Programmable Reverse Engineering[†]

Scott R. Tilley[‡] Kenny Wong Margaret-Anne D. Storey Hausi A. Müller

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, kenw, mstorey, hausi}@csr.uvic.ca

[†]This work was supported in part by the British Columbia Advanced Systems Institute, the IBM Software Solutions Toronto Laboratory, 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.

[‡]Author to whom correspondence should be sent.

Abstract

Program understanding can be enhanced using reverse engineering technologies. The understanding process is heavily dependent on both individuals and their specific cognitive abilities, and on the set of facilities provided by the program understanding environment. Unfortunately, most reverse engineering tools provide a fixed palette of extraction, selection, and organization techniques. This paper describes a *programmable* approach to reverse engineering. The approach uses a scripting language that enables users to write their own routines for common reverse engineering activities such as graph layout, metrics, and subsystem decomposition, thereby extending the capabilities of the reverse engineering toolset to better suit their needs. A programmable environment supported by this approach subsumes existing reverse engineering systems by being able to simulate facets of each one.

Keywords: End-user programming, extensibility, program understanding, reverse engineering, scripting.

1 Introduction

It has been estimated that fifty to ninety percent of software maintenance work is devoted to *program understanding* [1]. This is especially problematic for older software systems that are inherently difficult to understand (and hence, to maintain) due in part to their size, the lack of high-quality documentation, and their evolution history. Easing the understanding process can therefore have a significant impact on reducing the costs, economic and otherwise, of software evolution.

One of the most promising approaches to the problem of program understanding for software evolution is *reverse engineering*. Reverse engineering technologies have been proposed to help refurbish and maintain software systems. The process of reverse engineering identifies the system's current components, discovers their dependencies, and generates abstractions to manage complexity. It involves parsing the source code of the subject system and storing the extracted artifacts in a repository. To facilitate the understanding process, the subject system is represented in a form where many of its structural and functional characteristics can be analyzed. As maintenance and re-engineering costs for large legacy software systems increase, the importance of reverse engineering will grow accordingly.

No rigid program understanding environment will ever be suitable for all users in all software maintenance tasks. Users' disparate cognitive abilities and their diverse approaches to program understanding preclude the use of a static suite of extraction, selection, and organization techniques. While much research has been done on modeling how software engineers understand programs, the attitude that seems prevalent to many tool builders is that "if programmers would just learn to understand code the way they ought to" (i.e., the way the tools work), the code comprehension problem would be solved [2]. Such a builder-oriented view is unsuitable for the analysis of large programs [3]. Instead, we should provide an environment that supports the users' view; tools and interfaces that *support* the natural process of program understanding—not hinder it.

To meet this goal, a successful reverse engineering environment must provide mechanisms through which users can provide additional functionality. For example, it should be possible to extend the search and selection operations with user-defined algorithms or to interface with external tools. It is desirable to allow users as much freedom as possible in configuring the system to their liking. This configurability includes extending two key aspects of the environment: the core components [4] and the user interface [5].

This paper describes an approach to supporting program understanding through reverse engineering. In particular, the paper focuses on the *programmable* aspects of a reverse engineering environment. Through such a flexible environment, the application domain need not be limited to one area. Our approach uses a scripting language that enables users to write their own routines for common reverse engineering activities such as graph layout, metric and analysis, and subsystem decomposition. The underlying system supports the language-dependent extraction of software artifacts in 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-guided manner.

Section 2 overviews the end-user programming phenomenon. Section 3 examines several desirable aspects of a reverse engineering environment. Section 4 discusses the programmable editor, outlines its overall architecture, and describes its core components. This leads to Section 5 which provides examples of extending these core components through user-defined scripts. Section 6 summarizes our contributions to the field.

2 End user programming

"It's only a small matter of programming ..."

— Bonnie Nardi [6].

It has been repeatedly shown that no matter how much designers and programmers try to anticipate and provide for users' needs, the effort will always fall short. This is not the fault of the designers and programmers; in general, it is impossible to know in advance all that will be needed. Noone can foresee all the situations their systems and applications will encounter; customizations, extensions, and new applications inevitably become necessary. This lack of flexibility forces users to spend much of their time transferring their domain knowledge to the application programmer. A better approach would be to allow the users to exploit the domain knowledge themselves. Hence, the goal is to provide the user with as much flexibility as possible in customizing the environment to suit their needs. One way of providing this functionality is by providing to end user the ability to *program* the application.

2.1 Benefits of end user programming

The traditional definition of programming takes the programmer's perspective: an activity in which instructions are written in a language that is compiled or interpreted into the application. A better definition from a user's perspective is to define programming by its objectives: to create an application that serves some function for the user. This task-specific approach attempts to capitalize on users' strengths by exploiting their skills and interests [6].

This goal has lead to a veritable flood of end-user programmable applications, virtually all of which are task-specific. There are several major automation, customization, and integration benefits to this approach. Complex tasks and work processes can be automated for more consistency and repeatability. Administrative and routine tasks can be automated for better productivity. Agents and background jobs can be programmed to perform tasks unattended, at scheduled times or when certain conditions are satisfied. Customized applications can be assembled using a scripting language to transparently integrate existing capabilities. User interfaces and preference settings can be configured and adjusted as desired.

Such programmability has proven effective in numerous application domains, including hypertext systems (for example, [7]), custom database applications, Computer-Aided Design (CAD) systems, communications products, and statistical analysis packages. Custom database applications involve data entry screens that can be tailored to resemble paper forms and programmed for improved consistency and correctness of entries. CAD systems employ parametric programming whereby diagrammatic plans can be parameterized with design and engineering constraints. Communications products often use a recordable command language to automate the drudgery of dialup and log-on sequences. Number-crunching packages such as Mathematica [8] offer a wide range of mathematical, statistical, and combinatorial routines that users can build upon in their analysis programs.

2.2 End-user programmable applications

Perhaps the most widely used end-user programmable application available on personal computers is the spreadsheet. The entry of values, formulas, and dependencies in a spreadsheet is a form of programming well-suited to end-user exploitation. As feature-laden as they are, spreadsheets such as Lotus 1-2-3 and Microsoft Excel also offer macro languages for expressing more elaborate sequences of computation. Business application suites are also beginning to provide scripting languages as a kind of unifying coordination mechanism. For example, Microsoft intends to use Visual Basic for Applications (VBA) [9] as a common extension language for its suite of office applications. Similarly, Lotus is planning a cross-application scripting language for their product offerings. The customizable user interface of WordPerfect is being extolled as a significant value-added feature. In these suites, users have the power to tailor and configure menu bars, status bars, style ribbons, and scroll bars.

Text editors are a classic example of the difficulties that application designers face because of diverse user tastes and preferences. The question of which text editor is best is often the topic of seemingly unending debate. Devotees of the *vi* editor that comes with UNIX [10] trumpet that it works well with other UNIX tools. Disciples of *emacs* [11] have praised the capabilities of its built-in extension and customization facilities, provided through a variant of Lisp. Emacs was constructed through the composition of separate and independent functions. By providing access to the same language that was used to implement it, the user can customize the editor by adding new or replacing existing commands and previous extensions. Its extensibility has been proven: code browsers, mail readers, and news readers have been constructed on top of the base editor. Another extensible text editor is IBM's Xedit. It allows users to write REXX scripts to extend its functionality beyond simple text processing. The choice of REXX as the scripting language was guided by the fact that it is also the scripting language of choice on VM/CMS, the original host operating system for Xedit (it has since been ported to other platforms). Followers of more graphical user interfaces look to editors such as Alpha on the Macintosh, or BRIEF (Basic Reconfigurable Interactive Editing Facility) [12] for DOS and OS/2. Alpha incorporates an extension language based on Tcl [13]. BRIEF's basic premise is programmability: users can customize the editor by changing keystroke assignments and modifying existing commands. To support this end-user programmability aspect, it provides a flexible macro language, a completely reconfigurable set of key bindings, and the ability to run other programs inside an editing session.

Operating systems represent another application area that incorporates end-user programmability to facilitate ease of use. IBM's MVS/ESA operating system is extremely customizable, but it requires a system programmer working in System/370 assembler and JCL to exploit this capability. A more accessible environment that is adaptable, extensible, and nonspecialized is UNIX. It provides a set of core programs for common tasks. More complex tools are added to the UNIX toolkit by combining and connecting existing tools in various combinations. The tools are connected by a command language interpreter: the shell. The shell is an ordinary program, not a system program. It can be changed or replaced by other versions if the user so desires. In this way it is similar to emacs: core functionality may be extended or replaced by the end user as needed.

2.3 Summary

The power of end user programming is continuing to be developed. Existing systems that offer end users the capability to extend and customize their applications typically do so through a task-specific programming language. Such languages often lack the power of more general-purpose programming languages, but they also lack the steep learning curve.

While extensible to varying degrees, text editors such as vi, emacs, and BRIEF suffer from sometimes cryptic command sequences and the need for the user to learn yet another programming language. Even worse, the language they must learn is different for each application. Editors such as Alpha and Xedit rectify this problem somewhat by using embedded scripting languages that are used in other applications. Some word processor packages are also taking this approach of using common cross-platform and cross-application extension languages.

Given that end user programming has proven so successful in other application areas, it seems natural to use the same approach in the program understanding domain. Particularly since there are so many program understanding activities that lend themselves to customization. To do so, one must first know what the users of such a system want to accomplish. The next section discusses some of the desirable aspects of a reverse engineering environment. These aspects are key design issues that must be addressed to support and meet these goals via end user programming in the program understanding arena.

3 Desirable aspects of a reverse engineering environment

Programmers make use of programming knowledge, domain knowledge, and comprehension strategies when attempting to understand a program. They extract syntactic knowledge from the source code and rely on programming knowledge to form semantic abstractions. Brooks' work on the theory of domain bridging [14] describes the programming process as one of constructing mappings from a problem domain to an implementation domain, possibly through multiple levels of abstraction. Program understanding then involves reconstructing part or all of these mappings. This process is expectation driven, and proceeds by creation, confirmation, and refinement of hypotheses. It requires both intra-domain and inter-domain knowledge. A problem with this reverse mapping approach is that mapping from application to implementation is one-to-many, as there are many ways of implementing a concept.

To aid code comprehension, a reverse engineering environment must make the reverse mapping process easier by recovering lost information and making implicit information explicit. To do so, the environment must be flexible in three areas: (1) it must support different cognitive models and understanding processes; (2) it must provide an extensible toolset; and (3) it must be applicable to multiple domains, including "real-world" software systems. These three requirements form a *design space* [15] for reverse engineering tools, as illustrated in Figure 1. Each of these areas is discussed in more detail below.

3.1 Cognitive models and the understanding process

It is hard for any application designer to predict all the ways in which the application will be used. In a reverse engineering environment, the main goal is to facilitate code comprehension. Since 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 comprehension.

Two common approaches to code comprehension often cited in the literature are a functional approach that emphasizes cognition by *what* the program does, and a behavioral approach that emphasizes *how* the program works. Both top-down and bottom-up comprehension models have been used in an attempt to define how a software engineer understands a program. However, case studies have shown that, in industry, maintainers of large-scale programs frequently switch between several comprehension strategies [16]. The tools and environment must support the diverse cognitive processes of program understanding, rather than impose a process that is not justified by a cognitive model other than that of the environment's developers.

While creating the semantic abstractions during the code comprehension process, it should be possible to include human input and expertise in the decision making. There is a tradeoff between what can be automated and what should or must be left to humans; the best solution lies in a combination of the two. Hence, the construction of abstract representations manually, semi-automatically, or automatically (where applicable), should be possible. Due to user-control, the comprehension process can be based on diverse criteria such as business policies, tax laws, or other semantic information not directly accessible from the source code.

3.2 Extensible toolset

The most important goal for a successful reverse engineering environment is being able to aid users in solving their software maintenance problems. End-user extensibility of the system's functionality is one way of achieving this goal. The basic reverse engineering operations of gathering, organizing, and presenting information in terms that are useful to the user should not be fixed by the environment. The user should have the capability to provide their own tools for these activities.

3.2.1 Gathering information

Information extracted from the subject system's source code is used to identify its components and their dependencies. Users should be able to indicate what software artifacts they want extracted from the source code and be able to highlight important objects and dependencies, and de-emphasize immaterial ones. This functionality is not just important from an aesthetic point of view; it is also a matter of scalability.

Most reverse engineering systems parse source code and retain 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 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.

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 to look for—and what to ignore [17].

Once the artifacts have been identified and extracted from the source code, the next step in the reverse engineering process is to extract system abstractions and design information. This is usually accomplished with various clustering algorithms to aggregate lower-level objects into logical subsystems. The clustering criteria can take many forms, from completely domain-independent (such as those based strictly on graph-theoretic measures such as connectivity) to domain-dependent information (such as application naming conventions).

It should be possible to augment the search and selection operations built into the reverse engineering environment with user-defined algorithms, and to interface with external tools as required.

For example, 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 searching and clustering techniques, such as IRENE [18], SCRUPLE [19], and REFINE [20], and have the results of their searches made available to the user and the environment.

3.2.2 Organizing information

The parsing process builds a flat resource flow graph of the subject software, detailing components and specific relationships. The complexity of the graph can be reduced by organizing the constituent artifacts into layered hierarchies. Such hierarchies might be built using graph editors or outliners. The choice of whether or not to construct such hierarchies, and if so of what type, depends on the user and the domain. Moreover, algorithms are often more efficient on restricted variants of these hierarchies (e.g., trees and $(k, 2)$-partite graphs [21]). The environment must support a flexible and a sound data model with which the user can organize information, build hierarchies, and express operations efficiently.

3.2.3 Presenting information

The construction of hierarchies can be guided by partitioning the resource-flow graph based on established modularity principles such as *low coupling* and *strong cohesion* [22]. Exact interfaces and modularity quality measures (partition and encapsulation quality) may be programmed to evaluate the generated software hierarchies. Various hybrid metrics should also be programmable within the environment. The constructed software hierarchies should reflect semantics and not just pleasing graph layouts.

Most existing reverse engineering 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 will want something else. One cannot predict which aspects of a system are important for all users, and how these aspects should be gathered, organized, 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.

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.

3.3 Domain retargetability

The reverse engineering environment (and the approach supported by it) 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 over-burdened term. It refers to different *application* domains, such as banking or health information 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.

One way of maximizing the usefulness of a program understanding 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, this approach limits the system's usefulness to a particular domain. Using the same system on a different task, even one that is similar, may well 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 [23]. 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.

It is essential that any domain-retargetable 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 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 responsive, and the user interface must support the manipulation of very large graphs.

Many current reverse engineering environments support only relatively small programs. Others support just one programming language (or a subset of it), usually because their parsing system, database, and support environment are tightly coupled. 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.

3.4 Summary

A successful reverse engineering environment must be flexible with respect to cognitive models, toolset functionality, and domain applicability. The wide range of users' personal tastes and abilities mandate the integration of different techniques, technologies, and tools. To achieve high functionality, many systems are targeted toward a single application domain. 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.

One way of achieving this is by making the environment end-user programmable. The user would then be able to use the tools built into the environment, tools from third-party sources, or tools written by themselves. Communication in such an integrated environment can be achieved by scripts which each tool understands. Such an approach is described in the next section.

4 A programmable editor

Gathering, organizing, and presenting information were identified in Section 3 as basic operations of the reverse engineering process. To support end-user programmability of these operations, a core set of functional components must be made available to both the user and other tools. These core components must provide functionality upon which a more powerful and domain-specific toolset may be constructed. This section outlines the architecture of a programmable graph editor that meets these objectives.

As illustrated in Figure 2, the service architecture of the programmable editor is ring-based. Its components directly address each of the basic reverse engineering operations stated above. At the center of the architecture is the kernel, which is a script language interpreter. Just outside the kernel, the core provides a minimalist set of program understanding functionalities. Beyond the core is the human and tool interface ring which provides presentation services and access to external tools. The outer personality ring consists of domain-dependent scripts and other program understanding extensions. Beyond this outer ring one can have additional rings that further extend the capabilities of the environment.

4.1 The kernel

The kernel provides a consistent, system-wide basis for building scriptable and recordable functionality. It also serves as a router to coordinate control within the entire editor. Since the kernel is essentially the embeddable Tcl library, the scripting language used is Tcl. There are several advantages to using Tcl. It provides an application easy access to a powerful scripting language.

The implementation is interpreted; thus, there is no need to recompile when experimenting with a script. Moreover, scripts are easy to write and are generally fairly short. The language is easily extensible: it is possible for additional functionality to be written in a compiled language like C++, tied to a Tcl command, and invoked via the kernel's callback mechanism.

In the editor, all extensions to the Tcl language (including the core commands) are registered with the kernel. Since Tcl also provides primitives to access and coordinate external tools, even for those that were not written with scripting control in mind, the toolset available to the user is unlimited. The companion Tk library allows user-customizable user interface widgets with the Motif look-and-feel to be quickly built. Tcl commands can be tied to various Tk widgets and triggered on events such as keystrokes, mouse motions, button clicks, and menu selections. These features allow users to build personalized systems by tying together internal and external tools.

4.2 The core

The core ring provides fundamental program understanding capabilities in five categories: extraction, selection, organization, representation, and measures. The extraction component provides support for parsing source code and extracting software components and relationships, building a graph model that is managed by the representation component. The selection component provides support for selecting and gathering objects within the model. Taken together, extraction and selection contribute towards the basic reverse engineering goal of gathering information. The organization component provides the user with views of information within the representation model. The measures component computes basic statistics and metrics using the model.

The core components are registered with the kernel and available to the user and external tools. Extensions to the core make use of these capabilities to compose more powerful tools. These extensions are also registered with the kernel as Tcl commands.

4.3 The interface ring

The interface ring separates user interface concerns from the computational concerns of the core and kernel. It provides interfaces to the core components for both humans and tools. The human interface supports customization of the user interface through Tk, although other presentation systems may also be used. This ring also provides the application program interface by which external tools can access the functionality of the core and the extended functionality registered with the kernel.

4.4 The personality ring

In the personality ring, a user can extend the built-in core operations with algorithms for graph layout, complexity measures, pattern matching, slicing, clustering, and so on using scripts. Moreover, the user interface can be further tailored (if desired) to reflect new application domains. The capability of molding and adapting the editor to different domains is necessary for successful program understanding systems.

4.5 Summary

The architecture of the programmable graph editor directly reflects the main services needed by a program understanding system. The structure is stratified into rings, with the script language kernel at the center. The core provides the backbone upon which extensions can be built. The interface ring supports user interface customizability and access to editor functionality by external tools. The personality ring supports domain-retargetability.

The next section illustrates the use of a programmable editor that uses the architecture discussed in this section. The examples shown are taken from activities that take place in almost every reverse engineering scenario.

5 Examples

Program understanding is made up of many different activities, each of which is composed of operations built upon the core facilities discussed in Section 4. To illustrate how end user programmability can aid the program understanding process, three basic understanding operations are used as examples: (1) graph layout; (2) measures; and (3) subsystem decomposition. The operations are carried out using Rigi [24], a programmable reverse engineering environment that supports our approach.

5.1 Graph layout

Visualizing artifact dependencies is a common operation during program understanding. The layout of the graphical representation of the subject software system can greatly aid in its understanding. Aesthetically pleasing graph layouts are sometimes difficult to construct, either by hand or through an automatic algorithm. However, what is "pleasing" is subjective.

Figure 3 shows three different layouts of (part of) a COBOL program. In the figure, the nodes represent paragraphs, and the arcs connecting the nodes represent perform statements. The view of the program in the top-right window is a tree layout [25] of the call (perform) graph. The tree layout operation is currently built into the editor. The view in the left window is a Sugiyama layout [26] of the same graph, while the bottom-right window shows a spring layout [27].

The spring and Sugiyama layouts were done by exporting the graph representation and running the layout algorithms off-line, using stand-alone programs provided as part of the GraphEd [28] package. The script used to interface to these external layout algorithms is shown in Figure 4. It writes the graph in a form acceptable to the layout algorithms (GraphEd Format), executes the appropriate layout algorithm, and loads the result back into the editor.

5.2 Measures

Measurements are used during reverse engineering for a variety of purposes. As discussed in Section 3.2.3, measures such as coupling and cohesion can be used to guide subsystem decomposition. Once subsystem structures have been constructed, graph-theoretic measures such as cyclomatic complexity [29], graph quality [30], and structural complexity [31] may be used to refine the subsystems' hierarchies.

Figure 5 illustrates the use of scripts to compute a graph's cyclomatic complexity. In this way, one of the fundamental operations in reverse engineering, analysis, is aided by giving the end user access to the wide variety of software metrics and tools that implement them.

5.3 Subsystem decomposition

Subsystem decomposition is the process of iteratively reducing the complexity of the subject system by clustering artifacts based on some selection criteria and composing the selected artifacts into subsystems. In this way, a layered graph structure is constructed. At the highest level are subsystems representing major components of the subject system.

Using various measures as a guide, multiple co-existing hierarchical structures may be constructed. The programmability aspect enables the user to experiment with various decompositions. For example, a long-term goal of reverse engineering might be actual physical re-modularization (or re-engineering) of a system 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.

A common decomposition is one based on naming conventions. This is often useful when the maintainer is attempting to get an initial understanding of the subject system, and when it is implemented using specific identifier naming rules. While this is perhaps one of the simplest decomposition strategies, it is also one of the most intuitive. As an example, Figure 6 shows a script to decompose a program according to application-specific naming conventions. The result of running this script on part of a multi-million-line PL/AS program is shown in Figure 7.

6 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 adapt to various application domains, implementation languages, and working environments.

This paper described an end-user programmable approach to reverse engineering, which gives individuals the ability to tailor their environment to suit their needs. The approach is supported by an architecture that provides a core set of functions for basic program understanding operations. These functions may be extended by the user to perform more advanced gathering, organization, and presentation operations. External tools may also be easily integrated into this process. Rather than forcing users to work within a restricted and fixed environment provided by the tool builder, they can customize and extend it as they see fit.

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. Suites of program understanding techniques may be gathered, created, and maintained in script libraries.

Acknowledgments

The support work of Michael Whitney and Brian Corrie is greatly appreciated. They worked on the Rigi implementation, wrote and experimented with the scripting interface, and provided useful comments on early drafts of this paper.

Trademarks

IBM, System/370, and MVS/ESA are trademarks of International Business Machines Corporation.

The Software Refinery and REFINE and trademarks of Reasoning Systems Inc.

UNIX is a registered trademark licensed exclusively by X/Open Company Ltd.

References

[1] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[2] A. von Mayrhauser and A. M. Vans. From code understanding needs to reverse engineering tools capabilities. 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 230–239, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[3] E. Buss and J. Henshaw. Experiences in program understanding. Technical Report TR-74.105, IBM Canada Ltd. Centre for Advanced Studies, July 1992.

[4] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *CSM '93: The 1993 International Conference on Software Maintenance,* (Montréal, Québec; September 27-30, 1993), pages 142–151, September 1993. IEEE Computer Society Press (Order Number 4600-02).

[5] S. R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. In *International Conference on Software Maintenance (ICSM '94),* (Victoria, BC; September 19-23, 1994), September 1994. To appear.

[6] B. A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* MIT Press, 1993.

[7] P. D. Stotts and R. Furuta. Dynamic adaptation of hypertext structure. In *Proceedings of Hypertext '91* (San Antonio, Texas; December 15-18, 1991), pages 219–231, December 1991. ACM Order Number 614910.

[8] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, 2nd edition, 1991.

[9] T. J. Biggerstaff. Directions in software development & maintenance. University of Victoria invited talk, December 9, 1993.

[10] B. W. Kernighan and J. R. Mashey. The UNIX programming environment. *Computer*, 14(4):25–34, April 1981.

[11] R. M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. In *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation,* (Portland, Oregon; June, 1981), pages 147–156, Jne 1981.

[12] The BRIEF DOS-OS/2 user's guide. Part of the BRIEF software package.

[13] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1994.

[14] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[15] T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute; Carnegie-Mellon University, November 1990.

[16] A. von Mayrhauser and A. M. Vans. An industrial experience with an integrated code comprehension model. Technical Report CS-92-205, Colorado State University, 1992.

[17] 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.

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

[19] S. Paul and A. Prakash. Source code retrieval using program patterns. In *CASE'92: Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering,* (Montréal, Québec; July 6-10, 1992), pages 95–105, July 1992.

[20] G. Kotik and L. Markosian. Program transformation: The key to automating software maintenance and re-engineering. Technical report, Reasoning Systems, Inc., 1991.

[21] 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).

[22] G. Myers. *Reliable Software Through Composite Design.* Petrocelli/Charter, 1975.

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

[24] 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).

[25] E. Reingold and J. Tilford. Tidier drawing of trees. *IEEE Transactions on Systems, Man, and Cybernetics*, SE-7(2), March 1981.

[26] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(4):109–125, 1981.

[27] T. Fruchtermann and E. Reingold. Graph drawing by force-directed placement. Technical Report UIUC CDS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.

[28] M. Himsolt. GraphEd: The design and implementation of a graph editor. Part of the *GraphEd* distribution kit, Universität Passau, 1993.

[29] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-7(4):308–320, September 1976.

[30] 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.

[31] D. N. Card. Designing software for producibility. *Journal of Systems and Software*, 17(3):219–225, March 1992.

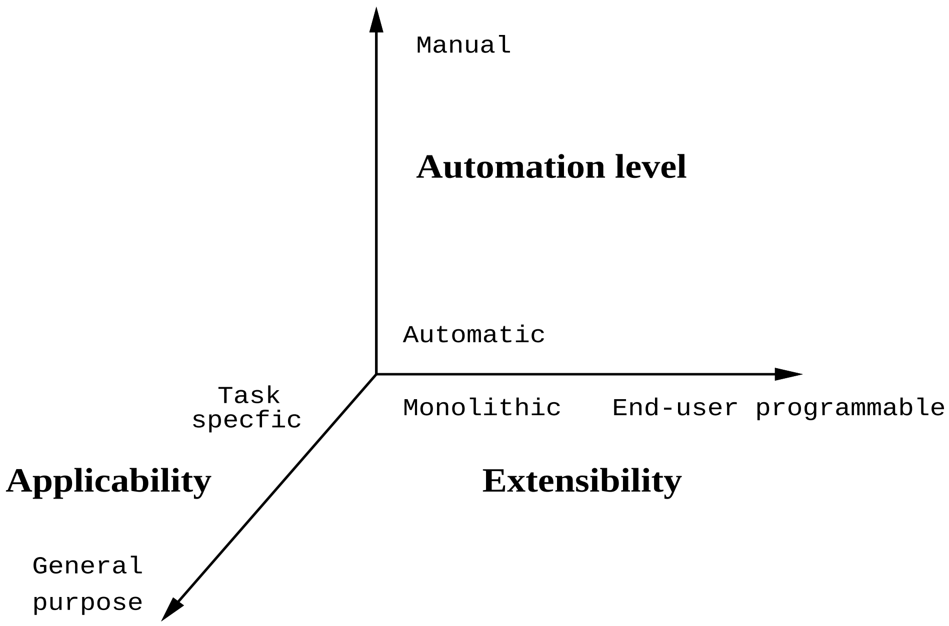

Figure 1: Reverse engineering design space

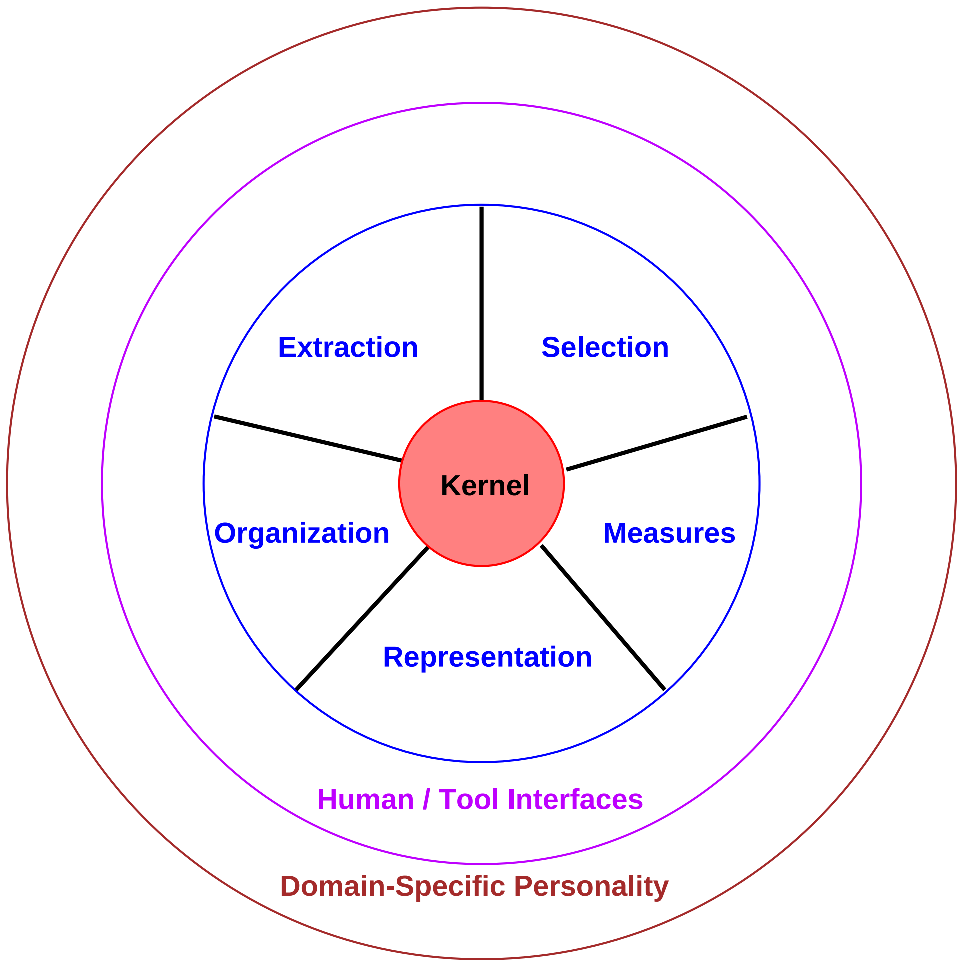

Figure 2: The programmable graph editor's ring architecture

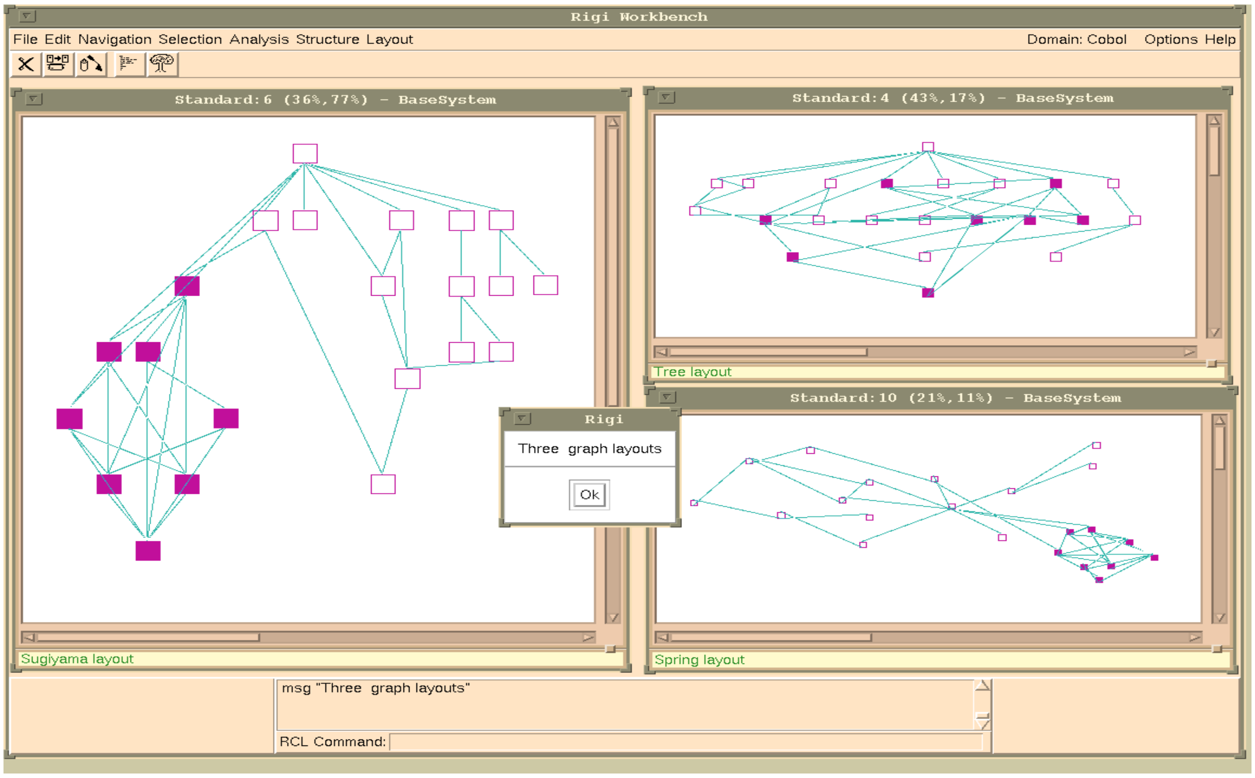

Figure 3: Various graph layouts of a COBOL program

```
# Layout algorithms [SRT 16Jun94]

# layout(): Use an off-line layout algorithm based on GraphEd's .gef format.
proc layout { program {window 0} {arctype any} } {
  if {$window == 0} {set window [get_window_id]}

  set graphin [format "/tmp/%s-in" $window]
  set graphout [format "/tmp/%s-out" $window]

  # Write, layout, read
  writeGEF $graphin $window $arctype
  exec $program < $graphin.gef > $graphout.gef
  readGEF $graphout $window

  # Cleanup
  exec rm $graphin.gef
  exec rm $graphout.gef
}


# spring(): Run spring layout algorithm. (Graph must be conected.)
proc spring { {window 0} {arctype any} } {
  if {$window == 0} {set window [get_window_id]}

  if {[is_connected $window $arctype]} {
    layout gel-spring $window $arctype
  } else {
    open_message_panel "Error: The graph is not connected."
  }
}


# sugiyama(): Run sugiyama layout algorithm. (No multiple edges.)
proc sugiyama { {window 0} {arctype any} } {
  if {$window == 0} {set window [get_window_id]}

  layout gel-sugiyama $window $arctype
}
```

Figure 4: Script to perform off-line graph layout

```
# Metrics

# McCabe's cyclomatic complexity [SRT 20Feb94]

# Given a graph G, the cyclomatic complexity measure V of G:
#   V(G) = e - n + 2p,
# where:
#   e = number of edges in G,
#   n = number of nodes in G,
#   p = number of disconnected subgraphs of G.
# Leaves results in VG().

proc mccabe { {arctype any} } {
  global VG

  # Boundary condition
  set n [num_nodes_of_current_window]
  if {$n == 0} {return 0}

  set e [num_arcs_of_current_window]
  if {$e > 1} {
    set edgecount 0
    set selectedarc [first_arc_of_current_window]
    for {set i $e} {$i > 0} {incr i -1} {
      if {$arctype == "any"} {
        incr edgecount 1
      } else {
        if {[get_arctypename $selectedarc] == $arctype} {incr edgecount 1}
      }
      set selectedarc [next_arc_of_current_window]
    }
    set e $edgecount
  }

  # p = number of subgraphs in the forest of disconnected subgraphs
  # (i.e., the number of connected components, computed by cc()).
  set p [cc]

  deselect

  # Calculate V(G)
  set VG(e) $e
  set VG(n) $n
  set VG(p) $p
  set VG(result) [expr $e-$n+2*$p]

  return $VG(result)
}
```

Figure 5: Script to compute cyclomatic complexity

```
# Usage: BUILD_SUBSYSTEMS prefix 1

# global variable for depth of recursion for creating subsystems
set limit 2

proc BUILD_SUBSYSTEMS { substring counter } {
   global limit
   if { $counter > $limit } {
      GRID_LAYOUT
      return
   }

   scan "A" "%c" char
   scan "Z" "%c" Zchar
   while { $char <= $Zchar } {
      set string [format "$substring%c" $char]
      CREATE_SUBSYSTEM "$string" $counter
      incr char
   }
}

proc CREATE_SUBSYSTEM { name counter } {
   set numnodes [ GREP $name ]
   if { $numnodes > 1 } {
      set parent_window [ GET_ID_CURRENT_WINDOW ]
      COLLAPSE
      RENAME $name
      set window [ OPEN $name ]
      BUILD_SUBSYSTEMS [ expr $counter + 1 ]
      CLOSE_WINDOW $window
      SELECT_WINDOW $parent_window
   }
}
```

Figure 6: Automatic subsystem decomposition using naming conventions

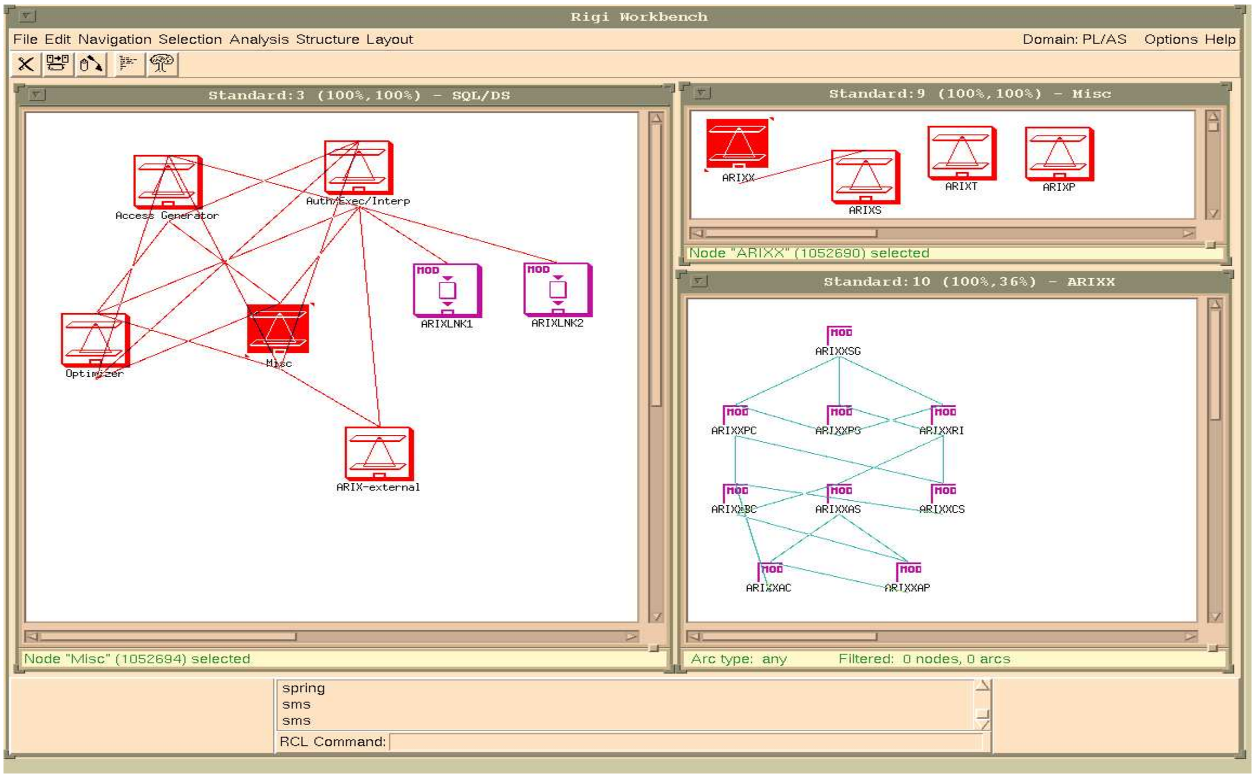

Figure 7: Result of decomposing based on naming conventions