we stored the presentation data as attributes on the entities from the data model, such as node or edges.

Although GXL is not a silver bullet, it goes a long way to helping resolve some of our issues, mainly because it is based on XML technologies which are widely used and supported. When we used GXL, we were vigilant not to abuse GXL or try to stretch its capabilities by including extensions past any reasonable limits.

4.6 ACRE Engine's Memory Module

One requirement for the system was to quickly and efficiently manipulate the reverse engineering graphs. This requirement could have been satisfied by modifying the data directly in the data files. However, this would not have been efficient as several slow passes over the file would have to be made before even a simple addition could be completed. This meant that the data should be loaded into memory. The only question was in what form. We decided to look for available parsers before making our decision.

There are basically two kinds of XML parsers which are freely available for the public to use. The DOM (Document Object Model) [38] parser, which parses the data into an instance of the DOM in memory, or an event driven parser. The second option does not leave a tree in memory, but rather provides events as the parser traverses the data tree in the

file. This is the SAX (Simple API for XML) [39] parser.

The question arose as to whether we wanted to use the DOM to represent our data or could we achieve a better data representation in a new model. Our view was that it would be cumbersome to use the DOM as our data represented graphs which had to be traversed, and the arcs between nodes were difficult to navigate using this model. As a result, we chose to create our own model of the data. This meant the SAX parser was simpler to use, as it was easily extensible to our needs.

In future similar implementations I would suggest using an additional layer of indirection between the data module interface and the instance data to improve scalability. This would allow for a dynamic interface to a database, providing support for very large data sets unable to fit in memory.

The resulting model followed the GXL graph definition closely, but allowed for easy traversal of the graph as the text based links were now memory model links. To improve efficiency, the model was implemented using hash tables arranged in the form of a tree (cf. Appendix F for a code Sample). This memory structure is also depicted in Figure 4.3. We also include a list of references to the edges alongside the list of nodes in the parent graph object to improve searching for particular attributes of edges.

4.7 ACRE Engine's Scripting Module

To allow the ACRE engine to be extensible, it was necessary to allow the end users to create complex data manipulation routines. The user or client application may wish to create new routines, or modify old routines to reduce the effort required by the client application. In some cases client applications may need assistance completing complex reverse engineering tasks, resulting in a simpler solution when the ACRE Engine is extended. Client applications may also choose to complete complex tasks at the ACRE Engine to reduce the maintenance costs, having only one copy of a task for multiple possible client applications.

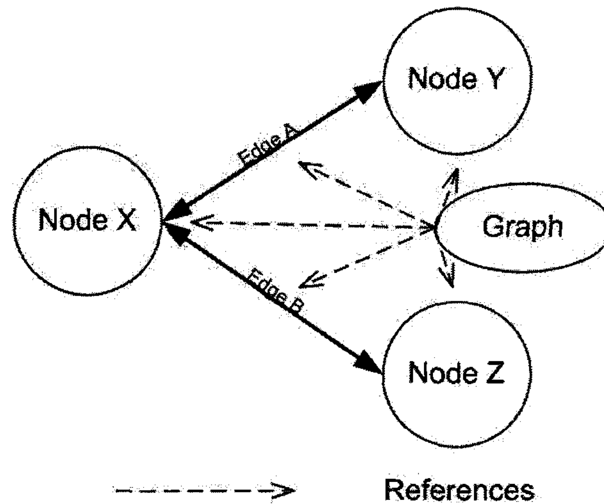

Figure 4.3: ACRE Engine Memory Model for a Graph

Creating and modifying the actions performed by the ACRE Engine is achieved by providing an interface to a scripting language extended to include data manipulation commands. We selected to implement a minimal set of commands based on the core actions required to perform the basic reverse engineering tasks as set out by Wong [20].

The TCL extensions listed in Appendix D fall into five distinct groups. The first group of extensions allow the user to modify the existing graph in memory. Another group of extensions allows the user to annotate the graph in memory, while the third group of actions provides graph persistence. The fourth group provides for script extensions and persistence, while the last group of extensions provide direct short term persistence.

When selecting our scripting language, there were two main requirements: the language had to have an easily extensible interpreter which could be used in our system, while maximizing code reuse. Suitable interpreters for both the JavaScript language and the Tcl language are easily available as open source Java implementations on the Web. The Java implementation was important as Web Sphere Web Services are often written in Java. To select between the two languages, we looked at some of the advantages of each language. Tcl had the advantage as we already had a large collection of Tcl reverse engineering scripts.

The other benefit in using Tcl was to remove one possible variation for our evaluation when we compare with the Rigi system.

The Tcl Interpreter consists of Jacl and TclJava, two open source packages from Sun Micro Systems [40]. These packages allow text to be interpreted as a script and store the results for the remainder of the session. The packages can also be extended to include reverse engineering features [20]. Besides the reverse engineering features, we also included some persistence options to the language consistent with the ACRE Engine requirements, as well as graph model manipulations, which are external to the Tcl interpreter. A complete listing of the Tcl commands and their parameters can be found in Appendix D.

As Tcl scripts are represented as text they are human readable. As with GXL data, the Tcl scripts can also be abused, and large amounts of data could be embedded in the scripts. To discourage this, Tcl scripts are passed as attachments to a SOAP message, allowing other data to be included and used for execution. Both the scripts and data can then be stored on the server for immediate or future use. It should be noted that the data passed does not need to be text, but can contain proprietary data formats. This was a difficult decision as it reduced the ability to interpret some of the stored data, but provides better extensibility for user applications.

4.8 Summary

This chapter documents the overall structure of the ACRE engine and many of its design decisions. We presented the main components of the system, including the scripting portion and the memory model. We also outlined how the components inter-relate within the system and how the external communication protocols are involved. We also described selected external interfaces in detail. In particular, how one protocol mapped to the other to provide better data management. Data storage and transportation formats have been discussed. We have also outlined the reasons we selected Tcl as our scripting language. Thus, this chapter showed how the ACRE engine provides persistence, extensibility and

interoperability.

Chapter 5

ACRE Interoperability

5.1 Introduction

To evaluate whether the ACRE system has better interoperability compared to other similar systems, we classify different types of interoperability. This allows for an objective comparison between two independent systems, showing each system's strengths and weaknesses. In Section 2.3, interoperability was defined to be essentially the ability of two systems to communicate. This chapter investigates how to integrate three types of communication: data, control and process information.

The ACRE Engine also uses two distinct mediums; therefore an investigation is needed into both types of Control Integration—Web Integration and Service Integration, alongside the other two related forms of integration—data and process integration. For each of the types of integration, we present an explanation of the integration type and how each type of integration relates to the ACRE Engine.

5.2 Data Integration

Data integration refers to the extent to which applications can share, or use each others data. Data integration deals with ease of passing data between systems and the quality of the data passed [41]. The ease of this integration is often a measure of the similarity of their internal data models and external data representations. The more similar two systems are with respect to data representations, the easier it is to pass data between the systems. Data quality refers to the ability to derive data within systems from a combination of persistent (stored) and non-persistent (runtime) data. In particular, data quality is the extent to which

the data is replicated and whether the data is consistent between applications.

As discussed in Chapter 3, the most common solution to providing data integration is a central database or repository where all the data is stored [42]. Unfortunately, this does not completely solve the problem as there may be non-persistent data which need be passed between systems to maintain data consistency. The solution is to find a common communication protocol that does not require all the data to be passed through a central repository.

The ACRE Engine does just this—it provides non-persistent data integration alongside a persistent data repository. This is achieved by dividing the data into three categories—well-known persistent data; unknown persistent data, and session data. The well-known persistent data is stored in a repository and can be loaded into or stored from the session data portion of the ACRE Engine. The unknown data, which is often presentation dependant information from a client system, is stored as an information blob in the repository without manipulation. Session data is stored in memory and can be either well-known data or unknown data.

The most important part of data integration that occurs in connection with the ACRE engine is the ability to pass data between the persistent portion of the system and the non-persistent portion. This allows the users to provide many useful functions, such as primitive version control and undo capabilities. These functions provide a high level of data integration between clients as data from multiple clients can be passed seamlessly through the server, independently of the type of persistence required.

Data integration can benefit the reverse engineering community in many ways. One example includes two developers who cooperate together on an open-source project. If one of the developers worked using a Visio client and the other using a Lotus Notes client, then both the developers could share data. But if the two clients were unable to read and write a common data format then data integration would be more difficult.

5.3 Control Integration

Control integration is the ability of one system to request another system to perform some action [41] [43]. Effectively, systems provide services to other systems and request other tools to provide services to them. The underlying implication is that the systems communicating requests actually have the ability to communicate these requests. In this respect, control integration is similar to data integration and could even be perceived as using data integration to achieve control integration. Control integration differs from data integration in that it implies that an action is taken by the recipient of the request.

Like other forms of integration, these are two systems integrating, the service provider and the service consumer. To understand each system, it is necessary to determine what the system is providing and what it is using. Requests between systems can be arranged in many ways, but two common forms are web integration and service integration. These forms are supported by the ACRE Engine.

5.3.1 Web Integration

Web integration occurs when control integration is applied to Internet data presentation [44]. This involves taking data of some form and translating it into a format that can be accessed by a web client. Since these are web-based systems there is the problem of contending with a stateless environment. These systems can use many well known technologies, such as SVG, HTML, or JPG.

In our reverse engineering scenario, a web browser was used to display combinations of SVG and HTML data. Responses to the server were in the form of web requests, where form data was passed to the server. This allowed a simple protocol to be developed that passes control between the server and the client using existing communication protocols.

The SVG client, which can be seen in Figure 5.1, has two main components. Like other SVG applications, this application is comprised of a set of client side scripts and presen-

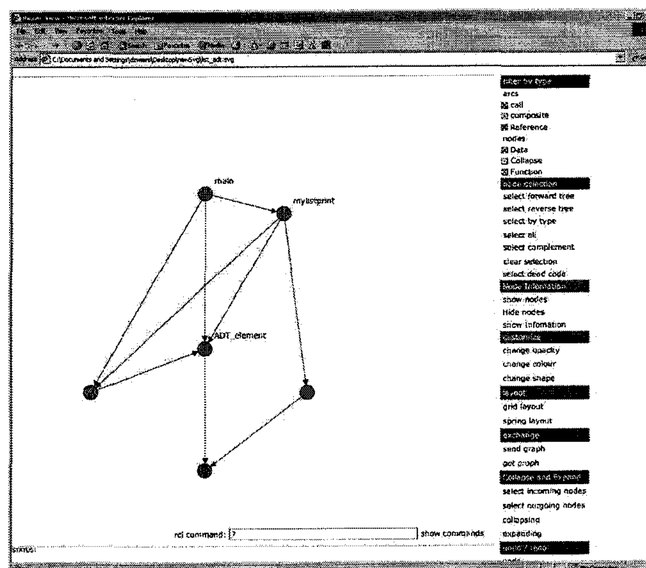

Figure 5.1: SVG Client Application

tation information. Much of the scripting in this application is for user interface manipulation that is not of particular interest for the ACRE Engine. These components of the scripts, which are of interest, perform communication between the ACRE Engine and the SVG client.

The communication between the ACRE Engine and the SVG client is accomplished by sending server requests in the same way as a web page posts data to a server. The scripting which enables these actions can be seen in Appendix E. This code is of particular interest because it represents the simulation of a SOAP client within the SVG client. In the ACRE Engine, a servlet receives the SVG client's generated request and passes it on to the same set of routines that the SOAP requests access. This means that the ACRE Engine's communication behavior remains the same even though a second communication protocol is introduced.

Web integration would help our open source developers from the earlier scenario. Suppose both our developers had collected a version of their reverse engineering data which they could share effectively. They each had important presentations in the near future in their

respective organizations, and wanted to share some presentation information. In this case they could both use a web client, such as the SVG client, to collaborate on the presentation of the data while still allowing them to both manipulate the data using their own respective clients.

5.3.2 Service Integration

Service Integration occurs when Control Integration is applied to a system which revolves around the idea of being a service provider [44]. Essentially, each component of the system can ask other components to provide a service, provide a service to others. This differs from simple Control Integration in that these requests use limited predefined protocols in which asynchronous messages are passed between users and providers, while Control Integration also allows synchronous messages. Service Integration is typically completed between two systems, as compared to Web Integration which provides presentations to end users (that is integration between multiple systems).

The ACRE Engine uses the SOAP protocol to provide system-to-system service integration. An interface is available to request the engine to provide and/or manipulate data (Section 4.4, Appendix B). This system also provides the ability to have/maintain states for multiple session lengths, where the transition of state from moderate to long term state must be completed explicitly. In Chapter 4, the ACRE Engine was described as communicating its requests using Tcl scripts (Section 4.7) passed in the body of SOAP messages, and the scripts are executed in the ACRE Engine. The scripts have been extended to allow access to the inner state of the ACRE Engine and its persistence functionality.

The connection to the ACRE Engine through the use of SOAP is similar to calling a system library, except the calling system requires two things. First, the system can connect to the SOAP provider. Generally this implies that the client is either the same machine as the SOAP provider or both the client and the SOAP provider can communicate using a network connection. Second, the calling system has a method of reading and writing SOAP

messages. This is generally achieved by using one of the commonly available SOAP connection utilities. The SOAP connection tools provided by Microsoft are used with Visual Basic in our MS Excel client. The SOAP connection and execution code for Visual Basic Scripting that was used in the MS Excel client is listed in Appendix C.

The ability to use a service remotely is once again useful in our open-source development scenario. This allows the developers to share functionality between their two distinct clients. Recall that one client was using Visio, while the other was using Lotus Notes. Normally it would be difficult to pass functionality between the two applications, as a script written for Visio uses Visual Basic while Lotus Notes scripts are written in LotusScript Extensions. In this case both clients can share functionality on the ACRE Engine as a script which is executed as a service on the server.

5.4 Process integration

Process Integration is different from the other forms of integration considered thus far in that it works on a much more global scale. Process integration considers how to traverse from one task to another in a user's daily routine easily [41] [45]. We deal with the transition between a user's set of tools for one task and the next set of tools. It should also be noted that while the same tool may be used in both cases, it is still important to look closely at every transition as some tools may not be used in the same way or for the same tasks.

The ACRE Engine has included some tools to facilitate process integration. One contributing factor is the ACRE Engine's ability to manipulate data through a scripting interface. The result is a system where scripts can be stored and run to assist in the transition between phases in a process. Scripts can also be run to transfer data between two individual processes. The ACRE Engine has not experienced any great difficulties communicating with other tools. As such, it could provide a solid foundation to provide process integration either within a process or between multiple processes.

Process integration can also help our open source developers. In this case the developers could benefit from routines which take some of their hard work from the design process where they painstakingly defined their design in a data form which could be integrated into Lotus Notes or Visio. Then after completing a portion of the implementation process, a script could be run to parse the resulting implementation and validate the implementation against the proposed design.

5.5 Summary

Throughout this chapter we have looked at ways in which the ACRE Engine can integrate with other systems, both at high levels of abstraction and down at the technical operational level. The ACRE Engine has been shown to have a clear and robust data definition which allows it to integrate with multiple data clients. The ACRE Engine could be controlled from multiple clients, some over the web, and others using control messages to access the services provided. Finally, it has been shown that the scripting provided by the ACRE Engine provides a base on which highly sophisticated process integrations can occur.

Chapter 6

Prototype Evaluation

6.1 Introduction

To investigate how to improve interoperability, it is necessary to be able to evaluate interoperability in some way. In this thesis, we evaluate our prototype by comparing it to the Rigi system with respect to interoperability. To compare the interoperability of the systems, we compare the two systems according to the integration discussed in Chapter 5. We can complete our evaluation based on the integration of a system because interoperability is automated integration [5] [46]. This provides a clearer understanding of where one system is better then the other system in terms of interoperability.

6.2 Data Integration

The Rigi system has many data integration features, most of which stem from its proprietary data format, Rigi Standard Format (RSF) which is an ASCII based format storing the graph data in tuples. The format's simplicity allows any text editor to read and modify the data in its native format, but even more importantly, allows individuals to automate data manipulations. RSF is similar in structure to other ASCII-based interchange formats, such as XML, because each RSF document is comprised of a domain model and graph data.

One such manipulation was the creation of a GXL (cf. Section 2.4.2) to RSF and RSF to GXL converter. This involved creating a mapping domain and mapping of data between these domains. The mapping proved relatively easy because the GXL Schema provided all the required data in the RSF C++ Domain. During the development of this automation, the developer found it relatively easy to produce a printer and modify parsers for the

two formats. This was mainly a result of both formats being ASCII based, with a similar conceptual data structure (i.e., graphs).

One of the limiting factors of the Rigi system is its inability to provide data to other applications directly on-the-fly. This can be overcome by using the built-in Tcl functionality through polling data files, but native support is not available. The reason for this deficiency is the stand-alone architecture of the original Rigi system. It was not designed as a callable engine.

The ACRE Engine is substantially different from the Rigi system with respect to data integration because the ACRE Engine was designed with data propagation in mind. It uses a similar, although possibly more common, data format. Unlike Rigi, the ACRE Engine is not a stand-alone application, and was designed to pass data to other applications for other applications to use. This resulted in the ACRE Engine having a more reliable, robust method for propagating data through SOAP messages to other applications as compared to Rigi's method of polling data files (cf. Section 2.4.3).

6.3 Control Integration

The ability to integrate control between applications can be affected by some architectural decisions. The Rigi System is no exception. By creating a stand-alone application, control integration was not a primary requirement in the system's design. This resulted in a system in which incoming control requests are stateless. The exception is when the built-in Tcl interpreter runs scripts that provide loose connections to other applications, mostly through polling applications.

6.3.1 Web Integration

The Rigi system does not have the ability to be directly accessed from outside applications, except through polling. However the Rigi System does accept state free requests to perform

selected operations. This ability was used to create the Reverse Engineering Notebook [20]. In this case a web server application made requests to the core of the Rigi system to perform fast data manipulations for web displays. Although the system did not contain any form of session persistence, it did execute as intended and provided a seamless web interface to reverse engineering data.

At first glance, the two systems appear similar and indeed there are many similarities. However there are also some important and subtle differences. The most significant differences are that the ACRE Engine has the ability to maintain state in a multitude of ways, and the presentation of the data is separated from the action requests. The ability to maintain state allows the ACRE Engine to perform operations on the results of the previous result, as well as maintaining intermediate data between requests. The division between presentation and request allows the request to either manipulate the presentation or perform requests, all using the same request interface.

6.3.2 Service Integration

As has been already mentioned, the Rigi system cannot be accessed directly from other applications through a connection which retains state. However the Rigi system can fulfill stateless requests from other applications, or can run in the background communicating through a polling routine written in the scripting interface. This means that there are two options for sending requests to the Rigi system to have it perform some action(s) for the calling system. Although both options have some obvious limitations, this functionality provides basis to build service integration protocols between the Rigi system and another system.

Creating service requests in the ACRE Engine is much simpler and more robust than the Rigi system, but does not provide any additional communication options that could not be otherwise completed using the Rigi system. For example, an application may either create a proprietary format and parser to pass requests through polling to the Rigi system. The

same requests could be formed into SOAP requests and passed to the ACRE Engine. The only difference is the ease with which the communication can be made, the ACRE Engine uses the SOAP protocol, a well supported and documented standard.

6.4 Process Integration

To perform effective process integration it is necessary to be able to help move the project from one stage to the next. This means that at times a tool may perform multiple different tasks, some of which may not have been the tool's originally intended purpose. Rigi has proven to manipulate graph data easily for many applications, as well as multiple views, providing a versatile graph display tool. Although graph data representing health or aeronautic systems do not usually have similarities with reverse engineering data, in fact all three data types may share common processes. For example orphan detection algorithms may be applied to all three data sets, sharing one process across multiple domains.

The data manipulation is best performed in Rigi through the use of the Tcl scripting interface. This allows for the customization of Rigi's actions to achieve the data manipulations required to pass data from one process to another. After the first instance of these conversions, it is often possible to reuse part or all of the scripts to accomplish similar future transitions.

The ACRE Engine has a similar set of capabilities compared to the Rigi system for transitions between processes or stages within a process. Both systems can execute scripts and reuse the scripts in future cases. Both systems work with generic graph data, implying that the meaning of the graph data is arbitrary and manipulations can be completed to reorganize the data based on the meaning of the graph data. In the context of process integration, there is not a significant difference between the two systems.

6.5 Summary

The comparisons of the various forms of integration between the Rigi system and the ACRE Engine have provided a comparison of the systems with respect to interoperability. In general, the ACRE Engine appears to have a simpler method for achieving the integrations. Thus, it is fair to say that the ACRE Engine is superior with respect to interoperability. In most cases, the Rigi system has incorporated customized forms of integration after the design phase. As a result, the Rigi system has more complex protocols and methods of achieving the integrations. The communication protocols for the ACRE Engine were designed with standard integration mechanisms and with system integration requirements as a high priority. As such, most solutions for various forms of interoperability have proven to be simpler and more robust with the ACRE Engine.

Chapter 7

Related Work

There is usually more than one way to solve a problem. We briefly investigate some options to discover some of the major differences in their solutions. This chapter presents selected solutions in the field contrast their approaches to our approach with the ACRE Engine.

7.1 JGraphPad

As the ACRE Engine communicates using GXL data, it would be appropriate to consider tools such as JGraphPad, which display GXL data. JGraphPad is a graph editor for GXL data. It does not have any extra functionality, is not extensible, and does not provide any interoperability to other applications beyond the GXL data exported to the file system. But this tool also has a library which contains a subset of the graph display functionality in a set of Java files which can be embedded in any other Java-friendly application. JGraph is used as a graph visualizer in ACRE Notes, our Lotus Notes client application [47].

7.2 Creole

While SOAP is used in the ACRE Engine to allow interaction with other system components, there are other methods available. Creole is the result of integrating SHriMP [30] with Eclipse [48] without using SOAP. Creole uses the plug-in architecture of Eclipse to integrate SHriMP with the Eclipse development platform. If the only client was the one in the Eclipse platform, then the plug-in architecture should be a suitable solution, as is the case with Creole. Unfortunately, our requirements include support for multiple clients, which require a more universal method of providing interoperability. Had a plug-in architecture

been used, it would not have reduced the need for a central registry available to provide each client a data repository and application extension. This could allow the client the data and extensibility option, but would still require some form of central repository. It is still necessary to have some form of central repository with a communication protocol. This is similar in nature to the ACRE Engine, except it does not satisfy our reverse engineering requirements.

7.3 TXL

TXL (Turing eXtender Language) was created to facilitate rule based transformation [49]. TXL is similar to XML and RSF in both its ASCII representation and the independent domain and data portions. Although created 15 years ago, TXL performed many of the tasks which XML transformation (e.g., using Extensible Stylesheet Language Transformations (XSLT)) can perform. The only difference is that XML works with XML, while TXL works on any text files that can be interpreted using a set of pre-defined rules.

The implication is that TXL can be used to create parsers for reverse engineering tools, as well as translators between reverse engineering tools. Jin states that the solution to interoperability is to create a schema to which all reverse engineering tools can translate their data, reducing the number of translators required for n systems down to $2n$ translators from $n(n-1)$ [50]. This idea is similar to the notion in the ACRE Engine, where the ACRE Engine uses one format to pass its messages and contains one set of parsers per domain. However, the common schema for the ACRE Engine is based on a memory model which represents the data.

Both the ACRE Engine and TXL share the same conceptual notion of how to improve interoperability. The only difference is in their implementation. Memory was too expensive to consider creating the intermediate schema in memory for TXL.

7.4 PBS

The Portable Bookshelf (PBS) has similarities to the ACRE Engine in that it is a service provided over the Internet. Unfortunately, this service only provides information to its web-based user interface. The data is passed in the form of HTML data and could be used to extract software engineering facts from the presentation information. Both systems are web based and serve text based data to the clients. The major difference is the level of persistence—PBS has a state-free system while the ACRE Engine does have state in its environment. This difference can help improve the ability of two programs to communicate as the previous state does not need to be explicitly re-created for each transaction.

7.5 BOOST

Broadband Object-Oriented Service Technology (BOOST) [51] focuses on creating development environments for network services. The BOOST project created a framework to assist with tool integration. In this framework each tool is wrapped to allow for a common communication protocol.

When compared with the ACRE Engine, the BOOST framework falls short. BOOST does provide data integration support through an entity-relationship data model. Unfortunately the BOOST framework does not directly support control integration [51]. BOOST does provide some process integration, but is substantially less as one of the BOOST framework's assumptions is that the tools play a passive role.

7.6 IPSEN

The Integrated Project Support Environment (IPSEN) [52] relies on precise tool specifications using an extended graph model. The graph model allows the designer to abstractly specify data transitions in a common format. These transitions can be interpreted to gener-

ate a framework around the project to improve interoperability between the system components.

When compared with the ACRE Engine, IPSEN has some similarities. Although both projects have a single data format and an associated schema, IPSEN's schema cannot be easily changed at runtime. It is also unclear as to whether IPSEN supports a distributed environment, as it was designed to generated glue code for software development environments, which have been to date mostly single user applications. IPSEN does provide for process integration and extensibility at design time through the additions of relevant nodes to the graph. IPSEN also supports control integration by defining transactions across the graph model, but like data integration it is unclear whether this could scale to a distributed system.

7.7 Summary

This chapter discussed research projects related to the ACRE Engine. We tried to explain where the ACRE Engine improves upon an existing tool. We have shown these improvements alongside the benefits of the existing technology to understand where this technology may be better suited.

Chapter 8

Conclusions

8.1 Summary

The goal of the Adoption Centric Reverse Engineering (ACRE) project is to investigate how to leverage cognitive support and how to improve interoperability in reverse engineering tools. The assumption is that improved cognitive support and interoperability lead to better tool adoption. In this thesis, we investigated interoperability issues in reverse engineering tools.

Our investigations began with the definition of interoperability; the ability of two or more systems or components to exchange information and use the information that was exchanged [5]. Our next step was to recognize different characterizations of the types of interoperability found in many reverse engineering system. To accomplish this characterization, we investigated issues associated with each type of interoperability. We proceeded to consider interoperability at multiple levels of abstraction—from the system architecture to the technology selected. This allowed us to begin our goal of creating a prototype ACRE Engine.

One of our goals was to create a measurably more interoperable prototype system than the Rigi system. We created the prototype system using our knowledge of interoperable systems and existing technologies which have been shown to improve interoperability in the field. The prototype system was evaluated through a series of comparisons with the Rigi system. Each of the comparisons consisted of a look into one of our characterizations of interoperability. This demonstrated where our prototypes had successes, but also where our prototype did little or nothing to improve the type of interoperability being measured.

Our findings of comparing our prototype system and the Rigi system are that for each type of interoperability our prototype is at least as good as Rigi. For all but one type of interoperability, our prototype showed significant advantages compared to the Rigi system.

8.2 Contributions

This thesis outlines four important contributions

- characterization of interoperability issues

  data integration

  control integration

  process integration

- design and implementation of the ACRE Engine prototype

  ACRE Engine prototype interface to SVG

  ACRE Engine prototype interface to SOAP

These contributions also include a comparison between the prototype system, and an existing established system. This comparison allowed us to demonstrate which decisions can affect the interoperability of a system.

The characterization of interoperability issues is a result of a collection of information and though synthesized for the reader. The value in this contribution comes from the resulting requirements and design methods which can be employed to improve interoperability in a system. This characterization also allowed for the comparison of our prototype system with the Rigi system. This provided concrete results which showed the potential avenues for other systems looking to improve their interoperability.

The documentation of the design and implementation of the ACRE Engine prototype allows future readers to both employ and evaluate the choices made during the development

of the prototype. Readers will hopefully be able to understand why selected solutions were selected, as well as understand why other options were not selected. This is of particular interest for an individual in the future who may disagree with the inclusion of a particular portion of the decision, and will as a result be able to correctly deduct the new intended result. For example, should a reader disagree that each distinct format being used to interact with other systems would require a parser, they may not agree that a single data transfer format would be required. This readers interoperability solution may then include a repository which "speaks" multiple languages.

The ACRE Engine prototype also demonstrated how a repository can interface with both SOAP and SVG. Both of these are interesting for distinct reasons. The SOAP interface demonstrates how a repository interface can communicate data, control and process integration information between multiple distinct user interfaces. The SVG interface is of particular interest because it shows how two existing protocols, which at first glance cannot interoperate, can work together to create a viable solution. In this case the SVG HTTP interface was extended to support SVG data, and SOAP was extended to accept requests using an alternate protocol.

8.3 Future Work

Although significant effort has gone into this thesis and the prototype, there is always room for improvement. We could be looking into future work on the following fronts:

- Replace the file system repository with a suitably optimized version;
- Separate graph data (Model + Presentation);
- Increase the server side functionality;
- Link code parsers directly to the ACRE Engine;
- Conduct a user study to measure how interoperability affects tool adoption;

- Implement a lightweight implementation of the repository interface

Currently the prototype uses a file based repository (cf. Section 4.2). This prototype repository interface could produce both increased speed and increased scalability if the file based repository were replaced with a commercial system. At this time, the model and presentation data could be separated. This separation would allow multiple views of the data to be stored with less data duplication.

Although the ACRE Engine provides a complete scripting interface, there will always be complex routines which have not yet been defined. The ACRE Engine would benefit from an increased number or well documented base scripts. In some cases (e.g. performing layout operations) the ACRE Engine would also benefit from providing additional functionality within the server to increase performance and scalability.

Currently none of the reverse engineering parsers directly connect to the ACRE Engine. The prototype would benefit from creating some connections to the parsers other than through the current client user interfaces.

Presently a user study has not yet been conducted. One of the following steps in the ACRE project includes conducting a user to study to test whether the increased interoperability found in the prototype presented in this thesis has affected tool adoption for any portion of the components of the ACRE System. Such a study could also provide insight into which types of interoperability, if any, have a larger or lesser effect on tool adoption.

Finally, a light-weight implementation of the ACRE Engine interface could be created. This would give the ACRE project an opportunity to test whether a completely lightweight application, a mix of lightweight and heavyweight applications or a heavy-weight application would best improve tool adoption.

Bibliography

[1] I. Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1996.

[2] H. Müller. Adoption-centric software engineering. http://www.acse.cs.uvic.ca/, University of Victoria, Department of Computer Science, 2003.

[3] Sun Microsystems, editor. *Java Servlet Technology*. http://java.sun.com/products/servlet/, October 2003.

[4] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[5] G. Kurpis and C. Booth, editors. *The New IEEE Standard Dictionary of Electrical and Electronic Terms*, pages 222, 226–7, 303, 306, 460, 676, 1022. Institute of Electrical and Electronics Engineers, Inc., 5th edition, 1993.

[6] B. Lientz and E. Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[7] J. McKee. Maintenance as a function of design. In Dennis Frailey, editor, *AFIPS Conference Proceedings*, pages 187–193, 1984.

[8] M. Gertz. Information Systems Interoperability. http://sirius.cs.ucdavis.edu/teaching/265-SQ03/Handouts/interop.pdf, 2003.

[9] J. Hamilton. Enabling Interoperability Via Software Architecture. http://www.drew-hamilton.com/ccrts2000/Interop_Via_Software_Architecture.pdf, 2000.

[10] D. Connolly. Tcl and Java integration. http://www.w3.org/MarkUp/SGML/, August 2000.

[11] C. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall, 2nd edition, 2000.

[12] R. Holt, A. Schürr, and S. Sim. Graph eXchange Language. http://www.gupro.de/GXL/, July 2002.

[13] A. Skonnard and M. Gudgin. *Essential XML: Quick Reference*. Addison-Wesley, 2002.

[14] D. Stone and J. Nestor. IDL: Background and Status. *SIGPLAN Notices*, 22(11):5–9, 1987.

[15] Y. Lafon. Hypertext Transfer Protocol. http://www.w3.org/Protocols/, October 2003.

[16] G. Huang and K. Mak. Web-based product design review: implementation perspective. In *6th IEEE Conference on Computer Supported Cooperative Work in Design*, pages 261–266, 2001.

[17] J. Ferraiolo and D. Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. http://www.w3.org/TR/SVG/, 2003.

[18] Adobe Systems Inc. The Adobe SVG Zone. http://www.adobe.com/svg/main.html, 2004.

[19] A. Fritze. Mozilla SVG Project. http://www.mozilla.org/projects/svg/, 2003.

[20] K. Wong. *The Reverse Engineering Notebook.* PhD thesis, University of Victoria, Department of Computer Science, Victoria, B.C., 2000.

[21] H. Müller and K. Klashinsky. Rigi: A system for programming-in-the-large. In *10th IEEE International Conference on Software Engineering (ICSE)*, pages 80–86, 1988.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 4–6, 293–303. Addison-Wesley, 1995.

[23] J. Vlissides. *Pattern Hatching: Design Patterns Applied*, pages 72–73. Addison-Wesley, Reading, Mass., 1998.

[24] J. Michaud. A software customization framework. Master's thesis, University of Victoria, Department of Computer Science, 2003.

[25] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *9th IEEE Working Conference on Reverse Engineering (WCRE 2002)*, pages 97–108, October 2002. http://www.cwi.nl/projects/renovate/javaQA/.

[26] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*, page 75. Addison-Wesley, Reading, Mass., 1999.

[27] A. Weber, H. Kienle, and H. Müller. Live documents with contextual, data-driven information components. In *SIGDOC 2002*, pages 236–247, Toronto, Canada, October 2002.

[28] J. Simpson and E. Weiner, editors. *The Oxford English Dictionary*, page 596. Clarendon Press, Oxford, 2nd edition, 1989.

[29] A. Rayman. Personal communications, 2003. IBM Views on Tool Deployment.

[30] M. Storey, H. Müller, and K. Wong. Manipulating and documenting software structure. *Software Engineering and Knowledge Engineering*, 7:244–263, 1996.

[31] M. Klein, R. Kazman, and R. Nord. A BASis (or ABASs) for Reasoning About Software Architectures. Technical report, Carnegie Mellon University, 2000. http://archives.distance.cmu.edu/library/arch/ICSE00.pdf.

[32] M. Klein and R. Kazman. Designing and Analyzing Software Architectures using ABASs. In *22nd IEEE International Conference on Software Engineering (ICSE)*, page 820, 2000.

[33] M. Klein and R. Kazman. Attribute-Based Architectural Styles. Technical Report CMU/SEI-99-TR-022, Carnegie Mellon University, 1999.

[34] T. Tsandilas. Integration and Interoperability in Open Hypermedia Systems. http://sirius.cs.ucdavis.edu/teaching/265-SQ03/Handouts/interop.pdf, 2000.

[35] J. Clark. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt, 1999.

[36] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[37] G. Schneider and J. Gersting. *An Invitation to Computer Science.* Thomson Learning, 1998.

[38] P. Le Hgaret. Document Object Model (DOM). http://www.w3.org/DOM/, 2004.

[39] D. Brownell. Official website for SAX. http://www.saxproject.org/, 2002.

[40] R. Johnson. Tcl and Java integration. http://www.tcl.tk/software/java/tcljava.pdf, October 2003.

[41] I. Thomas and B. Nehmeh. Definitions of tool integration for environments. *IEEE Software*, 9(2):29–35, March 1992.

[42] S. Reiss. Simplifying data integration: the design of the desert software development environment. In *18th IEEE International Conference on Software Engineering (ICSE 1996)*, pages 398–407, 1996.

[43] A. Brown. Control integration through message-passing in a software development environment. *IEEE Software Engineering Journal*, 8(3):121–131, 1993.

[44] Y. Zou and K. Kontogiannis. Migrating and specifying services for Web integration. *Lecture Notes in Computer Science*, 1999:253–271, 2001.

[45] T. Ajisaka. Meta-integration for process integrated case environments. In *9th IEEE International Conference on Software Process Workshop*, pages 62–66, 1994.

[46] U.S. Department of Transport. ITS Interoperability. http://www.standards.its.dot.gov/Documents/Interoperability.pdf, 2002.

[47] IBM. Lotus Notes. http://www.lotus.com/, October 2003.

[48] Eclipse. http://www.eclipse.org/, October 2003.

[49] J. Cordy. The TXL Programming Language. http://www.txl.ca/, 2003.

[50] D. Jin, J. Cordy, and T. Dean. Transparent reverse engineering tool integration using a conceptual transaction adapter. In *7th IEEE Conference on Software Maintenance and Reengineering*, pages 399–408, 2003.

[51] B. Gautier, C. Loftus, E. Sherratt, and L. Thomas. Tool integration: experiences and directions. In *Proceedings of the 17th international conference on Software engineering*, pages 315–324. ACM Press, 1995.

[52] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments. part i: tool specification. *ACM Trans. Softw. Eng. Methodol.*, 1(2):135–167, 1992.

[53] XML Schema. http://www.w3.org/TR/xmlschema-0/, October 2003.

Appendix A

XML Sample Document and Schema

As discussed in Chapter 4, the ACRE Engine uses technologies derived from XML, including SOAP and GXL. Below is a generic example of an XML document and schema. These following examples were taken from the W3C website [53]. Although this sample XML document and schema is substantially smaller than a GXL document and the GXL schema, this example shows a simplistic example which is easily understood.

```
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="1999-10-20">
   <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
   </shipTo>
   <billTo country="US">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
      <zip>95819</zip>
   </billTo>
   <comment>Hurry, my lawn is going wild!</comment>
   <items>
      <item partNum="872-AA">
```

```
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:annotation>
        <xsd:documentation xml:lang="en">
            Purchase order schema for Example.com.
            Copyright 2000 Example.com.  All rights reserved.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>
    <xsd:element name="comment" type="xsd:string"/>
    <xsd:complexType name="PurchaseOrderType">
        <xsd:sequence>
            <xsd:element name="shipTo" type="USAddress"/>
            <xsd:element name="billTo" type="USAddress"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="items" type="Items"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
<xsd:complexType name="USAddress">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
</xsd:complexType>
<xsd:complexType name="Items">
    <xsd:sequence>
        <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="productName" type="xsd:string"/>
                    <xsd:element name="quantity">
                        <xsd:simpleType>
                            <xsd:restriction base="xsd:positiveInteger">
                                <xsd:maxExclusive value="100"/>
                            </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="USPrice" type="xsd:decimal"/>
                    <xsd:element ref="comment" minOccurs="0"/>
                    <xsd:element name="shipDate" type="xsd:date" minOccurs="0
                </xsd:sequence>
                <xsd:attribute name="partNum" type="SKU" use="required"/>
```

```
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="3-[A-Z]2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Appendix B

WSDL

Web Services are like other programming modules as they need to have their interfaces and actions defined by a developer. The following XML fragment is the interface definition for the ACRE Engine (cf. Chapter 4). The ACRE Engine interface definition is written using the Web Service Description Language (WSDL), and is used by both the ACRE Engine and the clients, such as the Lotus Notes, Visio and Excel clients.

This interface is partially auto generated using IBM WebSphere, with some manual adjustments to provide multi-platform support. The resulting interface includes the names of the functionality provided along with the optional and required data. The expected return type is also included. The last portion of the description includes server directives for processing and routing requests to the appropriate interpreter.

The following section defines the document schemas, and informs the reader where to locate a copy of the schema. This portion is important for completing the validation of the format of an XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="rigiSoapService"
    targetNamespace="http://localhost:8080/testing/wsdl/rigiSoap.wsdl"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://localhost:8080/testing/wsdl/rigiSoap.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
```

Each of the message definitions below represents either an incoming or outgoing message. The incomming message names end with “Request” while outgoing message names end

with “Response”. Each message type also specifies the data components of the particular message.

```
<message name="exportGraphRequest">
    <part name="id" type="xsd:string"/>
    <part name="format" type="xsd:string"/>
</message>
<message name="exportGraphResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="executeRequest">
    <part name="tcl" type="xsd:string"/>
</message>
<message name="executeResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="listResourcesRequest">
    <part name="type" type="xsd:string"/>
</message>
<message name="listResourcesResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="setSessionPrefRequest">
    <part name="var" type="xsd:string"/>
    <part name="val" type="xsd:string"/>
</message>
<message name="setSessionPrefResponse">
    <part name="result" type="xsd:boolean"/>
</message>
<message name="importScriptRequest">
    <part name="id" type="xsd:string"/>
```

```
    <part name="script" type="xsd:string"/>
</message>
<message name="importScriptResponse">
    <part name="result" type="xsd:boolean"/>
</message>
<message name="exportBlobRequest">
    <part name="id" type="xsd:string"/>
</message>
<message name="exportBlobResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="importGraphRequest">
    <part name="format" type="xsd:string"/>
    <part name="graph" type="xsd:string"/>
</message>
<message name="importGraphResponse">
    <part name="result" type="xsd:boolean"/>
</message>
<message name="exportScriptRequest">
    <part name="id" type="xsd:string"/>
</message>
<message name="exportScriptResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="getSessionPrefRequest">
    <part name="var" type="xsd:string"/>
</message>
<message name="getSessionPrefResponse">
    <part name="result" type="xsd:string"/>
</message>
<message name="importBlobRequest">
```

```
    <part name="id" type="xsd:string"/>
    <part name="blob" type="xsd:string"/>
</message>
<message name="importBlobResponse">
    <part name="result" type="xsd:boolean"/>
</message>
<portType name="rigiSoapJavaPortType">
```

The following operation definitions create the link between the incoming message, the outgoing message and the functionality to execute. Notice that these definitions closely resemble a Java method call.

```
<operation name="exportGraph">
    <input name="exportGraphRequest"
    message="tns:exportGraphRequest"/>
    <output name="exportGraphResponse"
    message="tns:exportGraphResponse"/>
</operation>
<operation name="execute">
    <input name="executeRequest"
    message="tns:executeRequest"/>
    <output name="executeResponse"
    message="tns:executeResponse"/>
</operation>
<operation name="listResources">
    <input name="listResourcesRequest"
    message="tns:listResourcesRequest"/>
    <output name="listResourcesResponse"
    message="tns:listResourcesResponse"/>
</operation>
```

```
<operation name="setSessionPref">
   <input name="setSessionPrefRequest"
   message="tns:setSessionPrefRequest"/>
   <output name="setSessionPrefResponse"
   message="tns:setSessionPrefResponse"/>
</operation>
<operation name="importScript">
   <input name="importScriptRequest"
   message="tns:importScriptRequest"/>
   <output name="importScriptResponse"
   message="tns:importScriptResponse"/>
</operation>
<operation name="exportBlob">
   <input name="exportBlobRequest" message="tns:exportBlobRequest"/>
   <output name="exportBlobResponse"
   message="tns:exportBlobResponse"/>
</operation>
<operation name="importGraph">
   <input name="importGraphRequest"
   message="tns:importGraphRequest"/>
   <output name="importGraphResponse"
   message="tns:importGraphResponse"/>
</operation>
<operation name="exportScript">
   <input name="exportScriptRequest"
   message="tns:exportScriptRequest"/>
   <output name="exportScriptResponse"
   message="tns:exportScriptResponse"/>
</operation>
<operation name="getSessionPref">
   <input name="getSessionPrefRequest"
```

```
            message="tns:getSessionPrefRequest"/>
            <output name="getSessionPrefResponse"
            message="tns:getSessionPrefResponse"/>
        </operation>
        <operation name="importBlob">
            <input name="importBlobRequest" message="tns:importBlobRequest"/>
            <output name="importBlobResponse"
            message="tns:importBlobResponse"/>
        </operation>
    </portType>
    <binding name="rigiSoapBinding" type="tns:rigiSoapJavaPortType">
        <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
```

The following operation definitions map the SOAP requests to the web services provided. This allows the SOAP service to specify any additional information which may be required to process a valid SOAP request.

```
        <operation name="exportGraph">
            <soap:operation soapAction="" style="rpc"/>
            <input name="exportGraphRequest">
                <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/rigiSoap"/>
            </input>
            <output name="exportGraphResponse">
                <soap:body use="encoded"
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="http://tempuri.org/rigiSoap"/>
            </output>
```

```xml
</operation>
<operation name="execute">
    <soap:operation soapAction="" style="rpc"/>
    <input name="executeRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="executeResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="listResources">
    <soap:operation soapAction="" style="rpc"/>
    <input name="listResourcesRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="listResourcesResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="setSessionPref">
    <soap:operation soapAction="" style="rpc"/>
    <input name="setSessionPrefRequest">
        <soap:body use="encoded"
```

```
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="setSessionPrefResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="importScript">
    <soap:operation soapAction="" style="rpc"/>
    <input name="importScriptRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="importScriptResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="exportBlob">
    <soap:operation soapAction="" style="rpc"/>
    <input name="exportBlobRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="exportBlobResponse">
        <soap:body use="encoded"
```

```
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="importGraph">
    <soap:operation soapAction="" style="rpc"/>
    <input name="importGraphRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="importGraphResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="exportScript">
    <soap:operation soapAction="" style="rpc"/>
    <input name="exportScriptRequest">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </input>
    <output name="exportScriptResponse">
        <soap:body use="encoded"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://tempuri.org/rigiSoap"/>
    </output>
</operation>
<operation name="getSessionPref">
```

```
        <soap:operation soapAction="" style="rpc"/>
        <input name="getSessionPrefRequest">
            <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://tempuri.org/rigiSoap"/>
        </input>
        <output name="getSessionPrefResponse">
            <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://tempuri.org/rigiSoap"/>
        </output>
    </operation>
    <operation name="importBlob">
        <soap:operation soapAction="" style="rpc"/>
        <input name="importBlobRequest">
            <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://tempuri.org/rigiSoap"/>
        </input>
        <output name="importBlobResponse">
            <soap:body use="encoded"
            encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
            namespace="http://tempuri.org/rigiSoap"/>
        </output>
    </operation>
</binding>
<service name="rigiSoapService">
    <port name="rigiSoapPort" binding="binding:rigiSoapBinding">
        <soap:address
        location="http://localhost:8080/testing/servlet/rpcrouter"/>
    </port>
```

```
    </service>
</definitions>
```

Appendix C
Microsoft SOAP Connection

The ACRE Engine (cf. Chapter 4) provides a SOAP (Section 2.4.3) service as defined in Appendix B. To communicate with the service from a client, we need to include some communication code. The code below is used to communicate between the ACRE Engine and MS Office products. The code fragment is written in VBScript and requires that the Microsoft SOAP Toolkit is installed on the client machine. This toolkit can be downloaded free of charge from Microsoft at http://msdn.microsoft.com/webservices/building/soaptk/

.

```
Sub Testing()
    ' Testing Macro
    ' Macro created 6/20/2002
    Dim SOAPClient
    Set SOAPClient = CreateObject("MSSOAP.SOAPClient30")
    On Error Resume Next

    SOAPClient.mssoapinit("http://localhost:8080/
        testing/wsdl/rigiSoap.wsdl")
    If Err Then
        MsgBox "WSDL1= " + SOAPClient.faultString
        MsgBox "WSDL2= " + SOAPClient.detail
    End If
    MsgBox "mssoapinit done"



    outputText = SOAPClient.listResources(graph)
```

```
    If Err Then
        MsgBox "SOAP1= " + SOAPClient.faultString
        MsgBox "SOAP2= " + SOAPClient.detail
    Else
        MsgBox outputText
    End If
End Sub
```

Appendix D
Tcl Extensions

When the ACRE Engine was developed, the TCL scripting language [36] was used to create server extensions (cf. Section 4.7). To complete this task, we needed to provide some extra functionality to allow the TCL interpreter access to some of the internal functions of the ACRE Engine. Table D below provides a list of the names of the TCL extensions created including their parameters and return types.

Table D.1: Tcl API extension for the ACRE Engine

Command Name	Return Type	Parameter Type(s)
AddEdge	”void”	”edge id” ”edge type” ”edge start id” ”edge end id”
AddEdgeAttr	”void”	”node id” ”attr name” ”attr contents”
AddGraphAttr	”void”	”node id” ”attr name” ”attr contents”
AddNode	”void”	”node id” ”node type”
AddNodeAttr	”void”	”node id” ”attr name” ”attr contents”
CallScript	”result”	”script id” [params]
ExportGraph	”gxl graph”	[graph id] ”format”
ExportScript		”script id”
GetSessionPref	”string”	”var name”
ImportGraph		[graph id] ”attachment id”
ImportScript		”script id” ”attachment id”
LoadGraph	”void”	”graph id”
NewGraph	”void”	”graph id”
RemoveEdge	”void”	”edge id”
RemoveEdgeAttr	”void”	”edge id” ”attr name”
RemoveGraphAttr	”void”	”attr name”
RemoveNode	”void”	”node id”
RemoveNodeAttr	”void”	”attr name”
SetSessionPref	”void”	”var name” ”var value”
StoreGraph	”void”	”graph id”
UpdateEdge	”void”	”edge id” ”edge type” ”edge start id” ”edge end id”
UpdateNode	”void”	”node id” ”node type”

[...] Denotes the parameter is optional
” ... ” Denotes the parameter is required

Appendix E
SVG Communication Scripts

The ACRE Engine (cf. Chapter 4) provides a SOAP (Section 2.4.3) service as defined in Appendix B. To communicate with the service from a client, we need to include some communication code. The code below allows communication between the ACRE Engine and the SVG Client (cf. Figure 5.1). The code fragment is written in Java Script and is executed within the SVG Client using an SVG interpreter. We tested this code using Adobe's SVG 3.0 plug-in installed in Internet Explorer.

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\      PostGraphCommand
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

inherit(Command, PostGraphCommand);

function PostGraphCommand (graph, id)
{
    if (arguments.length > 0) {
        this.init(graph, id);
    }
}
```

```
PostGraphCommand.prototype.init = function (graph, id)
{
    this.graph = graph;
    this.id = id;
};
PostGraphCommand.prototype.execute = function ()
{
    var form = new Object();
    form["function"] = "importGraph";
    form["graph"] = (new ViewGXLEncoder()).toGXL(graph.view, this.id);
    form["format"] = "GXL";

    server.sendRequest(form, this.sent);
};

PostGraphCommand.prototype.sent = function (data)
{
    \\ expect "true" or "false"
    if (data.success && data.content != "true") {
        var msg = "";
        msg += "send request has failed :  ";
        msg += data.content; \\ expect "true" or "false"
        alert("RESULT\n" + msg);
    }
};
```

```
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
\\      FetchGraphCommand
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

inherit(Command, FetchGraphCommand);

    \**
    Fetch the graph from the server
    @param graph the graph component to display the fetched graph
    \
function FetchGraphCommand (graph, id)
{
    if (arguments.length > 0) {
        this.init(graph, id);
    }
}

FetchGraphCommand.prototype.init = function (graph, id)
{
    this.graph = graph;
    this.id = id;
};

FetchGraphCommand.prototype.execute = function ()
{
    var form = new Object();
    form["function"] = "exportGraph";
    form["id"] = this.id;
    form["format"] = "GXL";
```

```
server.sendRequest( form, new this.constructor(this.graph) );
    var url = "\testing\servlet\rigiPost";
    var callback = this; \\ probably not called with this.
    var type = "application\x-www-form-urlencoded";
    var enc = null; \\ gzip or deflate
    \\TODO why can't we just pass 'this'???
    postURL( url, text, new this.constructor(this.graph), type, enc );
};
FetchGraphCommand.prototype.operationComplete = function (data)
{
    if (data.success) {
        var gxldocfrag = parseXML(data.content);
        var view = new GXLImportGraphView(this.graph.model, gxldocfrag);
        this.graph.setView(view);
    }
};
```

Appendix F
Memory Model Graph Class

As discussed in Chapter 4, the ACRE Engine contains a memory model. Although much of the code base has been excluded from this thesis, we have chosen to include a sample. The following code represents a graph, containing references to the nodes, arcs and attributes associated with this particular graph.

```
public class Graph implements Serializable{

    /*
     * The edges have three pointers to each edge which need to be
     * maintained.  The Graph global one is for efficient searching
     * and edge retrieval, the other two reside within the start
     * and end nodes of the arc.  These lists allow each node to
     * keep track of edges for possible traversals.
     */

    private Hashtable nodes = null;
    private Hashtable edges = null;
    private Hashtable attrs = null;
    private String id = null;

    /**
     * This constructor should never be used.
     */
    private Graph(){}

    /**
     * This constructor creates an empty graph for the string specified.
```

```
 * @param String id The graph identifier
 * @throws InvalidId When the graph Id is not valid.  (ie.  null or '''')
 */
public Graph(String id) throws InvalidId{
    this.id = id;
    if(id == null ||id == '''')
        throw new InvalidId(''Id was not specified'');
    nodes = new Hashtable();
    edges = new Hashtable();
    attrs = new Hashtable();
}

/**
 * This constructor creates an empty graph for the string specified.
 * @param String id The graph identifier
 * @param Hashtable attrs The list of graph attributes.
 * @throws InvalidId When the graph Id is not valid.  (ie.  null or "")
 */
public Graph(String id, Hashtable attrs) throws InvalidId{
    this.id = id;
    if(id == null ||id == '''')
        throw new InvalidId(''Id was not specified'');
    nodes = new Hashtable();
    edges = new Hashtable();
    this.attrs = attrs;
}

/**
 * Adds a node to the graph.
 * @param Node n The node to add to the graph.
 */
```

```
public boolean addNode(Node n){
    if(n == null)
        return false;
    Node nn = (Node)nodes.put(n.getId(),n);
    if(nn!=null){
        nodes.put(nn.getId(),nn);
        return false;
    }
    return true;
}

/**
 * Adds an edge to the graph.
 * @param Edge e The edge to add to the graph.
 */
public boolean addEdge(Edge e){
    if(e==null)
        return false;
    Edge ee = (Edge)edges.put(e.getId(),e);
    if(ee!=null){
        edges.put(ee.getId(),ee);
        return false;
    }
    return (e.getStartNode().addEdge(e) && e.getEndNode().addEdge(e));
}
```

The remainder of this class follows the pattern set by the two add methods is other access methods to add attributes or to remove attribute, nodes or edges. Also included are methods to retrieve portions of the graph.