Domain-Retargetable Reverse Engineering[†]

Scott R. Tilley Hausi A. Müller Michael J. Whitney Kenny Wong

Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
Tel: (604) 721-7294, Fax: (604) 721-7292
E-mail: {`stilley, hausi, mwhitney, kenw`}`@csr.uvic.ca`

Abstract

Any response to the software maintenance challenge must address the underlying problem of program understanding. One way of doing this is through reverse engineering. A successful approach to reverse engineering must be both flexible *and* scalable. *Most reverse engineering tools provide a fixed palette of analysis, extraction, organization, representation, and selection techniques. This paper describes a* user-programmable *approach to reverse engineering. The approach uses a scripting language that enables users to write their own routines for these activities, making the system* domain-retargetable. *The environment supported by this programmable approach subsumes existing reverse engineering systems by being able to simulate facets of each one, and provides a smooth transition from semi-automatic to automatic reverse engineering.*

Keywords: domain-retargetable, program understanding, reverse engineering, software maintenance

1 Introduction

The primary business of software engineering has historically been new development; now it is maintenance [1]. This fundamental shift was inevitable: the software profession has reached a turning point, one where more people are employed to maintain existing applications than to develop new systems from scratch. Traditional approaches to the software process have placed too much emphasis on the artificial distinction between development and maintenance. A better term is *software evolution*, which refers to the on-going enhancements of existing software systems, involving both development and maintenance. Since evolution begins early in the development phase, the distinction between development and maintenance should be abandoned in favor of an evolutionary process [2, 3]. In fact, learning to master evolving systems was recently identified as paramount for future software engineering research [4].

Central to the software evolution challenge is program understanding. For large legacy systems,[1] this is a very difficult task. Unless one has been with a project since its inception, there is little chance of understanding a multi-million line program without considerable effort—if at all. Perhaps if the overall structure and maintenance history of the subject system had been better documented, program understanding would be easier. Unfortunately, systems that fall into this category are the exception rather than the rule. Without such information, a maintainer must rely on disparate and informal sources of information as an aid in program understanding. Ultimately, it is the source code that is the sole objective arbiter [5]. Maintenance personnel are forced to spend an inordinate amount of time attempting to create an abstract representation of the system's high-level architecture by exploring its low-level source code—not an easy task for large systems. One way of augmenting this understanding process is through *reverse engineering.*[2]

Our approach to reverse engineering involves extracting system abstractions and design information from existing software systems. This information can

[†]This work was supported in part by the British Columbia Advanced Systems Institute, IBM Canada Ltd., the IRIS Federal Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, the Science Council of British Columbia, and the University of Victoria.

[1]Legacy software systems are those that are 10-25 years old and often in poor condition because of prolonged maintenance.

[2]Specifically, *authorized* reverse engineering of one's own code. We do not advocate improper reverse engineering of source code when one does not have the right to do so.

then be used to improve subsequent development, facilitate maintenance and re-engineering, and aid project management. The process involves the identification of software artifacts in the subject system and the organization of these artifacts into more abstract system representations, thereby reducing complexity. Through reverse engineering, the overall structure of the subject system can be determined and some of its architectural design information recovered. Software structure refers to a collection of artifacts that software engineers use to form mental models when designing, documenting, implementing, integrating, inspecting, or analyzing software systems. Artifacts include software *components* such as procedures, modules, subsystems, and interfaces; *dependencies* among components such as supplier-client, composition, and control-flow relations; and *attributes* such as component type, interface size, and interconnection strength. The structure of the system is the organization and interaction of these artifacts [6].

This paper describes an approach to supporting software evolution through reverse engineering. In particular, the paper focuses on extensions to an existing reverse engineering environment to make it *flexible* and *user-programmable*, and hence *domain-retargetable*. By providing a user-programmable reverse engineering environment, the application domain need not be limited to one area. Most reverse engineering tools provide a fixed palette of extraction, selection, organization, and representation techniques. Our approach uses a scripting language that enables users to write their own routines for these activities. The underlying system is flexible enough to support the language-dependent extraction of artifacts from several programming languages, the language-independent organization of these artifacts into stratified subsystems through user-defined clusterings, and the presentation and documentation of the resultant structures in a user-selectable manner.

The sequel outlines three desirable properties of a reverse engineering system: flexibility, scalability, and domain independence. Section 3 describes the architecture of an existing reverse engineering environment and comments on how well it addresses these three goals. Section 4 presents the extensions to this system required to make it a programmable reverse engineering environment. Section 5 summarizes the paper and briefly discusses future work.

2 Domain retargetability

In our approach to reverse engineering, *flexibility* and *scalability* are key requirements. The approach must be flexible so that the results can be applied to many diverse program understanding scenarios as well as different target domains. "Domains" in this sense is an overloaded term. It refers to different *application* domains, such as banking or database systems; *implementation* domains, including the application's implementation language; and the *reverse engineering* domain, in which the software engineer models and represents the subject system. Our current objective with respect to scale is 2-3 million lines-of-code, which includes many evolving legacy systems.

One way of maximizing the usefulness of a reverse engineering system is to make it domain-specific. By doing so, one can provide users with a system tailored to a certain task and exploit any features that make performing this task easier. However, by taking this approach the system's usefulness becomes restricted to a particular domain. Using the same system on a different task, even one that is similar, may be impossible.

An alternative to making the system powerful by making it domain-specific, is to make it user-programmable and hence domain-retargetable. One would like to make the approach as flexible as possible—a subtle distinction from general. Software can be considered general if it can be used without change; it is flexible if it can be easily adapted to be used in a variety of situations [7]. General solutions often suffer from poor performance or lack of features that limit their usefulness. Flexible solutions may be tailored by the user to fully exploit aspects of the problem that make its solution easier.

2.1 Flexibility

Flexibility applies to many aspects of reverse engineering. In particular, we mean the *extensibility*, *tailorability*, and *configurability* of both the reverse engineering system's methodologies and its supporting environment. One achieves domain-retargetability through the flexibility of the system as it adapts to new domains. In fact, scalability could be considered to be an aspect of flexibility, but it is such an important aspect that we treat it separately.

2.1.1 Extensible functionality: Perhaps the most important goal for a successful reverse engineering environment is to provide a mechanism through which users can extend the system's functionality. For example, it should be possible to augment the search and selection operations of the graph editor with user-defined algorithms or to interface with external tools. Since change requests are often couched in terms of the user's view of the application, much of the effort involved in software maintenance is in locating the relevant code fragments that implement the concepts in the application domain. One should be able to use tools that support advanced clustering techniques, such as the teleologic-based IRENE [8] and DESIRE [9, 10], and have the results of their search made available to the user and the environment.

2.1.2 Incremental reverse engineering:

Most reverse engineering systems parse source code and extract complete abstract syntax trees with a large number of fine-grained syntactic objects and dependencies. This strategy works well for relatively small subject systems and programming-in-the-small tasks. However, for large legacy systems in the million-lines-of-code range, the resulting databases can be huge and unmanageable. One approach is to populate the database with coarse-grained objects only. Another strategy is to allow the user to specify pertinent subsets of the source code and/or the database. For example, one may only be interested in the call structure of a selected set of subsystems, not all dependencies of the entire program. Hence, incremental reverse engineering is a desirable feature for programming-in-the-large.

2.1.3 Semi-automatic and automatic subsystem composition:

Constructing abstract representations of the source code should be possible both automatically and semi-automatically. The user should be able to experiment with various decompositions. For example, a long-term goal of reverse engineering might be actual physical re-modularization of a system so as to minimize inter-module coupling and maximize intra-module cohesion. The system should be able to compute such modularizations automatically, and stop when a user-defined termination condition is met.

2.1.4 Tailorable user-interface:

It is desirable to allow users as much freedom as possible in configuring the system to their liking. This configurability includes modification of the system's interface components such as buttons, dialogs, menus, scrollbars, and so on. Experienced users should be able to create time-saving meta-commands or "accelerator" key sequences. More importantly, to achieve domain-retargetability it should be possible to alter the system's functionality by changing the commands associated with elements of the user interface.

2.1.5 User-defined extraction and filtering:

The users should be able to indicate what they want extracted from the source code at *extraction time*.[3] Moreover, they should be able to highlight important objects and dependencies, and de-emphasize immaterial ones.

2.1.6 User-defined multiple views:

Most existing systems provide the user with a fixed set of view mechanisms, such as call graphs and module charts. While this set might be considered large by the system's producers, there will always be users who want something else. One cannot predict which aspects of a system are important for all users, and how these aspects should be documented, represented, and presented to the user. This is an example of the trade-off between open and closed systems. An open system provides a few primitive operations and mechanisms for user-defined extensions. A closed system provides a "large" set of built-in facilities, but no way of extending the set.

2.2 Scalability

It is essential that any reverse engineering approach be applicable to large software systems. By large, we mean on the order of several million lines of code. Such a scale often precludes the use of many programming-in-the-small approaches to program understanding. There is a significant difference between programs of 1,000 lines, of 100,000 lines, and of 1,000,000 lines. The latter requires a significantly different approach to program understanding. The repository must be able to handle very large databases efficiently, the search strategies used must be effective, and the user interface must support the manipulation of very large graphs.

For very large systems, the information generated during reverse engineering is prodigious. Presenting the user with reams of data is insufficient. Knowledge is gained only through the understanding of this data. In a sense, a key to program understanding is deciding what information is material and what is immaterial: knowing what to look for—and what to ignore [11].

2.3 Domain independence

Many current systems support only relatively small programs. Others support just one programming language, usually because their parsing system, database, and support environment are tightly coupled. Some

[3] The term "extraction time" is similar to "compile time," except it does not imply generating code *per se*. It refers to the first phase of reverse engineering: populating a database with the information extracted from the source code.

environments support only a subset of the subject system's implementation language. This approach limits the application domain to small, "pure" programs rarely found in practice. One must take a pragmatic point of view; if the methodology does not work on real-world software systems, with all their "features," then it will not make an impact on existing systems.

To achieve high functionality, many systems are targeted toward a single application domain, such as COBOL banking programs. While such systems are useful in their particular area, they are not widely applicable in others. It would be better to provide users with a system that is flexible enough to be easily retargeted to new application domains, yet still maintain its full functionality. The next section describes our reverse engineering environment as it exists today, and comments on how well it meets the goals of flexibility, scalability, and domain independence.

3 The Rigi system

Rigi[4] [12] is a versatile system and framework under development at the University of Victoria for discovering and analyzing the structure of large software systems. It supports the following desirable features of a reverse engineering environment:

- A variety of parsing systems to support the common programming languages of legacy systems;[5]
- A repository to store the information extracted from the source code;
- An interactive graph editor that permits graphical manipulation of source code representations.

This section describes the overall architecture of the Rigi system, and comments on some of our early case-study experiences. These case studies helped guide the development of Rigi, and are the motivation behind the subject of this paper: making the reverse engineering environment programmable. We briefly outline the Rigi approach to reverse engineering below. A more detailed description of our methodology can be found in [13].

3.1 Reverse engineering

The process of reverse engineering a subject system involves the identification of the system's current components and their dependencies, and the extraction of system abstractions and design information. During this process, the subject system is not altered, although additional information about it is generated.

In our approach, the first phase of the reverse engineering process—the extraction of software artifacts—is automatic and language-dependent. It essentially involves parsing of the subject system and storing the artifacts in a repository. Some of our early work resulted in a graph model for software structures and a graph editor supporting the model [14]. The artifacts are stored in an underlying database specifically designed to represent graph structures. The Rigi graph editor allows the users to edit, maintain, and explore the objects stored in the repository.

Our approach to the second phase is semi-automatic and features language-independent subsystem composition methods that generate hierarchies of subsystems [15]. Subsystem composition is the process of constructing composite software components out of building blocks such as variables, procedures, modules, and subsystems. Hierarchical subsystem structures are formed by imposing equivalence relations on the resource-flow graphs of the source code. These relations embody software engineering principles concerning module interactions such as *low coupling* and *strong cohesion* [16]. We have also formulated software quality criteria and measures, based on exact interfaces and established software engineering principles, to evaluate the resultant subsystem structures [17, 18]. These measures are not meant to be absolute; they are used to judge the relative merits of one system decomposition with respect to another.

3.2 Rigi's architecture

Rigi is composed of three major subsystems: `rigireverse`, `rigiedit`, and `rigiserver`, each of which is discussed in turn below. The high-level architecture of Rigi is depicted in Figure 1. The philosophy behind Rigi is a set of cooperating and communicating tools. The parsing system extracts information from the source code and populates the database. The user manipulates the database using the graph editor. The entire system is distributed and multi-user, runs on a variety of hardware platforms (Sun 3, Sun 4, and IBM RS/6000[6]), and provides user interfaces under three different windowing systems (Motif,[7] Open Look,[8] and

[4]Rigi is named after a mountain in central Switzerland.

[5]Such systems are typically written in procedural, imperative programming languages such as C, COBOL, FORTRAN, and PL/I.

[6]IBM and Risc System/6000 are trademarks of the International Business Machines Corporation.

[7]Motif (OSF/Motif) is a trademark of the Open Software Foundation.

[8]Open Look (OPEN LOOK) is a trademark of AT&T and Sun Microsystems.

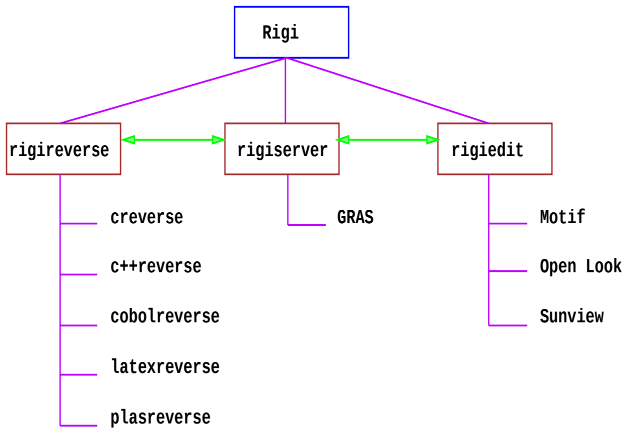


Figure 1: The Rigi environment's main components

SunView[9]).

3.2.1 `rigireverse`: `rigireverse` is the parsing system, which currently supports C, C++, COBOL, LaTeX, and PL/AS.[10] Based on a user-selectable option, `rigireverse` invokes a parsing subsystem specific to the application programming language. The parser also extracts file system information, such as directory structure and `#include` file dependencies, if appropriate. Users can choose to create an initial subsystem decomposition based on the current physical structure of the source code, or to ignore this information and generate a flat resource-flow graph.

In the long term, we would like to replace our parsers with an interface to a central repository. The repository would hold information extracted by an industrial-strength compilation system, thereby freeing us from the laborious task of writing a new parser every time we want to support a new programming language. From a reverse engineering point of view, parsing is essential but laborious. For example, we do not have the resources necessary to parse all variants of COBOL. It is better to leverage mature compiler technology and make use of the generated information.[11]

3.2.2 `rigiedit`: `rigiedit` constitutes the user interface of the Rigi system and a graph editor with which users manipulate graphical representations of the subject system during reverse engineering. The operations provided by the editor are rich because of parameterization, but the total set is fixed. A large set of selection and graph manipulation operations are provided. The implicit assumption within the editor is that one is reverse engineering an application written in one of the imperative programming languages mentioned above.[12] Consequently, the operations are geared toward coupling and cohesion as the guiding measurements used when selecting components to be collapsed into a subsystem. The selection operations depend strongly on client/supplier relationships.

The structuring mechanism supported by the editor is $(k, 2)$-partite graphs—a class of layered graphs [20]. Dependencies within each layer are based on resource flow and exchange of programming-language resources. Dependencies between layers represent aggregation relations.[13]

3.2.3 `rigiserver`: `rigiserver` is the central database repository. It is initially populated by the `rigireverse` program, and subsequently manipulated by the reverse engineer through the `rigiedit` program. The database we use is GRAS, a distributed, multi-user, network model database well suited to representing graph structures [21].

3.3 Experience with Rigi

Since 1990, we have performed three major case studies using Rigi to reverse engineer third-party applications. The first was in 1990 when we were asked by a local company to reverse engineer their 57,000-line COBOL program. In 1991, we analyzed an 82,000-line C program. We started work on our current project in late 1992. This most recent case study is concerned with analyzing a large commercial database management system (SQL/DS[14]) in conjunction with the IBM Canada Ltd. Centre for Advanced Studies in Toronto. The SQL/DS system consists of over a million lines of PL/AS and has gone through numerous revisions during its lifetime.

It was through our initial work on analyzing the SQL/DS code that shortcomings of the current Rigi system were exposed. Reverse engineering of such a large system necessitated changes to all three parts of the environment, but mainly to the graph editor. The

[9] SunView is a trademark of Sun Microsystems Inc.

[10] PL/AS is a PL/I-like development language used within IBM.

[11] We are investigating using the program information base produced by the IBM xlC C++ compiler to supplant our C++ parser.

[12] Except for LaTeX documents. In this case, `rigiedit` is used in a slightly different mode [19].

[13] The extensions to Rigi to support programmable reverse engineering will not alleviate the structuring restriction of $(k, 2)$-partite graphs. However, we are planning on supporting multiple user-defined hierarchies other than composition and aggregation in the future.

[14] SQL/DS is a trademark of the International Busisness Machines Corporation.

parsing system was augmented with a PL/AS interface and successfully processed the entire source code in an incremental manner. The database was able to handle the large graph structures produced by the parser, although its performance degraded somewhat. Overall, `rigireverse` and `rigiserver` needed only minor changes. It was in the user interface, `rigiedit`, that more fundamental changes were required.

Some of the changes required were cosmetic. For example, we changed the editor so that the screen is not redrawn every time a single graph operation is carried out. Graphs with over 10,000 nodes and arcs need to be refreshed effectively to avoid degrading interactive response time. A more dramatic change was needed in one of the basic philosophies underlying our approach. We have always felt that a semi-automatic reverse engineering environment is better than a fully automatic one, because human cognitive powers are still much more powerful and flexible than algorithms built-in to the environment. Moreover, it is important for the reverse engineer to be in charge of choosing editing operations. The tools provide information, and the reverse engineer acts upon the information when composing subsystems. However, many of the operations being performed during the initial decomposition of the SQL/DS code were repetitive, and could be automated. The user would still be in charge of accepting, rejecting, or modifying subsystem decompositions automatically produced by the environment, but the decomposition itself could be made easier.

We have demonstrated Rigi at several software engineering conferences in the last few years. Many of the questions and comments from those who viewed the system concerned incorporating their "favorite" metric into the editor and connecting the system to off-the-shelf tools. For example, the measures built-in to the system quantify encapsulation and partition quality of the generated graphs. They do not measure common metrics such as cyclomatic complexity. Rather than rewriting these routines from scratch, it would be better to interface the system to off-the-shelf software that already implements the desired functions. We would also like to integrate Rigi with complementary tools and systems under development by our research partners. For instance, we want to augment one of our simple selection operations, which is essentially a `grep` facility, with a more powerful search mechanism such as SCRUPLE [22], which uses advanced syntactic pattern-matching at the programming-language level.

To summarize, the current Rigi environment is reasonably flexible, scalable, and domain independent. The graph editor operations are language-independent, which is both an advantage and a detriment. It is an advantage, since it means a single tool will work for systems written in most imperative programming languages. It is a detriment because it means domain knowledge is lost. Our experiences with the SQL/DS code indicate the methodology scales-up into the million-lines-of-code range. However, to truly satisfy the desirable goals of flexibility, scalability, and domain independence, we need a programmable reverse engineering environment; one in which the user interface is tailorable by the user, in which the operations provided by the environment are selectable and augmentable by the user, and in which parts of the reverse engineering process are automated. Such a system is described in the next section.

4 Programmable reverse engineering

To support the goals outlined in Section 2 and achieve domain-retargetability, we extended the Rigi environment described in Section 3. This section describes the required changes to Rigi to support the approach. It also discusses the benefits of a library of reverse engineering scripts, and presents examples of scripts used in the extended environment.

4.1 Extending Rigi to support the approach

The main component to be changed was `rigiedit`, the graph editor. Previously, it consisted of two tightly-coupled subsystems: the user interface and the editor itself. All editing and selection operations were inter-mingled with operations for manipulating the user interface, such as window size, menu selection, and so forth. There are two phases to the changes in `rigiedit`. The first phase was to extend the editor's functionality. The second phase is to make the user interface tailorable.

4.2 Phase I: Extending the editor's functionality

To extend the editor's functionality, we decoupled the graph editor from the graphical user interface. We then introduced a transparent intermediate layer between the graphical user interface and the graph editor. This was needed to make the environment programmable.

4.2.1 Decoupling the graph editor and user interface: Phase one of the changes to `rigiedit` involves providing the users with a programmable graph editor. Decoupling the editor from the user interface makes it possible to program editor operations independently of graphical user actions, and to tailor the user interface to personal taste. The graph editor now provides a set of primitive operations for manipulating nodes and arcs. The meanings of these operations are implicitly defined by the user and the application domain.

4.2.2 Scripting language: It is hard for any application designer to predict all the ways the application will be used. In a reverse engineering environment, one of the goals is to facilitate program understanding. Because people learn in different ways (for example, goal-directed (top-down and inductive) versus scavenging (bottom-up and deductive)), the environment should be flexible enough to support different types of learning. Providing a programmable scripting language is one way of achieving this goal.

A scripting language amplifies the power of the environment by allowing users to write scripts to extend the tool's facilities. A perfect example is the UNIX[15] shell. A scripting language is extremely useful in a windowing environment; it just works "behind the scenes". Users who accept the default will be unaware of a scripting language—but its power is available to those users who want to take advantage of it.

Instead of writing yet another command language, we chose to use Tcl [23]. It provides an extendable core language, and was specifically written as a command language for interactive windowing applications. It also provides a convenient framework for communicating between Tcl-based tools. Each application extends the Tcl core by implementing new commands that are indistinguishable from built-in commands, but are specific to the application. These new commands may be implemented as C (or C++) procedures or as Tcl scripts.

The philosophy behind Tcl is that each event of any importance to the application should be bound to a Tcl command: each keystroke, mouse motion, button press, and menu entry. When the event occurs, it is first mapped to its Tcl command, and then executed by passing the command to the Tcl interpreter, which calls the appropriate C call-back routine that implements the Tcl command. Complex sets of operations can be described with a script and then associated with a menu entry or accelerator key so that the compound command can be easily invoked. We have changed the editor so that each Rigi command, which was previously only available by menu operations, is now implemented as a Tcl command. The menu operation actually invokes the Tcl script, which in turn calls the appropriate editor operation.

[15] UNIX is a trademark of Unix Systems Laboratories, Inc.

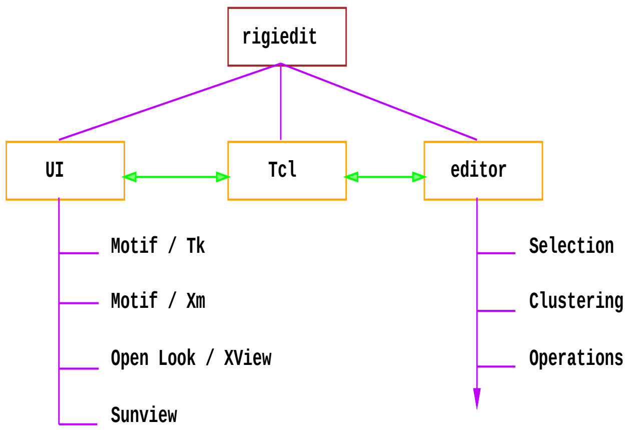


Figure 2: Extending `rigiedit`

Tcl is application-independent and provides two different interfaces: a textual interface to users who issue Tcl commands, and a procedural interface to the application in which it is embedded. In the new `rigiedit`, the Tcl interpreter sits between the graphical user interface and the graph editor, as shown in Figure 2. For those who have used the Rigi environment before, the creation of an intermediate layer and the use of Tcl is not visible; they continue to manipulate the editor using the graphical user interface alone.

4.3 Phase II: Making the user interface tailorable

Phase two of the changes to `rigiedit` involves providing the users with a configurable user interface.[16] Besides using Tcl as a scripting language to enable users to configure the reverse engineering environment's actions, one may use Tcl in conjunction with the Tk toolkit to configure the environments interface. Tk is an X11 toolkit that implements the Motif look-and-feel. It is similar in functionality to the Xm toolkit which we currently use for our Motif interface, except that it may be programmed in Tcl rather than C.

By using Tcl as an intermediary for all interface actions, users can write Tcl scripts to reconfigure the interface. For example, they can rebind keystrokes, change mouse buttons, or replace an existing operation with a more complex one specified as a set of Tcl commands. By using the Tk toolkit, one can recon-

[16] We expect to begin this second phase of the changes to `rigiedit` in the Fall of 1993.

figure the appearance of the environment. All of the system's interface components can be configured using Tcl commands. This makes it possible for users to write Tcl programs to personalize the layout and appearance of the environment as desired.

Users can write scripts that are read automatically upon `rigiedit` invocation, which tailor the interface to suit their needs. Moreover, since the scripting language is interpreted, the graphical user interface can be dynamically altered by using the appropriate Tcl/Tk commands.

4.4 Benefits of the approach

Previously, all interaction with the graph editor was interactive and human-intensive. Over time, experienced software engineers create a repertoire of commonly used reverse engineering techniques, which they must repeat every time. By separating the user interface from the structural-manipulation aspects of the editor, much of the reverse engineering can be done in batch procedures. Users can produce a library of useful reverse engineering scripts by bundling groups of often used commands together into procedures. Such scripts can assist in automating recurring tasks [24]. More important, users can create libraries of domain-dependent reverse engineering strategies. As their expertise in their application domain grows, so will their library of scripts. This approach is analogous to providing a separate I/O library in C; rather than defining a fixed set of primitives, I/O was left out of the language definition and made user-definable. The scripting language serves a similar function in our reverse engineering environment.

The scripting mechanism enables the environment to run in batch mode. In this manner, subsystem compositions produced in a previous session can be automatically recreated; all commands are logged to a script-based command file that can then be reloaded upon startup. Interactive sessions are recorded and replayed as a sequence of Tcl commands. It is also possible to write Tcl scripts that simulate the actions of the program for demonstration purposes.

The programmable approach provides a smooth transition from semi-automatic to automatic reverse engineering. It makes it possible to automatically decompose systems according to user-defined criteria, such as those based on coupling and cohesion metrics. Programmable decomposition criteria also enable the environment to simulate other reverse engineering approaches. For example, the system-restructuring algorithm based on *alteration distance* as described in [25] could be implemented as a combination of Tcl scripts and C code. Or, users could write scripts to select subsystems based on decomposition slices [26, 27, 28]. In essence, this programmable approach subsumes existing reverse engineering systems that provide a fixed set of operations, by being able to simulate facets of each one.

4.5 Example scripts

Scripts may be used to replace common editing operations. For example, the script `my_open` shown in Figure 3 is designed to replace the default open operation within the editor. Instead of simply opening a composite node to show its children without formatting or scaling, it opens the node and presents all its children in a grid format, scaled to the current window. The three steps {select, group, scale} are often used during exploratory reverse engineering.

The script `build_subsystems`, also shown in Figure 3, may be used to automatically decompose a system into subsystems using name search as the selection operation. This simple search technique has proven to be extremely useful when reverse engineering large applications which follow a naming convention. To use `build_systems`, the user invokes the procedure with a list of names to be used, as in '`build_subsystems ARIM ARIX ARIY`.'

One of the most common editing operations performed when using Rigi is a combination of selection and grouping operations. They are often used to get an initial view of the potential entry and exit points in a subsystem. The system by itself does not provide any advanced graph layout algorithms. However, by using Tcl's inter-process communication mechanism, one can use an external graph layout package to place the nodes as desired. For the purposes of this example, we will use Rigi's built-in layout operations, which place nodes either all on top of one another, or in a simple geometric pattern: horizontal rows, vertical columns, or on a grid. The script for performing this operation, called `sandwich`, is also shown in Figure 3. The result of using this script in our environment is depicted in Figure 4.

5 Summary

Controlled software evolution is attainable only with improved program understanding techniques. However, the understanding process is more dependent on individuals and their specific cognitive abilities than on the limited set of facilities that tools provide. Understanding also requires the ability to

```
proc my_open {node}
{
  open node $node
  select all
  group grid
  scale window
}
```

my_open

```
proc build_subsystems args
{
  foreach node $args {
    delect all
    select grep $node"*"
    edit collapse
    edit rename $node
  }
}
```

build_subsystems

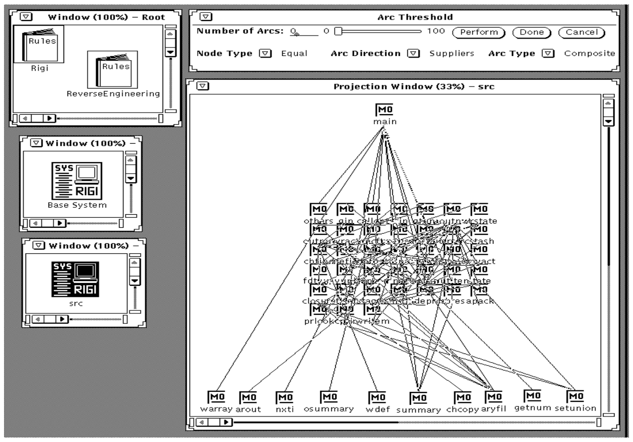
```
proc sandwich {}
{

  # Get to base-level RFG
  open node "Rigi"
  open node "Base System"
  open node "src"
  operation project subsystem
  select all
  move group grid

   # Top-layer
   deselect all
   select arc thresholds
   set arc threshold 0
   set arc direction clients
   set arc type composite
   set node type equal
   perform
   move group horizontal


   # Bottom-layer
   deselect all
   select arc thresholds
   set arc direction suppliers
   perform
   move group horizontal

}
```

sandwich

Figure 3: Sample scripts

adapt to various application domains, implementation languages, and working environments. This paper discusses an approach to program understanding through programmable reverse engineering, which gives individuals the ability to tailor their environment to suit their needs.

The approach achieves flexibility by means of a high-level separation between reverse engineering system components, and the incorporation of a scripting language that provides an interfacing mechanism. The power of the system is found in the cooperative use of small command scripts. The scripts give users the ability to extend the tools in their reverse engineering toolbox by defining, storing, and retrieving commonly-used operations. Using Tcl as the scripting language enables us to leverage skills of other people and groups who write Tcl scripts; if we had written



Number of Arcs: 0 0 100
Node Type Equal Arc Direction Suppliers Arc Type Composite

Figure 4: Result of running the **sandwich** script

our own scripting language, this would not have been possible to nearly the same extent.

A decoupled interface between system components provides an increased ability to integrate the reverse engineering environment with existing tools. This allows the environment itself to evolve—without requiring continual rewriting. Together with user-defined scripts, whole libraries of program understanding techniques may be gathered or created, and maintained.

We are currently in the process of decoupling our own Rigi reverse engineering system, so that the scripting language may be fully incorporated. Our next task will be to complete the support for user-tailorable interfaces, and integrate our system with other reverse engineering and compiler tools.

References

[1] R. L. Glass. We have lost our way. *The Journal of Systems and Software*, 18(3):111–112, March 1992.

[2] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15:225–252, 1976.

[3] D. C. Poo and P. J. Layzell. An evolutionary structural model for software maintenance. *The Journal of Systems and Software*, 18(2):113–123, May 1992.

[4] W. F. Tichy, N. Habermann, and L. Prechelt. Summary of the Dagstuhl workshop on future directions in software engineering. *ACM SIGSOFT Software Engineering Notes*, 18(1):35–48, January 1993.

[5] N. T. Fletton and M. Munro. Redocumenting software systems using hypertext technology. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988),

pages 54–59. IEEE Computer Society Press (Order Number 879), October 1988.

[6] H. L. Ossher. A mechanism for specifying the structure of large, layered systems. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, 1987.

[7] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering,* SE-5(2):128–137, March 1979.

[8] V. Karakostas. Intelligent search and acquisition of business knowledge from programs. *Journal of Software Maintenance: Research and Practice*, 4:1–17, 1992.

[9] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Software*, 22(7):36–49, July 1989.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program undertsanding. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 27–43. IEEE Computer Society Press (Order Number 3780-02), November 1992.

[11] M. Shaw. Larger scale systems require higher-level abstractions. *ACM SIGSOFT Software Engineering Notes*, 14(3):143–146, May 1989. Proceedings of the Fifth International Workshop on Software Specification and Design.

[12] H. A. Müller. *Rigi – A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications.* PhD thesis, Rice University, August 1986.

[13] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments,* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992. In *ACM Software Engineering Notes*, 17(5).

[14] H. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *ICSE '10: Proceedings of the 10th International Conference on Software Engineering,* (Raffles City, Singapore; April 11-15, 1988), pages 80–86, April 1988. IEEE Computer Society Press (Order Number 849).

[15] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 1993. In press.

[16] G. Myers. *Reliable software through composite design.* Petrocelli/Charter, 1975.

[17] H. Müller. Verifying software quality criteria using an interactive graph editor. In *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference,* (Portland, Oregon; October 29-31, 1990), pages 228–241, October 1990. ACM Order Number 613920.

[18] M. A. Orgun, H. A. Müller, and S. R. Tilley. Discovering and evaluating subsystem structures. Technical Report DCS-194-IR, University of Victoria, April 1992.

[19] S. R. Tilley, M. J. Whitney, H. A. Müller, and M.-A. D. Storey. Personalized information structures. *SIGDOC '93: The 11th Annual International Conference on Systems Documentation,* (Waterloo, Ontario; October 5-8, 1993), pages 325–337, October 1993. ACM Order Number 6139330.

[20] H. Müller and J. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance 1990,* (San Diego, California; November 26-29, 1990), pages 12–19, November 1990. IEEE Computer Society Press (Order Number 2091).

[21] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS: A graph-oriented database system for (software) engineering applications. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 272–286, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[22] S. Paul. SCRUPLE: A reengineer's tool for source code search. In *CASCON'92: Proceedings of the 1992 CAS Conference,* (Toronto, Ontario; November 9-12, 1992), pages 329–345. IBM Canada Ltd., November, 1992.

[23] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1993. To be published.

[24] E. A. Idler. A visual scripting language. Master's thesis, University of Victoria, 1989.

[25] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, January 1990.

[26] R. Gopal. *On Supporting Software Evolution: Automatic Decomposition Schemes Based on Static and Dynamic Analysis of Programs.* PhD thesis, Vanderbilt University, 1989.

[27] K. B. Gallagher. Evaluating the Surgeon's Assistant: Results of a pilot study. In *CSM'92: Proceedings of the 1992 Conference on Software Maintenance,* (Orlando, Florida; November 9-12, 1992), pages 236–244. IEEE Computer Society Press (Order Number 2980), November 1992.

[28] R. Gopal, R. Prasad, and R. Gopal. Supporting systems maintenance with automatic decomposition schemes. In B. D. Shriver, editor, *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 507–516, January 1992.