Seven Issues for the Next Generation of Program Understanding Systems[†]

Marijana Tomic
Centre for Advanced Studies
IBM Software Solutions
844 Don Mills Rd.
North York, ON M3C 1V7
mtomic@vnet.ibm.com

Scott R. Tilley
Department of Computer Science
University of Victoria
P.O. Box 3055, MS7209
Victoria, BC V8W 3P6
stilley@csr.uvic.ca

Abstract

This paper highlights seven of the major issues that must be addressed by the next generation of program understanding systems. These issues are based on some lessons learned using different reverse engineering tools and techniques, both separately and together as part of an integrated toolset, in an on-going program understanding project. The project's focus is on reengineering real-life software systems on the order of $O(10^6)$ lines of code.

Keywords: program understanding, reverse engineering, software systems

1 Introduction

Since 1991 the CAS[1] Program Understanding Project has been using reverse engineering technology to support SQL/DS[2] product development and maintenance. SQL/DS is an excellent example of a legacy software system: it is a large, successful, and mature product. It is also under continuous pressure to run on new environments and provide increased functionality. It is representative of systems that that are inherently difficult to understand and maintain because of their size, complexity, and evolution history. To address the challenges posed by such systems, reverse engineering for program understanding is often used [1].

Reverse engineering is concerned with the analysis of existing systems in order to make them more understandable for maintenance and evolution purposes. The importance of reverse engineering has grown tremendously as corporations face mounting maintenance and reengineering costs for large legacy software systems. Such systems are indispensable in day-to-day operations and have been built decades ago.

Program understanding is the process of developing mental models of a software system's architecture, purpose, and behavior. There have been numerous research efforts to develop tools that provide assistance during the understanding process, but it is clear that no one approach or tool is sufficient by itself. The software engineer can best be served through a collection of tools that complement each other in functionality.

The objectives of the program understanding project include the development of an integrated environment for reverse engineering which offers tools for subsystem identification and design recovery. In particular, the project address issues in the areas of software analysis technology, algorithms to extract system abstractions, integration technology applicable to CASE, user-interface technology to model, browse, navigate, and search large collections of software artifacts interactively, and reverse engineering process models. A common repository plays an integral part in the integration of the diverse tools into the environment.

The project involves a team from CAS and five research groups from University of Toronto, University of Victoria, McGill University, University of Michigan, and National Research Council. All groups are working cooperatively on complementary reverse engineer-

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

[2]SQL/DS (Structured Query Language/Data System) is a multimillion-line relational database system.

ing approaches, using the source code of SQL/DS as a common reference legacy system.

The next section discusses the primary reverse engineering tools and techniques used in the project. Section 3 compares and contrasts these tools. Section 4 lists the seven issues we feel the next generation of program understanding systems must address. These issues are based on our experiences in using the tools (both separately and together). Section 5 summarizes the paper.

2 Reverse engineering tools and techniques

The two main tools used in the project are Rigi [2] and the Software Refinery [3]. Rigi is a realization of the PHSE (*P*rogrammable *H*yper*S*tructure *E*ditor) [4], a domain-independent meta reverse engineering environment framework. It is instantiated for a particular application domain by specializing its *conceptual model*, by extending its *core functionality*, and by providing an application-specific *personality*.

The Software Refinery is a commercially available toolkit from Reasoning Systems. It is composed of three parts: Dialect (used to produce parsers and printers for programming languages), InterVista (used tools to create the user interfaces), and Refine (a high-level multi-paradigm programming language). It makes extensive use of freely available software such as the X Window System, bison (a yacc-like program), and emacs.

The reverse engineering techniques used in the project can be broadly categorized into three areas: defect filtering, structural redocumentation, and pattern-matching analyses. The CAS group is concerned with defect filtering [5]: improving the quality of the SQL/DS base code and maintenance process through application-specific analysis. They where using the Software Refinery to parse the source code of SQL/DS into a form suitable for analysis. Some of the quality-related defects searched for include: programming language violations (overloaded keywords, poor data typing), implementation domain errors (data coupling, addressability), and application domain errors (coding standards, business rules).

The University of Victoria team focuses on structural redocumentation [6]: the production of *in-the-large* documents describing high level subsystem architecture. Reconstructing the design of existing software is especially important for legacy systems such as SQL/DS, since there are great differences in understanding software systems of 1,000 lines of code versus those of 1,000,000 lines. The redocumentation is performed using Rigi.

NRC, the University of Michigan, and McGill University are working on pattern-matching approaches at various levels: textual, syntactic, and semantic. The purpose of pattern matching is to identify artifacts and relations of a subject system. In the project, three different levels of abstraction are used: textual, syntactic, and semantic. The textual pattern matching performed by NRC attempts to locate source code repetitions in the subject system, as might result from common cut/copy/edit/paste operations. The syntactic pattern matching performed by the University of Michigan attempts to locate specific source code fragments using a special-purpose query language called SCRUPLE. The semantic pattern matching performed by McGill University attempts to locate code fragments using a plan matching algorithm.

3 Tool comparison

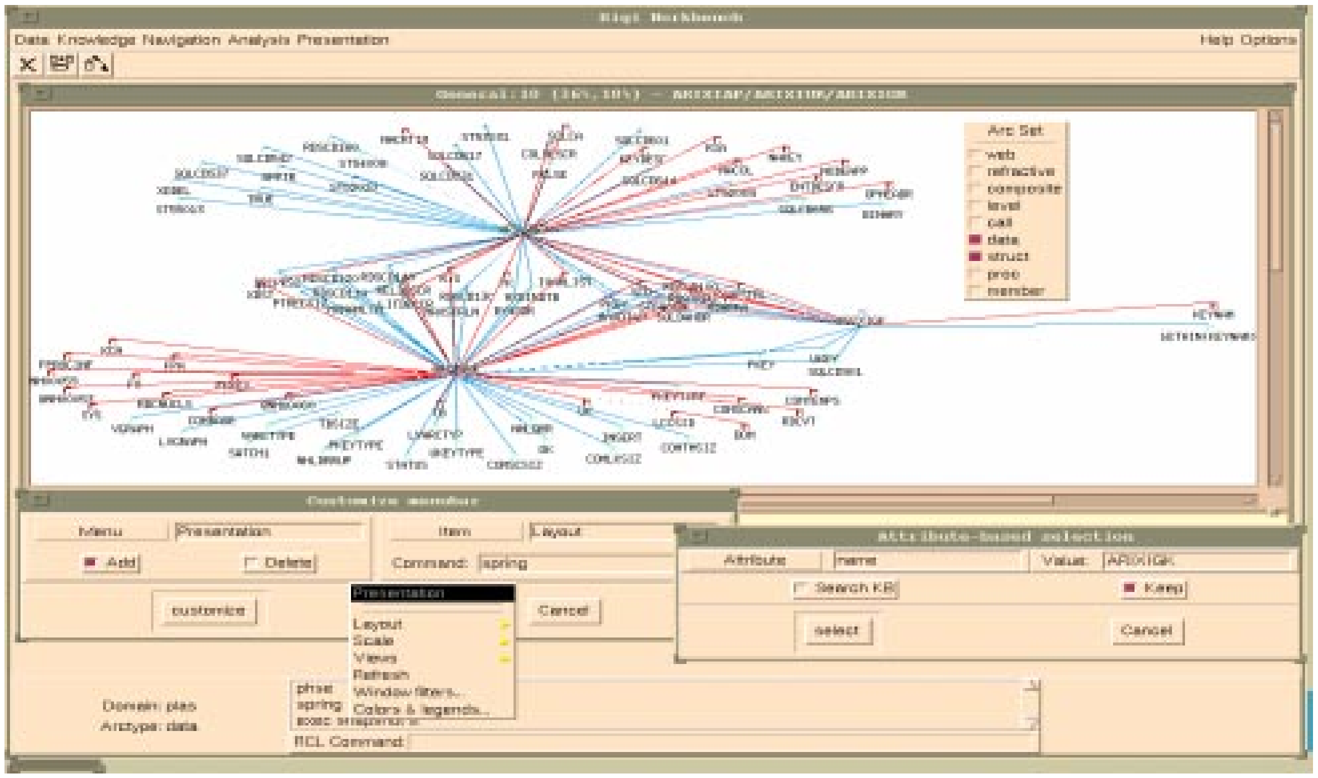
Rigi and Refine share the common goal of aiding system understanding through reverse engineering. Rigi provides a completely programmable interface that allows for easy modeling, tailoring, and extensibility. However, Refine provides more native capabilities for fine-grained analysis of a program. Working cooperatively, those two tools can greatly aid the comprehension process; at this point we have just scratched the surface of the benefits of the integration.

Rigi's flexibility is illustrated in Figure 1. The figure depicts a spring layout of a logical SQL/DS subsystem (the routines associated with adding a foreign key to the database, within the same physical component). A spring layout is chosen to show the data being shared by the modules. The data used for the visualization was produced through the Software Refinery and loaded into an instance of Rigi programmed with a PL/AS domain model.

Refine's analysis capabilities are illustrated in Figure 2. The figure shows a defect filter running on a C program (a C/C++ compiler). The defect filter searches for coding standards violations in the source code.

4 Seven issues

Using Rigi and the Software Refinery in the project has proven very valuable. It has also raised a number of issues that should be addressed in subsequent phases of the project. Many of them are germane to



Menu: Presentation
Item: Layout
Command: spring
Attribute: name
Value: ARIXIGK
Domain: plas
Arctype: data
RCL Command:

Figure 1: Data coupling visualization

the reverse engineering community as a whole. Seven of the most important are:

1. *Integration with business process reengineering*

   Software reverse engineering should be considered in the global picture together with business process reengineering. It is important to be aware that a reengineering project can fail if performed in isolation, and not merged in with the larger motif. Even with this knowledge, the question remains: How can we merge software reverse engineering with business process reengineering? There could be numerous benefits from putting these two activities together. Which tools and techniques are similar in the two domains? Which are different?

2. *Targeting non-traditional users*

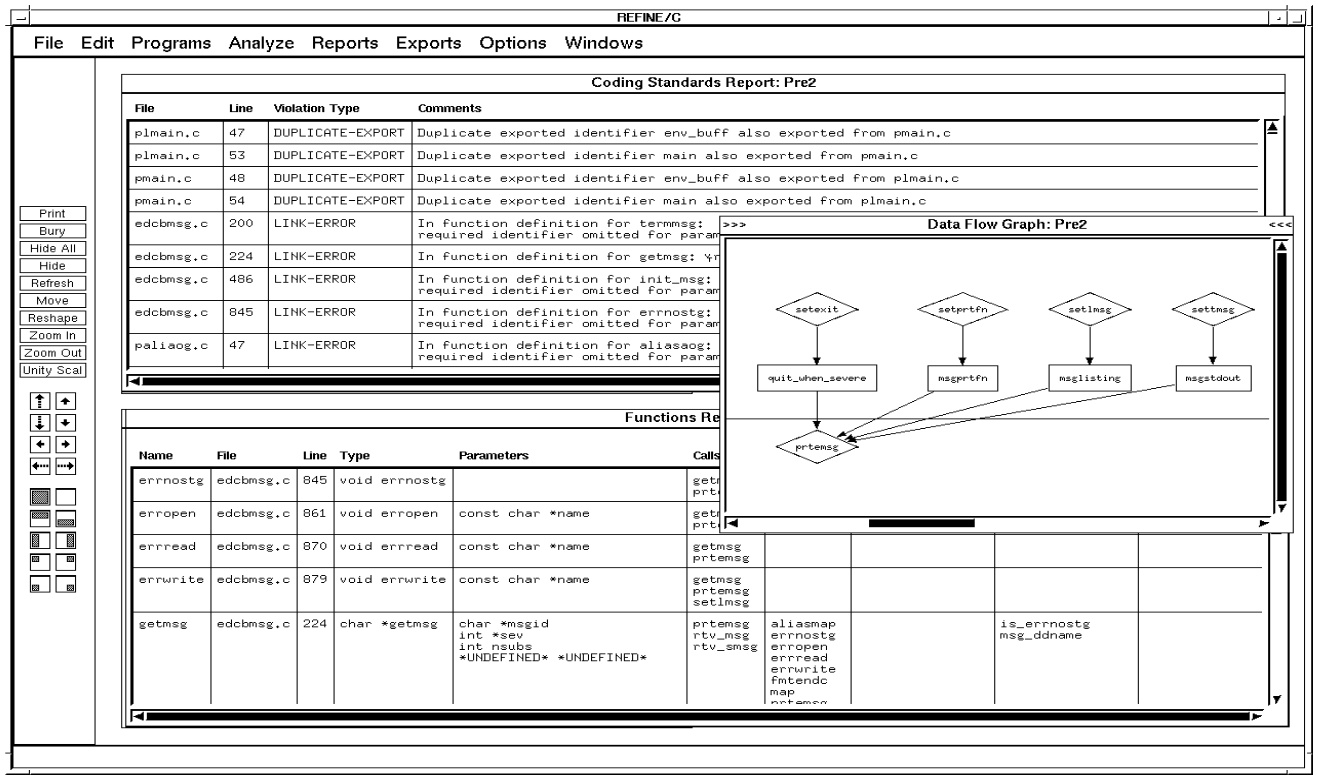
   The information produced through reverse engineering should be presented in a way that is understandable not just to programmers, but to non-programmers as well. During the project we learned that for adequate feedback, we needed not only developers and maintainers, but also testers, technical writers, and managers; developers are only one aspect of a large software project. Non-programmers can help in extracting business rules, but only if the reverse engineering results are presented in an acceptable and understandable form. How might these non-traditional clients use the information produced through reverse engineering?

3. *Reuse experience to guide tool selection*

   When starting a new reengineering task, how does one choose which reverse engineering tools and techniques are appropriate for a particular project? Can you reuse experience from previous projects? If the expertise from previous efforts could be reused in the new application domain, significant start-up benefits could be realized.

4. *Supporting heterogeneous toolsets*

   Researchers usually attempt to incorporate a wide variety of functions into their tool. This often results in a tool that is clumsy and too complicated to use. In our project, we combined different reverse engineering tools into one environment.



Coding Standards Report: Pre2

File	Line	Violation Type	Comments
plmain.c	47	DUPLICATE-EXPORT	Duplicate exported identifier env_buff also exported from pmain.c
plmain.c	53	DUPLICATE-EXPORT	Duplicate exported identifier main also exported from pmain.c
pmain.c	48	DUPLICATE-EXPORT	Duplicate exported identifier env_buff also exported from plmain.c
pmain.c	54	DUPLICATE-EXPORT	Duplicate exported identifier main also exported from plmain.c
edcbmsg.c	200	LINK-ERROR	In function definition for termmsg: required identifier omitted for param
edcbmsg.c	224	LINK-ERROR	In function definition for getmsg: ¥r
edcbmsg.c	486	LINK-ERROR	In function definition for init_msg: required identifier omitted for param
edcbmsg.c	845	LINK-ERROR	In function definition for errnostg: required identifier omitted for param
paliaog.c	47	LINK-ERROR	In function definition for aliasaog: required identifier omitted for param

Functions Re

Name	File	Line	Type	Parameters	Calls				
errnostg	edcbmsg.c	845	void errnostg		getr prt				
erropen	edcbmsg.c	861	void erropen	const char *name	getr prt				
errread	edcbmsg.c	870	void errread	const char *name	getmsg prtemsg				
errwrite	edcbmsg.c	879	void errwrite	const char *name	getmsg prtemsg setlmsg				
getmsg	edcbmsg.c	224	char *getmsg	char *msgid int *sev int nsubs *UNDEFINED* *UNDEFINED*	prtemsg rtv_msg rtv_smsg	aliasmap errnostg erropen errread errwrite fmtendc map		is_errnostg msg_ddname	

Figure 2: Coding standard violations

That way, each tool is defined for one specific reverse engineering technique. In our case, Rigi is more focused on the overall structure of the system's components, and Refine is more focused on the internal structure of individual source files.

We learned some things about integration in the project, but many questions remain. What is the appropriate integration strategy to use when combining different reverse engineering tools into one's environment? Is it worthwhile to commit resources to building separate setups for each tool? Is it possible to build an environment which can accommodate every tool? Some combination of tools?

5. *Support exploratory reverse engineering*

   The goal is not always clear during reverse engineering. Even the overall goal of "program understanding' is ill-defined. For this reason, the environment should support exploratory reverse engineering. This would permit "what if" scenarios to be evaluated, before actual resources are committed to a large reengineering effort.

6. *Support incrementality*

   Incremental reverse engineering is needed to support large software systems, where it is unlikely one would attempt to understand its entire structure all at once. It is also needed to support the caching of domain knowledge created during reverse engineering, so that the next session can "pick up" where the previous one left off. Incrementality also means that when the underlying source code changes, the results of reverse engineering should not be completely invalidated.

7. *Incorporate cost/benefits analysis into the process*

   Can we answer the question: Should we continue to maintain the system, or redesign it? If we decided to redesign our software, should we reengineer it, or write a version from scratch? What are the cost/benefits to reverse engineering? These questions are critical in real-world reengineering exercises, yet few systems allow for this type of information to be accurately incorporated into the reverse engineering process.

5 Summary

The program understanding project's investigation into different reverse engineering tools and techniques has proven very enlightening. Using the power of any single tool produces limited benefits when applied to large legacy systems. Integrating complementary tools into a unified environment seems to be the most favorable approach—but it is a non-trivial activity. The seven issues described in Section 4 are just some of the many important items that the next generation of program understanding systems must address.

TrademarksSQL/DS is a trademarks of International Business Machines Corporation.

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

References

[1] E. Buss, R. D. Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.

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

[3] P. Newcomb and L. Markosian. Automating the modularization of large cobol programs: Application of an enabling technology for reengineering. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 222–230. IEEE Computer Society Press (Order Number 3780-02), May 1993.

[4] S. R. Tilley. *Domain-Retargetable Reverse Engineering.* PhD thesis, Department of Computer Science, University of Victoria, 1994 (in progress).

[5] J. Troster, J. Henshaw, and E. Buss. Filtering for quality. In the Proceedings of *CASCON '93,* (Toronto, Ontario; October 25-28, 1993), pages 429–449, October 1993.

[6] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. To appear in *IEEE Software*, January 1995.