Structural Redocumentation: A Case Study[†]

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

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: {`kenw, stilley, hausi, mstorey`}`@csr.uvic.ca`

Abstract

Documentation has traditionally played a key role as an aid in program understanding. However, most documentation is "in-the-small," describing the program at the algorithm and data structure level. For large, legacy software systems, one needs "in-the-large" documentation describing the high-level structural aspects of the software system's architecture from multiple perspectives. One way of producing such structural documentation for existing software systems is to use reverse engineering technologies. This paper describes a case study in *structural redocumentation*: an analysis of SQL/DS (a multi-million line relational database system) using a flexible reverse engineering approach developed as part of the Rigi project.

Keywords: Documentation, legacy software, program understanding, reverse engineering, software architecture.

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

Submitted to *IEEE Software*, Special issue on legacy software systems (January 1995).

1 Introduction

> Programmers have become part historian, part detective, and part clairvoyant.
>
> — Thomas A. Corbi, IBM [1].

Challenges in rediscovering system structure

Design may be difficult, but reconstructing and effectively (re)documenting the design of existing software systems is even more difficult. Recognizing abstractions in real-world systems is as crucial as designing adequate abstractions for new ones. This is especially true for *legacy* software systems written 10–25 years ago, which are often in poor condition because of prolonged, sometimes dramatic (even traumatic) maintenance. Evolving over many years, legacy systems embody substantial corporate knowledge and cannot be replaced without reliving their entire maintenance history. Thus, managing long-term software evolution is critical, especially considering the economic value of these systems.

Understanding system structure

It is widely accepted that over fifty percent of software evolution work is devoted to program understanding. Documentation has traditionally served an important role in this regard. There are, however, significant differences in documentation needs for software systems of vastly different scales (1,000 lines versus 1,000,000 lines). Most software documentation is *in-the-small*, since it typically describes the program at the algorithm and data structure level. For large legacy systems, an understanding of the structural aspects of the system's architecture is more important than any single algorithmic component.

Program understanding is especially problematic for software engineers and technical managers responsible for the maintenance of such systems. The documentation that exists for these systems usually describes isolated parts of the system; it does not describe the overall architecture. Moreover, the documentation is often scattered throughout the system and on different media. It is left to maintenance personnel to explore the low-level source code and piece together disparate information to form high-level structural models. Manually creating just one such architectural document is always arduous; creating the necessary documents that describe the architecture from multiple points of view is often impossible. Yet it is exactly this sort of *in-the-large* documentation that is needed to expose the structure of large software systems.

Using reverse engineering to discover software structure

Software structure is the collection of artifacts used by software engineers when forming mental models of software systems. These artifacts include software *components* such as procedures, modules, and interfaces; *dependencies* among components such as client-supplier, inheritance, and control-flow; and *attributes* such as component type, interface size, and interconnection strength. The structure of a system is the organization and interaction of these artifacts [2]. One computer-aided technique of reconstructing structural models is *reverse engineering.*

The process of reverse engineering identifies the system's current components, discovers their dependencies, and generates abstractions to manage complexity. This understanding can then improve subsequent development, ease maintenance and re-engineering, and aid project management. Using reverse engineering to reconstruct the architectural aspects of software may be termed *structural redocumentation.* As a result, the overall "gestalt" of the subject system can be derived, and some of its architectural design information can be recaptured. In addition, structural redocumentation does not involve physically restructuring the code (although this might be a desirable outcome).

Real-world experiences

The reverse-engineering approach developed under the Rigi[1] project has been successfully applied to several real-world software systems. These include a physician's patient-record information system (written in COBOL), a control program for a particle accelerator (written in C), and numerous UNIX[2] utilities. Early experience has shown that we can produce views that are compatible with the mental models used by the maintainers of the subject software.

These maintainers benefitted from the documentation produced by the Rigi system in several ways. First, they were able to see, in visual and concrete form, the logical software structure previously held only in their minds. Second, the views highlighted critical areas of the software structure that needed more attention, such as central components that have a large number of incident dependencies. Third, the views provided an objective basis for discussion and software maintenance, since they are based on the actual source code instead of out-of-date system documentation. Fourth, the views verified that the software structure of their system was, at least, understandable to an experienced analyst from the outside.

Previously, the largest program analyzed using this methodology was about 120,000 lines of code. While such a program is reasonably large in an academic setting, it is not exceptional for commercial legacy software. It was not until the challenge of redocumenting SQL/DS that we began to validate our approach effectively.

Outline

The next section outlines the Rigi system and methodology. Section 3 introduces the subject software of a case study in structural redocumentation: a large relational database management system called SQL/DS.[3]

[1] Rigi is named after a mountain in central Switzerland.
[2] UNIX is a trademark of Unix System Laboratories, Inc.
[3] SQL/DS is a trademark of International Business Machines Corporation.

This overview is followed by a description of the changes to the Rigi system needed to analyze SQL/DS. Section 4 discusses the results of our case study, including feedback from the management and development teams of the SQL/DS product. Finally, Section 5 summarizes our results and briefly outlines future work.

2 The Rigi environment for structural understanding

Rigi is a flexible environment for architectural understanding under development at the University of Victoria. It provides the following components of a reverse engineering system:

- a parsing system to support the common imperative programming languages[4] of legacy software,
- a repository to store the information extracted from the source code,
- an interactive, window-oriented graph editor to manipulate program representations.

Phases of structural redocumentation

In Rigi, the first phase of structural redocumentation is automatic, and involves parsing the source code of the legacy system and storing the extracted artifacts in the repository. This produces a flat resource-flow graph of the software. Software maintainers can use this graph to represent the structural dependencies of interest, such as function calls and data accesses. To manage the complexity, the second phase involves human pattern recognition skills and features language-independent subsystem composition techniques to generate multiple layered hierarchies for higher-level abstractions [3]. For example, the analyst can cluster functions into subsystems according to business rules or by accepted principles of software modularity, providing the multiple, alternative perspectives needed for maintaining the software.

[4]Such as C and COBOL. The Rigi system also includes a parser for LaTeX to support the understanding of documentation.

Subsystem composition is a recursive process of grouping building blocks such as data types, procedures, and other components into composite subsystems to help manage the complexity of understanding the software structure. The composition criterion depends on the application. For program understanding purposes, the process is guided by partitioning the resource-flow graph based on established modularity principles such as *low coupling* and *strong cohesion* [4]. Exact interfaces and modularity quality measures are used to evaluate the generated software hierarchies.

What Rigi provides to deal with legacy software

There are several requirements of a reverse engineering tool for dealing with large legacy software. Our research has focused on meeting these requirements, which include:

Dynamic views: Rigi presents structural documentation using a collection of "views." A view is a bundle of visual and textual frames that contain, for example, resource flow graphs, overviews, projections, exact interfaces, and annotations. A view is similar to a database view and is a dynamic snapshot that reflects the current reverse engineering state. Because views are ultimately based on the underlying source code, they remain up-to-date.

Flexibility: Because program understanding involves many different facets and applications, it is wise to make the approach as flexible as possible for use in many different domains. Most reverse engineering tools provide a fixed set of extraction, selection, filtering, organization, documentation, and representation techniques. We provide a scripting language that allows analysts to customize, combine, and automate these activities in novel ways. For example, analysts have used this language to express or access additional software metrics, clustering strategies, and graph layout algorithms.

Human input: There is a tradeoff in program understanding environments between what can be automated and what should (or must) be left to humans. The best solution lies in a combination of the two. The Rigi approach relies heavily on the experience of the analyst using it; the analyst makes all the important decisions. For example, as the software engineer forms subsystems based on various high-level criteria, the Rigi system can offer selection and search algorithms based on aspects such as graph connectivity, component type, and dependency type, and provide statistics such as exact interfaces between subsystems and graph quality metrics. Nevertheless, the process is one of synergy as the analyst also learns and discovers interesting relationships by exploring software systems using the environment. We advocate a "hands-on" approach to reverse engineering to help transfer the constructed abstractions into the minds of the software engineers.

Multiple views: Because the user is in charge, the subsystem composition process can be based on diverse criteria, such as business rules, tax laws, requirements, or other semantic information. These alternative and orthogonal *decompositions* may exist simultaneously under the structural representation supported by Rigi. Views can accurately capture co-existing architectural decompositions, providing many different perspectives for later inspection. In effect, multiple, virtual representations of the software's architecture can be created, manipulated, and saved.

Scalability: To deal effectively with legacy software, we must train program understanding tools and methods on large, multi-million line source codes. Techniques that work on toy projects often do not scale up. Our current scalability objective is to analyze systems of up to five million lines of code.

3 Analyzing the source code of SQL/DS

This section introduces the subject software of the case study and describes the changes to the Rigi system needed to handle the analysis.

The evolution of SQL/DS

SQL/DS (Structured Query Language/Data System) is a large relational database management system that has evolved since 1976. It is based on a research prototype and has undergone numerous revisions since the first release in 1982. Originally written in PL/I to run on VM, SQL/DS is now over 2,000,000 lines of PL/AS code and runs on several different operating systems, including VM and VSE.[5] PL/AS is a proprietary IBM systems programming language that is PL/I-like and allows embedded S/370 assembler. Simultaneous support of SQL/DS for multiple releases on multiple operating systems requires multi-path code maintenance, increasing the difficulty for its many maintainers.

SQL/DS consists of about 1,300 compilation units, roughly split into three large systems (and several smaller ones). Because of the size and complex evolution, no individual alone can comprehend the entire program. Developers are forced to specialize in a particular component, even though the various components interact. Existing program documentation is also a problem; there is too much to maintain and keep current with the source code, and too much to read and digest. SQL/DS is a typical legacy software system: successful, mature, and supporting a large customer base while adapting to new environments and growing in functionality.

Extending Rigi

Since SQL/DS is written in a proprietary language, commercial off-the-shelf analysis tools are often unsuitable. This presented us with an enticing and rare opportunity to exercise our approach on a classic industrial legacy system and an excellent test of whether the tool and method would scale up. Before we could perform the analysis, however, some aspects of our tool had to be enhanced to respond to the unique challenges posed by SQL/DS: its proprietary implementation language, its size and complexity, and its application domain.

[5] S/370, VM/XA, VM/ESA, VSE/XA, and VSE/ESA are trademarks of International Business Machines Corporation.

Our initial work on analyzing the SQL/DS code exposed some shortcomings of the Rigi system. The sheer amount of code compelled changes to all three components of the environment, but mostly to the graph editor `rigiedit`. The parsing system `rigireverse` was augmented with a PL/AS interface and successfully processed the entire source code in an incremental manner. The repository `rigiserver` was able to handle the large graph structures produced by the parser.

Managing scale

The parsing system `rigireverse` is composed of several subsystems, one for each supported programming language. Each subsystem communicates with `rigireverse` via a tuple stream. Users can specify which artifacts to extract, and at various levels of detail. For example, an option selects whether the parser should extract calls to system routines. The ability to pinpoint specific subsets of the software system to be considered is important for scalability reasons.

After deciding what information to extract from the source code, we added a new parsing subsystem to handle PL/AS. Storing entire abstract syntax trees for such a large system would require several hundred megabytes of storage. While this level of detail may be necessary for tasks such as control-flow analyses or code optimization, it is not necessary for understanding and redocumenting the software architecture. For program understanding, it is important to build abstractions that emphasize important themes and suppress irrelevant details; deciding what to include and what to ignore is an art. Ignoring intra-procedural details, we reduced the repository size for a multi-million line program significantly, making a major difference when retrieving data interactively. For example, the generated database for SQL/DS is under two megabytes.

```
EXAMPLE: PROC(FOO,BAR)
         OPTIONS(ID('EXAMPLE  PLAS'),REENTRANT;

@EJECT ASM;
@MACINCL SYSLIB(MEMBER1);

DCL VERSION FIXED(15) CONSTANT(7);

CALL ARIXEDB(FETCH,RSIBASEP,RSIRSSRC);

IF RDATRACI >= '1' THEN DO;
  CALL ARIXETR(SCORDSCD);
  GEN CODE( DC H'6715');
END;

ALLDONE:
RETURN;
END EXAMPLE;
```

plas2c

```
/* File example.c created by plas2c */
/* from file example.plas on        */
/* Mon Jan 24 13:08:14 PST 1994.    */

void EXAMPLE()
{
  ARIXEDB();
  ARIXETR();

  return;
}
```

Figure 1: A PL/AS code fragment and its C representation

Handling a new proprietary language

Parsing PL/AS is problematic because the language is context-sensitive. However, it was not necessary for us to parse the entire language completely. Interested in the high-level architecture, we focused on extracting only external[6] procedure definitions and calls. For our initial experiments, we did not extract intra-module procedure calls, nor procedure calls to library routines or builtin functions.

The easiest way to add support for PL/AS within Rigi was to extract the relevant information from the SQL/DS source code using a collection of *csh*, *awk*, and *sed* scripts, translate this information to its skeletal representation in C, and feed the result into the existing C parser. In this way, the `rigireverse` program was isolated from most of the changes. In addition, by extracting just a subset of the available information in the PL/AS source, we immediately reduced one of the problems of scale. For example, a 400,000 line subsystem of SQL/DS is reduced by two orders of magnitude to only 2,000 lines of C. A sample PL/AS code fragment and its C equivalent (for Rigi's purposes) is shown in Figure 1.

The automatic parsing of all 1,303 modules of SQL/DS into Rigi took about two and three quarter hours

[6] PL/AS supports nested procedures.

and required roughly ten megabytes of virtual memory on an IBM RISC System/6000[7] M375. The resultant database uses 1.6 megabytes of disk space.

Scaling up the editor user interface

The graph editor `rigiedit` is the heart of our reverse engineering environment. As such, it is extremely important that it be easily usable and respond in real time to commands given by the user. This is one major aspect of the scalability of program understanding approaches. One of the challenges in editing graphical representations for programs such as SQL/DS is managing visual complexity.

We changed the editor so that the screen is not redrawn every time a single graph operation is carried out. Graphs with over 1,000 nodes and arcs need to be refreshed efficiently to avoid degrading interactive response time. Thus, we tuned the user interface, redesigning it to allow the user to batch sequences of operations and specify when to update a window.

Adding a scripting layer to the editor

A more dramatic change was needed in one of the 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 abilities are still much more powerful and flexible than "hard-wired" algorithms. In essence, the user should always be in control. However, many of the operations 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 automatically generated subsystem decompositions, but the decomposition itself could be made easier. These observations led us to introduce a scripting layer into the editor.

[7] Trademark of International Business Machines Corporation.

Previously, the graph editor consisted of two tightly-coupled subsystems: the user interface and the editor itself. All editing and selection operations were intermingled with operations for manipulating the user interface, such as window size, menu selection, and so forth. We separated the user interface from the graph editor and added a transparent intermediate layer to make the environment programmable.

Instead of writing yet another command language, we used Tcl [5]. It provides an extendable core language and was specifically written to be embedded into interactive windowing applications. Tcl is application-independent and provides two different interfaces: a textual interface to users who issue Tcl commands, and a procedural interface to the host application. In the new `rigiedit`, the Tcl interpreter sits between the graphical user interface and the graph editor. This integration process is more fully described in [6].

Incorporating and exploiting domain knowledge

Program understanding takes place within the context of a specific application domain. Aspects of the domain that affect reverse engineering include artifact representation, application semantics, and environmental concerns. We initially analyzed SQL/DS without using any domain-dependent knowledge. However, it shortly became clear that to make effective use of the extracted information we had to leverage existing informal application-specific domain knowledge.

Our approach to reverse engineering adapts to new application domains through scripting. This enables users to write customized routines for common activities such as artifact extraction, graph presentation, and object search and selection. This makes the system domain-retargetable. For the SQL/DS source code, we created a library of specific scripts to aid in our analysis. One such script is shown in Figure 2. It is used to construct an initial decomposition of the subsystems of SQL/DS based on existing documentation and the current physical modularization.

```
# Build a subsystem using naming conventions.
proc BUILD_SUBSYSTEM {name} {
  set numnodes [GREP "$name"]
  if {$numnodes > 1} {
    CREATE_NEW_SUBSYSTEM "$name"
  }
}


# Build Level 1 of SQL/DS
proc SQL_LEVEL1 {} {
  scan "A" "%c" char
  scan "Z" "%c" lastchar
  while { $char <= $lastchar } {
    # ARI is the product code of SQL/DS.
    set string [format "ARI%c" $char]
    BUILD_SUBSYSTEM $string
    incr char 1
  }
}
```

Figure 2: Domain-dependent script for SQL/DS

4 Lessons learned in the case study

The target audiences for our case study were the development and management teams of the SQL/DS product. The study was guided, in part, by the need to produce results directly applicable to them. An increased emphasis on product quality mandated a different approach to software maintenance than had previously been used.

For the individual subsystems, we proceeded to analyze their inter-module dependencies, summarizing them in a set of views depicting different architectural perspectives. We then presented these views to the development teams with a series of carefully designed one-hour demonstrations. These demonstrations allowed us to exhibit the structural views of the subsystems pertinent to a particular development group, allowed the audience to interact with the software structures using the views as starting points, and allowed individual developers to create new views on the fly to reflect and record specific domain knowledge. The case study involved three experiments.

Experiment I: Call graph

For our first experiment, we generated a view of the entire call graph without considering any SQL/DS-specific domain knowledge. The result was not as encouraging as we would have liked: the developers did not recognize the abstractions we generated, making it difficult for them to give us constructive feedback. This reaffirmed our belief that successful reverse engineering must do more than manipulate system representations independent of their domain; the results must add value for its customers. Informal information and application-specific knowledge provided by existing documentation and expert developers are rich sources of data that should be leveraged whenever possible.

Experiment II: Naming conventions

Our second experiment used product-naming conventions and existing physical modularizations to construct another set of views. The prefix ARI is the unique code for the SQL/DS product, hence all module names begin with these three letters. The fourth letter in each module's name represents the physical subsystem to which the artifact belongs, the fifth letter represents further subsystem refinement, and so on. This information was captured by Rigi using a series of scripts and views, resulting in decompositions the developers readily recognized. These views were of value to the developers because they established a common ground for further discussions and analysis. Moreover, the developers were able to use them to verify their system documentation as well as to suggest where our decomposition did not fully conform their mental models.

One such view created using Rigi is shown in Figure 3. It contains three different windows, each with icons representing different components of the software system. Arcs connecting icons represent resource-flow relationships between components, such as a procedure call. The arcs are typed and give rise to a multi-layered semantic network representing the many different dependencies within the system. The top

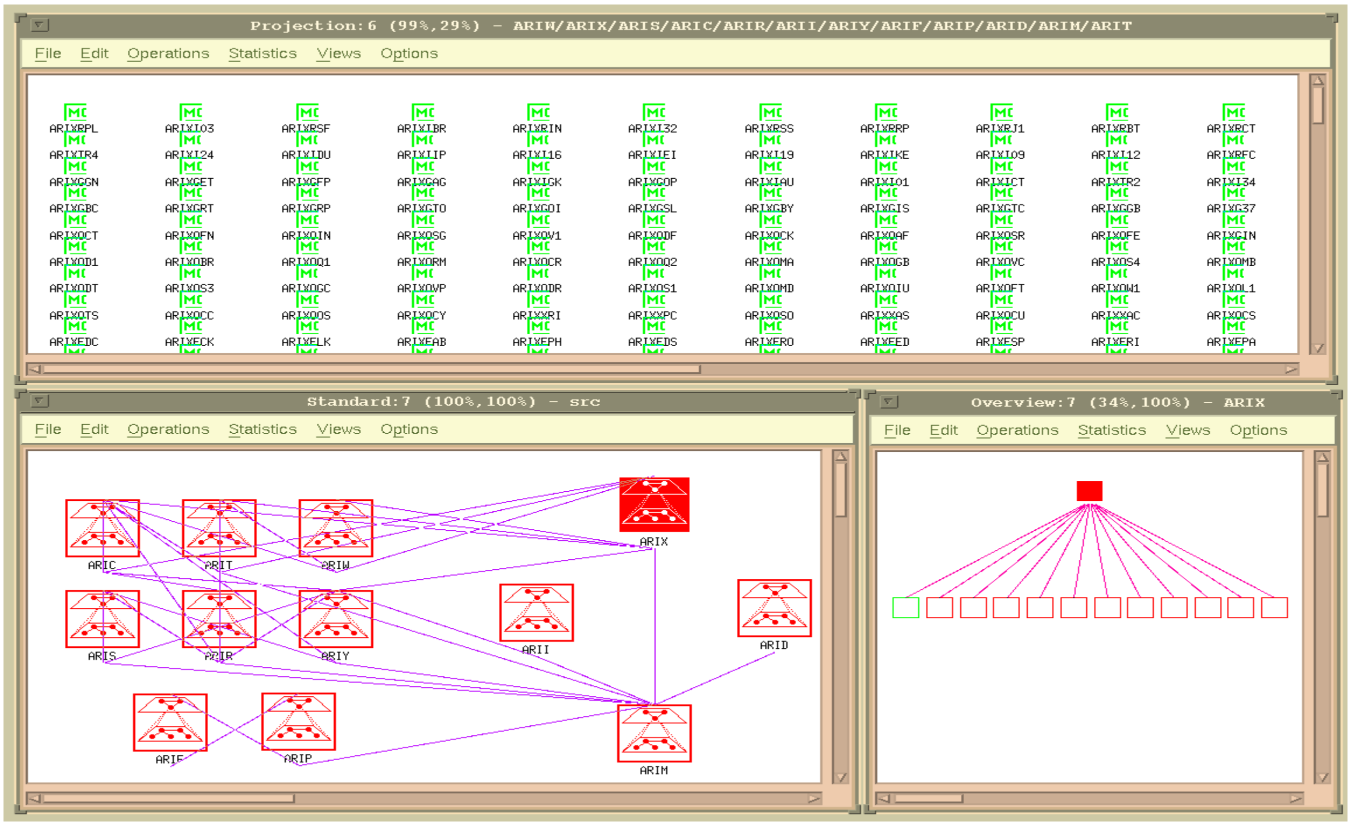
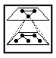


Figure 3: A view of SQL/DS and the ARIX subsystem

window shows a portion of all the low-level modules of SQL/DS.[8] Each module is represented by a MOD icon, such as  ARIXI32 for the ARIXI32 module. The lower-left window shows a high-level "horizontal" view of the major subsystems, each depicted by a SYS icon, such as  ARIX for the ARIX component. The arcs in this window represent composite dependencies between the major subsystems. The hierarchical subsystem structure within the highlighted ARIX component in this window is expanded, filtered, and presented in the bottom-right overview window.

[8]For clarity, the arcs have been filtered from the view.

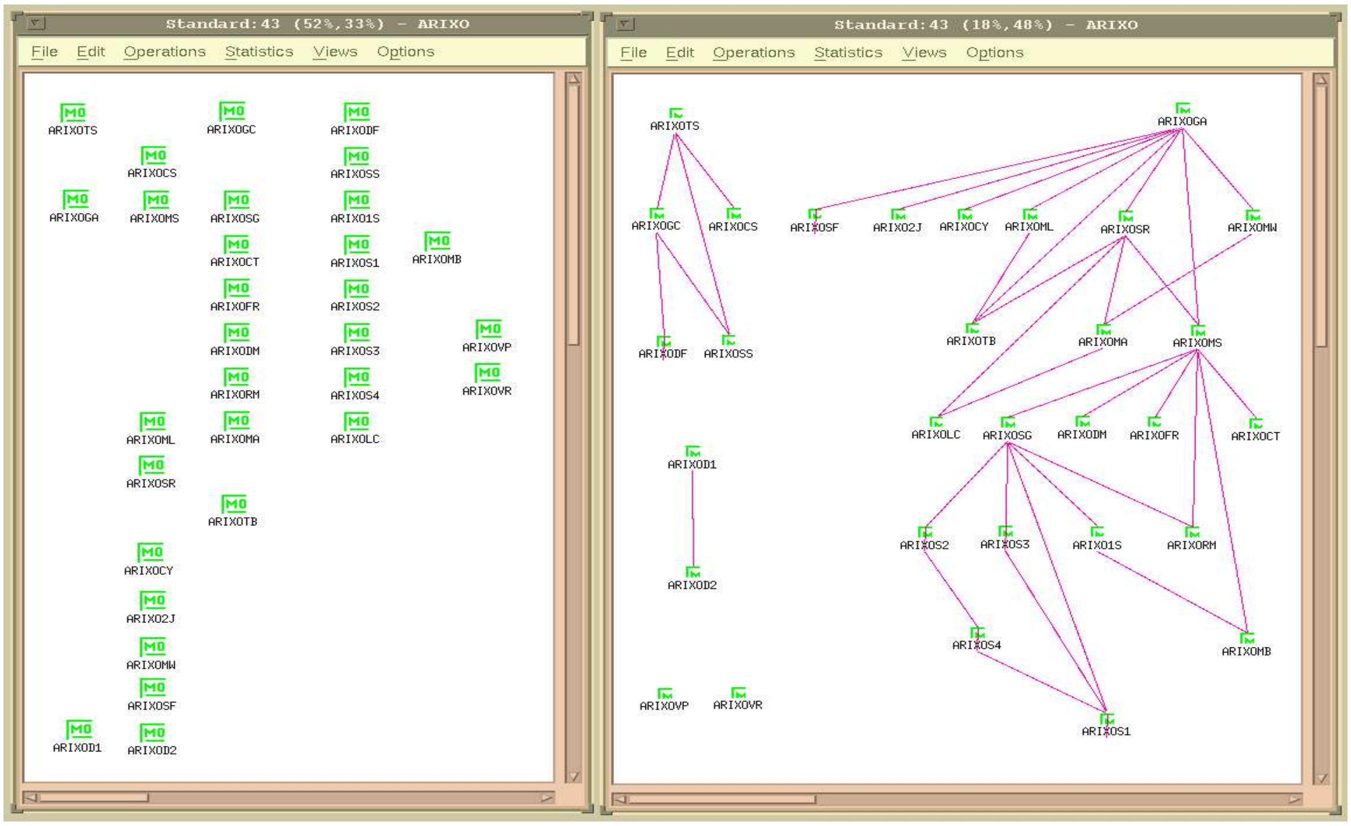


Figure 4: Two views of the path selection optimizer subsystem

Experiment III: A subsystem in detail

Decomposing the ARIX relational data subsystem was the focus of our third experiment. This subsystem is roughly one million lines and consists of nine physical subsystems. With the help of an SQL/DS developer, we decomposed it into four logical subsystems: (1) runtime access generator, (2) optimizer pass one and two, (3) path selection optimizer, and (4) executive, interpreter, and authorization. Four distinct development teams are in charge of each of these subsystems.

We then further decomposed the path selection optimizer subsystem ARIXO. The developer in charge of this subsystem had created her own diagrams of its structure, based on product development logbooks and the mental model she had formed from her maintenance experience. A view was easily created using Rigi to portray this mental model. More importantly, an alternative view was created, based on the actual structure as reflected by the current source code. These two views of ARIXO are shown in Figure 4. The window on the left contains the maintainer's view and the window on the right the newly constructed view.

The maintainer's view reflects a left to right ordering of the major product components according to functional design layers. The new view reflects a top-down control flow based on actual information extracted from the source code. This second view presented a somewhat different perspective and in fact conflicted with existing architectural documentation in some respects. However, because it was constructed using automatically extracted information, it was a more accurate representation of the subsystem's "operational" architecture. The maintainer was able to form a more accurate mental model based on a combination of information gained from maintenance experience and information extracted from the code. The two perspectives can be unified in a single Rigi view.

Developer feedback

While our prepared views did not uncover the exact mental model of each developer, the audience readily recognized the presented structures. There were two main reasons for this. First, these developers knew their subsystems intimately. Second, and more importantly, the views represented the right level of abstraction. Most satisfying for us was when the developers used their individual knowledge to design additional views to reflect their personal mental models more closely. This was usually done by emphasizing components of particular interest and filtering irrelevant information (from the point of view of each individual developer).

Could we have achieved the same result without tool support? Yes and no. It is true that system description diagrams have been in use for a long time. However, recreating and updating them for legacy systems of this magnitude would be ponderous, if not impossible. Moreover, our semi-automatic approach enables several such documents to be created and maintained simultaneously. These system-level documents are always up-to-date since they are based on the underlying source code.

5 Summary

There will always be old software that needs to be understood. It is critical for the software industry to deal effectively with the problems of software evolution and the understanding of legacy software systems. Tools and methodologies that effectively aid software engineers in understanding large and complex software systems can have a significant impact.

Legacy software systems require a different approach to software documentation than has traditionally been used. As an aid to program understanding for large, evolving software systems, structural redocumentation through reverse engineering plays a key role. Through this process, one can produce accurate in-the-large design documents describing the architecture of the software system's current state—not that of the original system before numerous maintenance changes were made.

The Rigi environment focuses on the architectural aspects of the subject system under analysis. The environment supports a method for identifying, building, and documenting layered subsystem hierarchies. Critical to its usability is the ability to store and retrieve views—snapshots of reverse engineering states. The views are used to transfer information about the abstractions to the software engineers.

Script-based decompositions capture domain-specific knowledge and can be generated quickly. For example, it took two days to semi-automatically create a decomposition using Rigi, but only minutes to automatically produce one via a prepared script. Either method would be much faster and more efficient of the analyst's time and effort than a manual process of reading program listings and consulting volumes of, perhaps out-of-date, system documentation.

Our analysis of the source code of SQL/DS has proven to be very valuable to the developers and our own research. It has shown that our methodology scales up to the million-lines-of-code range and that the structural redocumentation produced during system analysis is an aid in understanding such legacy software.

Finally, we have shown that we can provide businesses with valuable information about the architecture of their legacy systems. Experienced software developers who are familiar with the system can design additional views in Rigi according to multiple, alternative perspectives. This produces charts and graphs that can ease software maintenance, thereby save the company time and money, as well as extending the useful life of their legacy system.

Future work

Further analysis of SQL/DS is underway. One of our research associates is using the Software Refinery to parse PL/AS and export more fine-grained relationships among procedures and variables. Most of our work until now has been focused on exploring control dependencies. However, the architecture of SQL/DS is based on global data structures manipulated by many different software modules. While the the developer looks at code that is over 90% control logic, the compiler sees over 90% as data structure declarations, placed in shared '%INCLUDE' files. It will be fruitful to investigate this aspect of the system.

We are currently designing and developing a more ambitious reverse engineering environment based on seven years of experience gained with Rigi. This new environment involves three universities and IBM as an industrial partner; collaboration is the main theme. McGill University is extending the structural pattern matching capabilities of the Rigi system to support syntactic, semantic, functional, and behavioral search patterns. The University of Toronto is building a more flexible repository for storing software artifacts, pattern matching rules, and software engineering knowledge. The University of Victoria is making the Rigi system more extensible by enhancing the scripting language, improving the user interface, and providing a method for modelling the domain of discourse.

Acknowledgments

We would like to thank Michael Whitney for his work on the Rigi system. Without his efforts, much of our analysis would not have been possible. The constructive comments made by the anonymous referees were appreciated. We would also like the thank the SQL/DS group members at IBM for their participation in the case study, and CAS for inviting us to take part in this research. In particular, we would like to thank Jacob Slonim, Director of CAS, for his guidance and support.

References

[1] T. A. Corbi. Program understanding: Challenge for the 1990's. Technical Report RC 14370, IBM T.J. Watson Research Center, January 1989.

[2] 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.

[3] 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*, 5(4):181–204, December 1993.

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

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

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