Investigating Reverse Engineering Technologies: The CAS Program Understanding Project

E. Buss R. De Mori M. Gentleman J. Henshaw H. Johnson K. Kontogiannis
E. Merlo H. Müller J. Mylopoulos S. Paul A. Prakash M. Stanley
S. Tilley J. Troster K. Wong

Abstract

Corporations face mounting maintenance and re-engineering costs for large legacy systems. Evolving over several years, these systems embody substantial corporate knowledge, including requirements, design decisions, and business rules. Such knowledge is difficult to recover after many years of operation, evolution, and personnel change. To address this problem, software engineers are spending an ever-growing amount of effort on program understanding and reverse engineering technologies. This article describes the scope and results of an on-going research project on program understanding undertaken by the IBM Software Solutions Toronto Laboratory Centre for Advanced Studies (CAS). The project involves, in addition to a team from CAS, five research groups working cooperatively on complementary reverse engineering approaches. All groups are using the source code of SQL/DS (a multi-million line relational database system) as the reference legacy system. The article also discusses the approach adopted to integrate the various toolsets under a single reverse engineering environment.

Keywords: Legacy software systems, program understanding, software reuse, reverse engineering, software metrics, software quality.

1 Introduction

Software maintenance is not an option. Developers today inherit a huge legacy of existing software. These systems are inherently difficult to understand and maintain because of their size and complexity as well as their evolution history. The average Fortune 100 company maintains 35 million lines of code and adds an additional ten percent each year just in enhancements, updates, and other maintenance. As a result of maintenance alone, software inventories will double in size every seven years. Since these systems cannot easily be replaced without reliving their entire history, managing long-term software evolution is critical. It has been estimated that fifty to ninety percent of evolution work is devoted to *program understanding* [1]. Hence, easing the understanding process can have significant economic savings.

One of the most promising approaches to the problem of program understanding for software evolution is *reverse engineering*. Using reverse engineering technologies has been proposed to help refurbish and maintain software systems. To facilitate the understanding process, the subject software 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.

This paper describes the use of several complementary reverse engineering technologies applied to a real-world software system: SQL/DS. The goal was to aid the maintainers of SQL/DS in improving product quality by enhancing their understanding of the three-million lines of source code. Section 2 provides background on the genesis of the program understanding project and its focus on the SQL/DS product. Subsequent sections detail the individual research programs. Section 3 describes defect filtering as a way of improving quality by minimizing design errors. The abundance of defect filtering information needs to be summarized by effective visualization and documentation tools. Section 4 discusses a system to reconstruct and present high-level structural documentation for software understanding. A comprehensive approach to reverse engineering requires many different techniques. Section 5 outlines three techniques that analyze source code at textual, syntactic, and semantic levels. The convergence of the separate research prototypes into an integrated reverse engineering environment is reported in Section 6. Finally, Section 7 summarizes the important lessons learned in this endeavor.

2 Background

Faced with demanding and ambitious quality-related objectives, the SQL/DS product group offered the opportunity to use their product as a candidate system for analysis. Faced with this challenge, the program understanding project was established in 1990 with goals to *investigate the use of reverse engineering technologies on real-world (SQL/DS) problems, and to utilize program understanding technologies to improve the quality of the SQL/DS product and the productivity of the SQL/DS software organization.*

The CAS philosophy encourages complementary research teams to work on the same problem, using a common base product for analysis. There is little work in program understanding that involves large, real-world systems with multiple teams of researchers experimenting on a common target [2]. Networking opportunities ease the exchange of research ideas. Moreover, colleagues can explore related solutions in different disciplines. This strategy introduces new techniques to help tackle the problems in industry and, as well, strengthens academic systems to deal with complex, industrial software systems. In addition, universities can move their research from academia into industry at an accelerated rate.

Six different research groups participated in and contributed to the CAS program understanding project: the IBM Software Solutions Toronto Laboratory Centre for Advanced Studies, the National Research Council of Canada (NRC), McGill University, the University of Michigan, the University of Toronto, and the University of Victoria. All groups focused on the source code of SQL/DS as the reference legacy software system.

2.1 The reference system: SQL/DS

SQL/DS (Structured Query Language/Data System) is a large relational database management system that has evolved since 1976. It was based on a research prototype and has undergone numerous revisions since its first release in 1982. Originally written in PL/I to run on VM, SQL/DS is now over 3,000,000 lines of PL/AS code and runs on VM and VSE. PL/AS is a proprietary IBM systems programming language that is PL/I-like and allows embedded System/370 assembler. Because PL/AS is a proprietary language, commercial off-the-shelf analysis tools are unsuitable. Simultaneous support of SQL/DS for multiple releases on multiple operating systems requires multi-path code maintenance, increasing the difficulty for its maintainers.

SQL/DS consists of about 1,300 compilation units, roughly split into three large systems (and several smaller ones). Because of its complex evolution and large size, 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 to keep current with the source code, too much to read and digest, and not enough one can trust. 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.

The top-level goals of the program understanding project were guided by the maintenance concerns of the SQL/DS developers. Two of the most important were code correctness and performance enhancement. Specific concerns included: detecting uninitialized data, pointer errors, and memory leaks; detecting data type mismatches; finding incomplete uses of record fields; finding similar code fragments; localizing algorithmic plans; recognizing inefficient or high-complexity code; and predicting the impact of change.

2.2 Program understanding through reverse engineering

Programmers use programming knowledge, domain knowledge, and comprehension strategies when trying to understand a program. For example, one might extract syntactic knowledge from the source code and rely on programming knowledge to form semantic abstractions. Brooks's work on the theory of domain bridging [3] describes the programming process as one of constructing mappings from a problem domain to an implementation domain, possibly through multiple levels. Program understanding then involves reconstructing part or all of these mappings. Moreover, the programming process is a cognitive one involving the assembly of programming plans—implementation techniques that realize goals in another domain. Thus, program understanding also tries to pattern match between a set of known plans (or mental models) and the source code of the subject software.

For large legacy systems, the manual matching of such plans is laborious and difficult. One way of augmenting the program understanding process is through computer-aided reverse engineering. Although there are many forms of reverse engineering, the common goal is to extract information from existing software systems. This knowledge can then be used to improve subsequent development, ease maintenance and re-engineering, and aid project management [4].

The reverse engineering process identifies the system's current components, discovers their dependencies, and generates abstractions to manage complexity [5]. It involves two distinct phases [6]: (1) the identification of the system's current components and their dependencies; and (2) the discovery of system abstractions and design information. During this process, the source code is not altered, although additional information about the system is generated. In contrast, the process of re-engineering typically consists of a reverse engineering phase, followed by a forward engineering or re-implementation phase that alters the subject system's source code. Definitions of related concepts may be found in [7].

The discovery phase is a highly interactive and cognitive activity. The analyst may build up hierarchical subsystem components that embody software engineering principles such as *low coupling* and *high cohesion* [8]. Discovery may also include the reconstruction of design and requirements specifications (often referred to as the "domain model") and the correlation of this model to the code.

2.3 Program understanding research

Many research groups have focused their efforts on the development of tools and techniques for program understanding. The major research issues involve the need for formalisms to represent program behavior and visualize program execution, and focus on features such as control flows, global variables, data structures, and resource exchanges. At a higher semantic level, it may focus on behavioral features such as memory usage, uninitialized variables, value ranges, and algorithmic plans. Each of these points of investigation must be addressed differently.

There are many commercial reverse engineering and re-engineering tools available; catalogs such as [9, 10] describe several hundred such packages. Most commercial systems focus on source-code analysis and simple code restructuring, and use the most common form of reverse engineering: information abstraction via program analysis. Research in reverse engineering consists of many diverse approaches, including: formal transformations [11], meaning-preserving restructuring [12], plan recognition [13], function abstraction [14], information abstraction [15], maverick identification [16], graph queries [17], and reuse-oriented methods [18].

The CAS program understanding project is guided, in part, by the need to produce results directly applicable to the SQL/DS product team. Hence, the work of most research groups is oriented towards analysis. However, no single analysis approach is sufficient by itself. Specifically, the IBM group is concerned with defect filtering: improving the quality of the SQL/DS base code and maintenance process through application-specific analysis. The University of Victoria is focused on structural redocumentation: the production of "in-the-large" documents describing high-level subsystem architecture. Three other groups (NRC, the University of Michigan, and McGill University) are working on pattern-matching approaches at various levels: textual, syntactic, and semantic.

One goal of this project is to integrate the results of the complementary (but sometimes overlapping) research efforts to produce a more comprehensive reverse engineering toolset; this integration process is described more fully in Section 6. The following sections describe the program understanding project's main research results on defect filtering, structural redocumentation, and pattern matching.

3 Defect filtering

The IBM team, led by Buss and Henshaw, perform *defect filtering* [19] using the commercial Software Refinery product (REFINE) [20] to parse the source code of SQL/DS into a form suitable for analysis. This work applies the experience of domain experts to create REFINE "rules" to find certain families of defects in the subject software. These defects include programming language violations (overloaded keywords, poor data typing), implementation domain errors (data coupling, addressability), and application domain errors (coding standards, business rules).

Their initial work resulted in several prototype toolkits, each of which focuses on detecting specific errors in the reference system. Troster performed a *design-quality metrics analysis* (D-QMA) study of SQL/DS [21]. These measurements guided the creation of a more flexible defect filtering approach, in which the reverse engineering toolkit automatically applies defect filters against the SQL/DS source code. *Filtering for quality* (FQ) proved be to a fruitful approach to improving the quality of the reference system [22]. This section describes the evolution of the defect filtering process: from the investigation and construction of a reverse engineering toolkit for PL/AS, the construction of prototype analysis systems, the measurement of specific design-quality metrics of SQL/DS, and, finally, to filtering for quality.

3.1 Building a reverse engineering toolkit

Most application problem domains have unique and specialized characteristics; therefore, their expectations and requirements for a reverse engineering tool vary. Thus, reverse engineering toolkits must be extensible and versatile. It is unlikely that a turn-key reverse engineering package will suffice for most users. This is especially true for analyzing systems of a proprietary nature such as SQL/DS. Unless one knows exactly what one wants to accomplish, one should place a premium on toolkit flexibility. Because of these considerations, the Software Refinery was chosen as the basis upon which to build a PL/AS reverse engineering toolkit for the defect filtering process.

⇒ Insert sidebar "The Software Refinery" here. ⇐

The PL/AS reverse engineering toolkit was used to aid qualitative and quantitative improvement of the SQL/DS base code and maintenance process. The key to this improvement is analysis. The Software Refinery was used to convert the SQL/DS source code into a more tractable form. Considerable time was spent creating a parser and a domain model for PL/AS. This was a difficult process: there was no formal grammar available, the context-sensitive nature of the language made parsing a challenge, and the embedded System/370 assembler code further complicated matters. A lexical analyzer was first built to recognize multiple symbols for the same keyword, to skip the embedded assembler and PL/AS listing format directives, and to produce input acceptable to the parsing engine.

Initial experiments produced numerous parsing errors, due to incorrect (or inappropriate) use of some of PL/AS's "features." Although it is never easy to change legacy source code, it was sometimes easier to repair the source code than to augment the parser to handle the offending syntax. This process uncovered several errors in the reference system's source code. Such errors were usually incorrect uses of language constructs not identified by the PL/AS compiler.

This early experience with the PL/AS reverse engineering toolkit confirmed that large scale legacy software systems written in a proprietary context-sensitive language can be put into a form suitable for sophisticated analysis and transformation. The toolkit can be (and has been) adapted by other IBM developers to similar programming languages and evolves as their implementation rules change.

Once SQL/DS's source code was put in this tractable form, it was time to revisit the "customer" (the SQL/DS maintainers) to determine how best to utilize this technology for them. The answer came back crystal clear: "Help us remove defects from our code." The challenge was how to do it effectively. The solution was to apply the power of the prototype environment to analyzing the reference system. Since rules can be written to identify places in the software where violations of coding standards, performance guidelines, and implementation or product requirements exist, the environment can be used to detect defects semi-automatically.

3.2 Experiences with the PL/AS reverse engineering toolkit prototypes

The construction of the prototype reverse engineering toolkit, and the transformation of the base code into a more tractable form, made analysis of the reference system possible. The analysis was strongly biased towards defect detection, due in part to the SQL/DS product group's quality-related objectives. The analysis focused on implementation language irregularities and weaknesses, functional defects, software metrics, and unused code. A specific instance of the prototype toolkit was constructed for each analysis realm.

The areas of interest can be classified into two orthogonal pairs of analysis domains: analysis-in-the-small versus analysis-in-the-large, and implementation domain versus problem domain. The analysis-in-the-small is concerned with analysis of code fragments (usually procedures) as a closed domain, while analysis-in-the-large is concerned with system-wide impact. Analysis-in-the-large tends to be more difficult to perform with manual methods, and therefore more benefits may be realized through selective automation.

Implementation domain analysis is concerned with environmental issues such as language, compiler, operating system and hardware. This analysis can usually be readily shared with others who have a similiar environment. Conversely, the problem domain analysis is concerned with artifacts of the problem such as business rules, algorithms, or coding standards. They cannot be easily shared.

The prototypes for SQL/DS where specifically built to demonstrate the capability for analysis in all of these domains. Some of the prototypes are documented in [23]. The results from these prototype toolkits were encouraging. The experiments demonstrated the feasibility of defect detection in legacy software systems. The next step in the use of such reverse engineering technologies was formalizing and generalizing the process of using defect filters on the reference system.

3.3 Design-quality metrics analyses

While maintenance goals continue to focus on generally improved performance and functionality objectives, an emerging emphasis has been placed on IBM's product quality. With developers mounting quality improvement goals, a paradigm shift beyond simply "being more careful" is needed. Judicious use of software quality metrics is one way of obtaining insight into the development process to improve it. To confirm the applicability of such metrics to IBM products, Troster initiated the design-quality metrics analysis (D-QMA) project.

The purpose of assessing design-quality metrics [24] is to examine the design process by examining the end product (source code), to predict a product's quality, and to improve the design process by either continuous increments or quantum leaps. To justify the use of D-QMA for IBM products, the experiment had to:

- relate software defects to design metrics;
- identify error-prone and high-risk modules;
- predict the defect density of a product at various stages;
- improve the cost estimation of changes to existing products; and
- provide guidelines and insights for software designers.

The experiment assessed the high-level and module-level metrics of SQL/DS and related them to the product's defect history.

Inter-module metrics for module-level design measure inter-module coupling and cohesion, data flow between modules, and so on. These "black-box" measures require no knowledge of the inner workings of the module. Intra-module design metrics include measures of control flow, data flow, and logic within a module. These "clear-box" measures require knowledge of the inner working of the module. Both the inter-module and intra-module versions of structural complexity, data complexity, and system complexity [25] were measured. Other module-level measurements are shown in Figure 1.

⇒ Insert sidebar "The Conformance Hierarchy" here. ⇐

The experiment applied the reverse engineering toolkit developed by Buss and Henshaw (described in Section 3.1) to extract the metrics from the reference system. Defect data was gathered from the defect database running on VM/CMS; the data was then correlated using the SAS statistical package running on OS/2. For SQL/DS V3R3, about nine hours of machine time (on a RISC System/6000 M550) were required to analyze all 1,303 PL/AS modules. This time does not include a previous 40–50 person-hours required to prepare a persistent database for the SQL/DS source code.

The unique characteristics of the SQL/DS reference system lead to several problems in assessing the metrics. One of the most important is the nonhomogeneity of the product. SQL/DS consists of functional components that are quite different. There are preprocessors, communications software, a relational database engine, utilities, and so on. Each component displays different metric characteristics.

Upon analyzing the results, it was found that defects caused by design errors accounted for 43 percent of the total product defects. The next largest class of defects was coding errors. The probability of injecting a defect when maintaining a module increased as the percentage of changes to the module decreased. The greatest probability of introducing a defect occured when the smallest change was made. This counter-intuitive result makes more sense when it is realized that when small changes are made, maintainers typically do not take the time to fully understand the entire module.

Another result is that maintainers have an increased probability of injecting a defect as the complexity of the module increases—up to a threshold. As the module complexity increases beyond this threshold, the probability of injecting an error dramatically decreases. This suggests that the maintainer recognizes the module is complex and "tries harder," or that as modules become more complex, maintainers avoid changing them altogether.

The past three releases of SQL/DS have shown new modules to have low complexity, with older ones growing in complexity. As this complexity increases, merely "working harder" to ensure code quality will not be enough. It is becoming increasingly difficult to make small changes to the more mature modules: a classic example of the "brittleness" suffered by aging software systems. The D-QMA work is continuing with analyzing other IBM products written in PL/AS, PL/MI, C, and C++.

3.4 Applying defect filters to improve quality

An increased focus on quality has forced many organizations to re-evaluate their software development processes. Software process improvement concerns improved methods for managing risk, increasing productivity, and lowering cost: all key factors in increased software quality. The meaning of the term "quality," however, is often subject to debate and may depend on one's perspective. The definition of quality we use is *quality is the absence of defects.* This somewhat traditional definition relates quality to "fitness for use" and ties software's quality to conformance with respect to function, implementation environment, and so on. The traditional quality measurement, measuring defects, is one that measures the artifacts created by the software development process.

By extending the meaning of what constitutes a defect, one can expand the definition of quality. For example, the recognition of defects caused by coding standard violations means that quality is no longer bound to purely functional characteristics; quality attributes can be extended to include indirect features of the software development process.

Further extension to the quality framework may include *assertions* that must be adhered to; assertion non-conformance can be treated as a defect. Like functional defects, these assertions can address issues at a variety of levels of abstraction. Our definition of software quality is then extended to include robustness, portability, improved maintenance, hidden defect removal, design objectives, and so on; "fitness for use" is superseded by "fitness for use and maintenance." Figure 2 illustrates a conformance hierarchy. This hierarchy begins at the base with immediate implementation considerations and climbs upward to deal with broader conceptual characteristics. Beginning with "what is wrong" (defects), it moves up to "what is right" (assertions). By tightening the definition of "correctness," one can build higher quality software.

In order to ensure that a software product is fit for use, developers carefully review the software, checking for possible defects and verifying that all known product-related assertions are met. This is commonly known as the "software inspection process." An approach to automating the inspection process incorporates the reverse engineering technologies discussed in Section 3.1. This filtering process, termed *filtering for quality*, involves the formalization of corrective actions using a language model and database of rules to inspect source code for defects. The rules codify defects in previous releases of the product. This is a context-driven approach that extends the more traditional language-syntax-driven methods used in tools such as *lint*.

There are many benefits of automation to the filtering for quality (FQ) process. A greater number of defects can be searched simultaneously. Moreover, the codified rules can be generalized and restated to eliminate entire classes of errors. Actions are expressed in a canonical rule-based form; therefore, they are more precise, less subject to misinterpretations, and more amenable to automation. Because the knowledge required to prevent defects is maintained as a rule-base, the knowledge instilled in each action remains even after original development team members have left. This recording of informal "corporate knowledge" is very important to long-term success. Finally, actions can be more easily exchanged with other groups using the same or similar action rule-bases. This sharing of such defect filters means that development groups can directly profit from each other's experience.

Application domain knowledge can be very beneficial in the development of defect filters, largely because the capability to enforce application domain-specific rules has been unavailable to date. Whether one wants to enforce design assertions about a software product or to identify exceptions to the generally held principles around which a software product has evolved, one should pay attention to the filter's domain. The problem domain consists of business rules and other aspects of the problem or application—independent of the way they are implemented. The implementation domain consists of the implementation programming language and support environment.

3.5 Summary

Meeting ambitious quality improvement goals such as "100 times quality improvement" requires an improved definition of defects and an improved software development process. Defect filtering by automating portions of the inspection process can reap great rewards. A tractable software representation is key to this analysis.

It is easier to use defect filtering than it is to build the tool that implements it. Nevertheless, it is critical that the analysis results be accessible to developers in a timely fashion to make an identifiable impact on their work. The success of moving new technology into the workplace depends crucially on the acceptance of the system by its users. Its introduction must have minimal negative impact on existing software processes if it is to be accepted by developers. Issues such as platform conflict should not be underestimated. The prototype tools discussed in Sections 3.1 and 3.2 have been partially integrated into the mainstream SQL/DS maintenance process.

Measurable results come from measurable problems. Defect filtering produces directly quantifiable benefits in software quality and can be used as a stepping stone to other program understanding technology. For example, presentation and documentation tools are needed to make sense of the monumental amount of information generated by defect filtering. This critical need is one focus of the environment described in the following section.

4 Structural redocumentation

Reconstructing the design of existing software is especially important for legacy systems such as SQL/DS. Program documentation has always played an important role in program understanding. There are, however, great differences in documentation needs for software systems of 1,000 lines versus those of 1,000,000 lines. Typical software documentation is *in-the-small*, describing the program in terms of isolated algorithms and data structures. Moreover, the documentation is often scattered and on different media. The maintainers have to resort to browsing the source code and piecing disparate information together to form higher-level structural models. This process is always arduous; creating the necessary documents from multiple perspectives is often impossible. Yet it is exactly this sort of *in-the-large* documentation that is needed to expose the overall architecture of large software systems.

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 [26]. One class of techniques of reconstructing structural models is reverse engineering.

Using reverse engineering approaches to reconstruct the architecture aspects of software can be termed *structural redocumentation*. The University of Victoria's work is centered around Rigi [27]: an environment for understanding evolving software systems. Output from this environment can also serve as input to conceptual modelling, design recovery, and project management processes. Rigi consists of three major components: a tailorable parsing system that supports procedural programming languages such as C, COBOL, and PL/AS; a distributed, multi-user repository to store the extracted information; and an interactive, window-oriented graph editor to manipulate structural representations.

4.1 Scalability

Effective approaches to program understanding must be applicable to huge, multi-million line software systems. Such scale and complexity necessitates fundamentally different approaches to repository technology than is used in other domains. For example, not all software artifacts need to be stored in the repository; it may be perfectly acceptable to ignore certain details for program understanding tasks. Coarser-grained artifacts can be extracted, partial systems can be incrementally investigated, and irrelevant parts can be ignored to obtain manageable repositories. Program representation, search strategies, and human-computer interfaces that work on systems "in-the-small" often do not scale up. For very large systems, the information accumulated during program understanding is staggering. To gain useful knowledge, one must effectively summarize and abstract the information. In a sense, a key to program understanding is deciding what information is material and what is immaterial: knowing what to look for—and what to ignore [28].

4.2 Redocumentation strategy

There are tradeoffs in program understanding environments between what can be automated and what should (or must) be left to humans. Structural redocumentation in Rigi is initially automatic and involves parsing the source code of the subject system and storing the extracted artifacts in the repository. This produces a flat resource-flow graph of the software. This phase is followed by a semi-automatic one that exploits human pattern recognition skills and features language-independent subsystem composition techniques to manage the complexity. This approach relies very much on the experience of the software engineer using the system. This partnership is synergistic as the analyst also learns and discovers interesting relationships by interactively exploring software systems using Rigi.

Subsystem composition is a recursive process whereby building blocks such as data types, procedures, and subsystems are grouped into composite subsystems. This builds multiple, layered hierarchies for higher-level abstractions [29]. The criteria for composition depend on the purpose, audience, and domain. For program understanding purposes, the process is guided by dividing the resource-flow graph using established modularity principles such as low coupling and strong cohesion. Exact interfaces and modularity/encapsulation quality measures can be used to evaluate the generated software hierarchies.

Subsystem composition is supported by a program representation known as the $(k, 2)$-partite graph [29]. These graphs are layered or stratified into strict levels so that arcs do not skip levels. The levels represent the composition of subsystems. This structuring mechanism was originally devised for managing the complexity of hypertext webs and multiple hierarchies.

4.3 Multiple dynamic views

Visual representations enhance the human ability to recognize patterns. Using the graph editor, diagrams of software structures such as call graphs, module interconnection graphs, and inclusion dependencies can be automatically produced. The effective capability to analyze these structures is necessary for program understanding. Responsiveness is very important. For presenting the large graphs that arise from a complex system like SQL/DS, the response time may degrade even on powerful workstations. The Rigi user interface is designed to allow users, if necessary, to batch sequences of operations and to specify when windows are updated. Thus, for small graphs, updates are immediate for visually pleasing feedback; for large graphs, the user has full control of the redrawing.

Rigi presents structural documentation using a collection of *views.* A view is a group of visual and textual frames that contain, for example, resource flow graphs, overviews, projections, exact interfaces, and annotations. Because views are dynamic and ultimately based on the underlying source code, they remain up-to-date. Collected views can be used to retrieve previous reverse engineering states.

Dramatic improvements in program understanding are possible using semi-automatic techniques that exploit application-specific domain knowledge. Since the user is in control, the subsystem composition process can depend on diverse criteria, such as tax laws, business policies, personnel assignments, requirements, or other semantic information. These alternate and orthogonal decompositions may co-exist under the structural representation supported by Rigi. These decompositions provide many possible perspectives for later review. In effect, multiple, logical representations of the software's architecture can be created, manipulated, and saved.

4.4 Domain-retargetability

Because program understanding involves many diverse aspects, applications, and domains, it is necessary that the approach be very flexible. Many reverse engineering tools provide only a fixed palette of extraction, selection, filtering, arrangement, and documentation techniques. The Rigi approach uses a scripting language that allows analysts to customize, combine, and automate these activities in unforeseen ways. Efforts are proceeding to also make the user interface fully user-customizable. This approach permits analysts to tailor the environment to better suit their needs, providing a smooth transition between automatic and semi-automatic reverse engineering. The goal of *domain-retargetability*, having a single environment sufficiently flexible so as to be applicable and equally effective in multiple domains, is achieved through this customization.

To make the Rigi system programmable and extensible, the user interface and editor engine were decoupled to make room for an intermediate scripting layer based on the embeddable Tcl and Tk libraries [30]. This layer allows each event of importance to the user (for example, key stroke, mouse motion, button click, menu selection) to be tied to a scripted, user-defined command. Many previously tedious and repetitive activities can now be automated. Moreover, this layer allows an analyst to complement the built-in operations with external, possibly application-specific, algorithms for graph layout, complexity measures, pattern matching, slicing, and clustering. For example, the Rigi system has been applied to various selected domains: project management [31], personalized hypertext [32], and redocumenting legacy software systems.

4.5 Redocumenting SQL/DS

The analysis of SQL/DS using Rigi has shown that the subsystem composition method and graph visualizing editor scale up to the multi-million lines of code range. The results of the analysis were prepared as a set of structural views and presented to the development teams. Informal information and knowledge provided by existing documentation and expert developers are rich sources of data that should be leveraged whenever possible. By considering SQL/DS-specific knowledge such as naming conventions and existing physical modularizations, team members easily recognized the constructed views. Domain-dependent scripts were devised to help automate the decomposition of SQL/DS into its constituent components.

For example, the relation data subsystem of SQL/DS was analyzed in some depth. The developer in charge of the path-selection optimizer had her own mental model of its structure, based on development logbooks and experience. This model was recreated using Rigi's structural redocumentation facilities. An alternate view was also created, based on the actual structure as reflected by the source code. This second view constitutes another reverse-engineering perspective and was a valuable reference against which the first view was compared.

4.6 Summary

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 pertinent information about the abstractions to the software engineers.

Rigi supports human- and script-guided structural pattern recognition, but does not provide built-in operations to perform analysis such as textual, syntactic, and semantic pattern matching. Such operations are necessary for complete program understanding. However, the scripting layer does support access to external tools that cover these areas of analysis, allowing Rigi to function as the cornerstone of a comprehensive reverse engineering environment. These required areas are addressed by the prototypes described in the following section.

5 Pattern matching

One of the most important reverse engineering processes is the analysis of a subject system to identify components and relations. Recognizing such relations is a complex problem solving activity that begins with the detection of cues in the source and continues by building hypotheses from these cues. One approach to detecting these cues is to start by looking at program segments which are similar to each other.

Program understanding techniques may consider source code in increasingly abstract forms, including: raw text, preprocessed text, lexical tokens, syntax trees, annotated abstract syntax trees with symbol tables, and control/data flow graphs. The more abstract forms entail additional syntactic and semantic analysis that corresponds more to the meaning and behavior of the code and less to its form and structure. Different levels of analysis are necessary for different users and different program understanding purposes. For example, preprocessed text loses a considerable amount of information about manifest constants, inline functions, and file inclusions. Three research groups affiliated with the program understanding project focus on textual, syntactic, and semantic pattern-matching approaches.

5.1 Textual analysis

Anything that is big and worth understanding has some internal structure; finding and understanding that internal structure is the key to understanding the whole. In particular, large source codes have lots of internal structure as a result of their evolution. The NRC research focuses on techniques that consider the source code in raw or preprocessed textual forms, dealing with more of the incidental implementation artifacts than other methods. The work by Johnson [33] at NRC concerns the identification of exact repetitions of text in huge source codes. One goal is to relax the constraint of exact matches to approximate matches, while preserving the ability to handle huge source texts. The general approach is to automatically analyze the code and produce information that can be queried and reported.

For some understanding purposes, less analysis is better; syntactic and semantic analysis can actually destroy information content in the code, such as formatting, identifier choices, whitespace, and commentary. Evidence to identify instances of textual cut-and-paste is lost as a result of syntactic analysis. Tools for syntactic and semantic analysis are often more language and environmentally dependent; slight changes in these aspects can make the tools inapplicable. For example, C versions of such tools may be useless on PL/AS code.

More specifically, these techniques discover the location and structure of long matching substrings in the source text. Such redundancies arise out of typical editing operations during maintenance. Measures of repetition are a useful basis for building practical program understanding tools. There are several possibilities for redundancy-based analysis, including: determining the effects of cut-and-paste, discovering the effects of preprocessing, measuring changes between versions, and understanding where factoring and abstraction mechanisms might be lacking.

The NRC approach works by fingerprinting an appropriate subset of substrings in the source text. A fingerprint is a shorter form of the original substring and leads to more efficient comparisons and faster redundancy searches. Identical substrings will have identical fingerprints. However, the converse is not necessarily true. Differing substrings may also have the same fingerprint, but the chance of this occurring can be made extremely unlikely. A file of substring fingerprints and locations provides the information needed to extract source-code redundancies.

There are several issues to be addressed: discovering efficient algorithms for computing fingerprints, determining the appropriate set of substrings, and devising postprocessing techniques to make the generated fingerprint file more useful. Karp and Rabin [34] have proposed an algorithm based on the properties of residue arithmetic by which fingerprints can be incrementally computed during a single scan. A modified version of this algorithm is used. Appropriate substrings, called snips, are selected to exploit line boundary information; the selection parameters are generally based on the desired number of lines and maximum and minimum numbers of characters. Even then, an adjustable culling strategy is used to reduce the sheer number of snips that would still be fingerprinted. Since snips can overlap and contain the same substring many times, this culling strategy represents substrings by only certain snips. Particularly important postprocessing includes merging consecutive snips that match in all occurrences, thus producing longest matching substrings. Extensions of this can identify long substrings that match except for short insertions or deletions.

An experimental prototype has been built and applied to the source code of the SQL/DS reference legacy system. This led to a number of observations. The expansion of inclusions via preprocessing introduces textual redundancy. These redundancies were easily detected by the prototype. When the prototype was applied to a small part of the source code (60 files, 51,655 lines, 2,983,573 characters), considering matches of at least 20 lines, there appeared to be numerous cut-and-paste occurrences—about 727 copied lines in 13 files. Processing of the entire 300 megabyte source text ran successfully in under two hours on an IBM RISC System/6000 M550. To perform a more complete and useful analysis of SQL/DS, research is now focused on approximate matching techniques and better postprocessing and presentation tools. Textual analysis complements other analysis tools by providing information that these tools miss.

5.2 Syntactic analysis

The effort by Paul and Prakash at the University of Michigan focuses on the design and development of powerful *source code search systems* that software engineers (or tools designed by them) can use to specify and detect "interesting" code fragments. Searching for code is an extremely common activity in reverse engineering, because maintainers must first find the relevant code before they can correct, enhance, or re-engineer it. Software engineers usually look for code that fits certain patterns. Those patterns that are somehow common and stereotypical are known as clichés. Patterns can be structural or behavioral, depending on whether one is searching for code that has a specified syntactic structure, or looking for code components that share specific data-flow, control-flow, or dynamic (program execution-related) relationships.

5.2.1 Deficiencies with current approaches

Despite the critical nature of the task, good source code search systems do not exist. General string-searching tools such as *grep*, *sed*, and *awk* can handle only trivial queries in the context of source code. Based on regular expressions, these tools do not exploit the rich syntactic structure of the programming language. Source code contains numerous syntactic, structural, and spatial relationships that are not fully captured by the entity-relation-attribute model of a relational database either.

For example, systems such as CIA [35] and PUNS [36] only handle simple statistical and cross-reference queries. Graph-based models represent source code in a graph where nodes are software components (such as procedures, data types, and modules), and arcs capture dependencies (such as resource flows). The SCAN system [37] uses a graph-based model that is an attributed abstract syntax representation. This model does capture the structural information necessary; however, it does not capture the strong typing associated with programming-language objects. Moreover, it fails to support type lattices, an essential requirement to ensure substitutability between constructs that share a supertype-subtype relationship. Object-based models, such as the one used by REFINE, adequately capture the structural and relational information in source code. However, the focus in REFINE has not been on the design of efficient source code search primitives.

5.2.2 SCRUPLE

The University of Michigan group has developed the SCRUPLE source code search system (Source Code Retrieval Using Pattern LanguagEs) [38]. SCRUPLE is based on a pattern-based query language that can be used to specify complex structural patterns of code not expressible using other existing systems. The pattern language allows users flexibility regarding the degree of precision to which a code structure is specified. For example, maintainers trying to locate a matrix multiplication routine may specify only a control structure containing three nested loops, omitting details of contents of the loops, whereas those trying to locate all the exact copies of a certain piece of code may use the code piece itself as their specification.

The SCRUPLE pattern language is an extension of the source code programming language. The extensions include a set of symbols that can be used as substitutes for syntactic entities in the programming language, such as statements, declarations, expressions, functions, loops, and variables. When a pattern is written using one or more of these symbols, it plays the role of an abstract template which can potentially match different code fragments.

The SCRUPLE pattern matching engine searches the source code for code fragments that match the specified patterns. It proceeds by converting the program source code into an abstract syntax tree (AST), converting the pattern into a special finite state machine called the *code pattern automaton* (CPA), and then simulating the behavior of the CPA on the AST using a CPA interpreter. A matching code fragment is detected when the CPA enters a final state. Experience with the SCRUPLE system shows that a code pattern automaton is an efficient mechanism for structural pattern matching on source code.

5.2.3 Source code algebra

SCRUPLE is an effective pattern-based query system. However, current source code query systems, including SCRUPLE, succeed in handling only subsets of the wide range of queries possible on source code, trading generality and expressive power for ease of implementation and practicality. To address the problem, Paul and Prakash have designed a *source code algebra* (SCA) [39] as the formal framework on top of which a variety of high-level query languages can be implemented. In principle, these query languages can be graphical, pattern-based, relational, or flow-oriented.

The modeling of program source code as an algebra has four important consequences for reverse engineering. First, the algebraic data model provides a unified framework for modeling structural as well as flow information. Second, query languages built using the algebra will have formal semantics. Third, the algebra itself serves as low-level applicative query language. Finally, source code queries expressed as algebra expressions can be optimized using algebraic transformation rules and heuristics.

Source code is modeled as a *generalized order-sorted algebra* [40], where the sorts are the program objects with operators defined on them. The choice of sorts and operators directly affects the modeling and querying power of the SCA. Essentially, SCA is an algebra of objects, sets, and sequences. It can be thought of as an analogue of relational algebra, which serves as an elegant and useful theoretical basis for relational query languages. A prototype implementation of the SCA query processor is underway. The next step is to test it using suites of representative queries that arise in reverse engineering. The final goal is to automatically generate source code query systems for specific programming languages from high-level specifications of the languages (that is, their syntax and data model). The core of the query system will be language independent. This tool generation technique is similar to *yacc*, a parser generator.

5.3 Semantic analysis

The McGill research [41] involves four subgoals. First, program representations are needed to capture both the structural and semantic aspects of software. Second, comparison algorithms are needed to find similar code fragments. Third, pattern matching algorithms are needed to find instances of programming plans (or intents) in the source code. Fourth, a software process definition is needed to direct program understanding and design recovery analyses.

5.3.1 Program representation

A suitable program representation is critical for plan recognition because the representation must encapsulate relevant program features that identify plan instances, while simultaneously discarding implementation variations. There are several representation methods discussed in the literature, including data and control flow graphs, Prolog rules, and λ-calculus. McGill's representation scheme is an object-oriented annotated AST.

A grammar and a domain model for the language of the subject system is constructed using the Software Refinery. The domain model defines an object hierarchy for the AST nodes and the grammar is used to construct a parser that builds the AST. Some tree annotations are produced by the parser; others are produced by running analysis routines on the tree. Annotations produced by the parser include source code line numbers, include file names, and links between identifier references and corresponding variable and datatype definitions. Annotations produced by analysis routines include variables used and updated, functions called, variable scope information, I/O operations, and complexity/quality metrics. Annotations stored in the AST may be used by other analysis routines.

5.3.2 Programming plans

More generally, comparison methods are needed to help recognize instances of programming plans (abstracted code fragments). There are several other pattern matching techniques besides similarity measures. GRASP [42] compares the attributed data flow subgraphs of code fragments and algorithmic plans and uses control dependencies as additional constraints. PROUST [43, 44] compares the syntax tree of a program with suites of tree templates representing the plans. A plan-instance match is recognized if a code fragment conforms to a template, and certain constraints and subgoals are satisfied. In CPU [45], comparisons are performed by applying a unification algorithm on code fragments and programming plans represented by lambda calculus expressions.

Textual- and lexical-matching techniques encounter problems when code fragments contain irrelevant statements or when plans are delocalized. Moreover, program behavior is not considered. Graph-based formalisms capture data and control flow, but transformations on these graphs are often expensive and pattern matching algorithms can have high time complexity. This poses a major problem when analyzing huge source codes.

In addition, plan instance recognition must contend with problems such as syntactic variations, interleaved plans, and implementation differences. One major problem is the failure of certain methods to produce any results if precise recognition is not achieved. The McGill group is focusing on plan localization algorithms that can handle partial plans. Human assistance is favored over a completely automatic approach based on a fixed plan library.

Plans should stand for application-level concepts and not simply be abstracted code fragments. Concepts might be high-level descriptions of occurrences or based on more familiar properties such as assertions, data dependencies, or control dependencies. Within McGill's approach, plans are user-defined portions of the annotated AST. A pattern-matching and localization algorithm is used to find all code fragments that are similar to the plan. The plan, together with the similar fragments, forms a "similarity" class. The object-oriented environment gives flexibility in the matching process because some implementation variations are encoded in the class hierarchy. For example, *while*, *for*, and *repeat-until* statements are subclasses of the *loop-statement* class. The object hierarchy that classifies program structure and data types is defined within a language-specific domain model.

5.3.3 Similarity analysis

One focus in pattern matching is on identifying similar code fragments. Existing source code is often reused within a system via "cut-and-paste" text operations (cf. Section 5.1). This practice saves development time, but leads to problems during maintenance because of the increased code size and the need to propagate changes to every modified copy. Detection of cloned code fragments must be done using heuristics since the decision whether two arbitrary programs perform the same function is undecidable. These heuristics are based on the observation that the clones are not arbitrary and will often carry identifiable characteristics (features) of the original fragment.

The McGill approach to identifying clones uses various complexity metrics. Each code fragment is tagged by a signature tuple of its complexity values. This transformational technique simplifies software structures by converting them to simpler canonical forms. In this framework, the basic assumption is that, if code fragments c_1 and c_2 are similar under a set of features measured by metric M, then their metric values $M(c_1)$ and $M(c_2)$ for these features will also be close. Five metrics have been chosen that exhibit a relatively low correlation coefficient, and are sensitive to a number of different program features that may characterize a code fragment. They are:

1. the number of functions called from a software component (i.e., fan-out);
2. the ratio of I/O variables to the fan-out;
3. McCabe's cyclomatic complexity [46];
4. Albrecht's Function Point quality metric [47]; and
5. Henry-Kafura's information flow quality metric [48].

Similarity is gauged by a distance measure on the tuples. The distances currently used are based on two measures: (1) the Euclidean distance defined in the 5-dimensional space of the above measures; and (2) on clustering thresholds defined on each individual measure axis (and on intersections between clusters in different measure axes).

Another analysis is to determine closely related software components, according to criteria such as shared references to data, data bindings, and complexity metrics. Grouping software components by such varied criteria provides the analyst with different views of the program. The data binding criteria tracks uses of variables in one component that are defined within another (a kind of interprocedural resource flow). The implementation of these analyses uses the REFINE product.

5.3.4 Goal-driven program understanding

Another design recovery strategy that has been explored by the McGill group is a variation of the GQM [49] model: the *goal, question, analysis, action* model [50]. A number of available options are compared, and the one that best matches a given objective is selected. The choice is based on experience and formal knowledge.

This process can be used to find instances of programming plans. The comparison process is iterative, goal-driven, and affected by the purpose of the analysis and the results of previous work. A moving frontier [51] divides recognized plans and original program material. Subgoals are set around fragments that have been recognized with high confidence. The analysis continues outward seeking the existence of other parts of the plan in the code. Interleaved plans can be handled by allowing gaps and partial plan recognition.

5.4 Summary

Research prototypes have been built for performing textual, syntactic, and semantic analysis of the SQL/DS system. Both the McGill and Michigan tools can process PL/AS code, but have also been applied to C code. The NRC tool found numerous cut-and-paste redundancies in the SQL/DS code. Research is continuing on improving these tools. The NRC group is focusing on better visualization techniques. Michigan is investigating better program representations and pattern matching engines. McGill is exploring techniques for plan recognition and similarity distances between source code features.

A number of common themes have arisen from this research. Domain-specific knowledge is critical in easing the interpretation of large software systems. Program representations for efficient queries are essential. Many kinds of analyses are needed in a comprehensive reverse engineering approach. An extensible environment is needed to consolidate these diverse approaches into a unified framework. An architecture for a multi-faceted reverse engineering environment to addresses these requirements is presented in the next section.

6 Steps toward integration

The first phase of the program understanding project produced practical results and usable prototypes for program understanding. In particular, the defect filtering system developed by Buss, Henshaw, and Troster is used daily by several development groups, including SQL/DS and DB2. The second phase of the program understanding project is focusing on the integration of selected prototype tools into a comprehensive environment for program understanding.

The prototype tools individually developed by each research group offer complementary functionalities and differ in the methods they use to represent software descriptions, to implement such descriptions in terms of physical data structures, and in the mechanisms they deploy to interact with other tools. Ideally, the output of one prototype tool should be usable as input by another. For example, some of the many dependencies generated by the defect filtering system might be explored and summarized using the Rigi graph editor. However, the defect detection system uses the REFINE object-oriented repository, and the Rigi system uses the GRAS graph-based repository [52]. Integrating the representations employed by REFINE and Rigi is a non-trivial problem.

With such integration in mind, a new phase of the project was launched early in 1993. Some of the key requirements for the integration were:

- smooth data, control, and presentation integration among components of the environment;
- extensible data model and interfaces to support new tools and user-defined objects, dependencies, and functions;
- domain-specific, semantic pattern matching to complement the facilities developed during the first phase of the project;
- the representation and support of processes and methodologies for reverse engineering; and
- robust program representations, user interfaces, and algorithms, capable of handling large collections of software artifacts.

The rest of this section describes the steps that have been taken to provide data integration through a common repository for a variety of tools for program understanding. In addition, the section describes the subsystem of the environment responsible for control integration.

6.1 Repository schema

The University of Toronto contribution focuses on the development of an information schema and the implementation of a repository to support program understanding. The repository needs to store both the extracted information gathered during the discovery phase as well as the abstractions generated during the identification phase of reverse engineering. The information stored must be readily understandable, persistent, shareable, and reusable. Moreover, the repository must have a common and consistent conceptual schema that is a superset of the sub-schemas used by the program understanding tools, including those for REFINE and Rigi. The repository should also provide simple repository operations to select and update information pertinent to a specific tool. The schema is expected to change, and therefore it must support dynamic evolution.

The schema is under development and is being implemented in three phases. The first phase, which has already been implemented, captures the information currently required by REFINE and Rigi. This information consists of programming language constructs from C, which are discovered through parsing, as well as user-defined and tool-generated objects. For example, the concept of a Rigi subsystem is captured in a class called "Module." This concept, however, is not supported by REFINE and therefore does not exist in the REFINE sub-schema. Similarly, the programming language construct of an arithmetic expression is captured in the REFINE sub-schema using the class "Expression." This construct has no Rigi sub-schema representation since Rigi does not currently deal with intraprocedural details. As an example of a shared concept, the notion of a function is common to both tools and is captured in the shared class "Function." Each tool has a slightly different view of this class, seeing only the common portions and the information pertinent to itself. The second phase classifies the patterns used and captures the analysis results generated from each tool. The third phase will record other information relevant to reverse engineering, such as designs, system requirements, domain modelling, and process information. The remainder of this section describes the schema developed for the first phase.

The information model adopted for the repository schema is Telos, originally developed at the University of Toronto [53]. Features of Telos include: an object-oriented framework that supports generalization, classification and attribution, a meta-modelling facility, and a novel treatment of attributes including multiple inheritance of attributes and attribute classes. Telos was selected over other data models (for example, REFINE, ObjectStore, C++-based) because it is more expressive with respect to attributes and is extensible through its treatment of metaclasses. To support persistent storage for the repository, however, we adopted the commercial object-oriented database ObjectStore.

As illustrated in Figure 3, the schema consists of three tiers. The top level (MetaClass Level) exploits meta modelling facilities to define: (1) the types of attribute values that the repository supports; and (2) useful groupings of attributes to distinguish information that is pertinent to each of the individual tools. For example, "RigiClass" is used to capture all data that pertains to Rigi at the level below and thus it defines the kinds of attribute classes that the lower level Rigi classes can have. The use of this level eases schema evolution and provides an important filtering and factoring mechanism. The middle level (Class Level) defines the repository schema, classifying in terms of the metaclasses and attributes defined at the top level. For instance, "RigiObject," "RigiElement," "RigiProgrammingObject," and "Function" (grouped in the grey shaded area in Figure 3, all use the attribute metaclasses defined in "RigiClass" above, to capture information about particular Rigi concepts. As the example suggests, a repository object is categorized based on the pertinent tool and whether it is automatically extracted or produced through analysis. The bottom level (Token Level) stores the software artifacts needed by the individual tools. Figure 3 shows three function objects: "listinit," "mylistprint," and "listfirst" corresponding to the actual function definitions. These are created when Rigi parses the target source code.

6.2 Environment architecture

A generic architecture is one important step toward the goal of creating an integrated reverse engineering environment. The main integration requirements of this environment concern data, control, and presentation. Data integration is essential to ensure that the individual tools can communicate with each other; this is accomplished through a common schema. Control integration enhances interoperability and data integrity among the tools. This is realized through a data server built using a customizable and extensible message server called the Telos Message Bus (TMB), as shown in Figure 4. This message server allows all tools to communicate both with the repository and with each other, using the common schema as interlingua. These messages form the basis for all communication in the system. The server has been implemented on top of existing public domain software bus technology [54] using a layered approach that provides both mechanisms and policies specifically tailored to a reverse engineering environment. For example, the bottom layer provides mechanisms by which a particular tool can receive messages of interest to it. The policy layer is built on top of the mechanism layer to determine if and how a particular tool responds to those messages.

This architecture has been implemented. The motivation for the layered and modular approach to the schema and architecture came from an earlier experience by the University of Toronto group in another proejct. This project faced similar requirements, such as the need for a common repository to help integrate disparate tools. Additional experience with this architecture for reverse engineering purposes is currently ongoing.

7 Summary

There will always be old software that needs to be understood. It is critical for the information technology sector in general, and software industry in particular, 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.

Buss and Henshaw at IBM built several prototype toolkits in REFINE, each focused on detecting specific errors in SQL/DS. Troster, also at IBM, developed a flexible approach that applies defect filters against the source code to improve the quality. Defect filtering produces measurable results in software quality.

Müller's research group at the University of Victoria developed the Rigi system, which focuses on the high-level architecture of the subject system under analysis. Views of multiple, layered hierarchies are used to present structural abstractions to the maintainers. A scripting layer allows Rigi to access additional external tools.

Johnson of the National Research Council studies redundancy at the textual level. A number of uses are relevant to the SQL/DS product: looking for code reused by cut-and-paste, building a simplified model for macro processing based on actual use, and providing overviews of information content in absolute or relative (version or variant) terms.

Paul and Prakash of the University of Michigan match programming language constructs in the SCRUPLE system. Instead of looking for low-level textual patterns or very high-level semantic constructs, SCRUPLE looks for user-defined code clichés. This approach is a logical progression from simple textual scanning techniques.

Kontogiannis, De Mori, and Merlo of McGill University study semantic or behavioral pattern-matching. A transformational approach based on complexity metrics is used to simplify syntactic programming structures and expressions by translating them to tuples. The use of a distance measure on these tuples forms the basis of a method to find similar code fragments.

Defect filtering generates a overwhelming amount of information that needs to be summarized effectively to be meaningful. Extensible visualization and documentation tools such as Rigi are needed to manage these complex details. However, Rigi by itself does not offer the textual, syntactic, and semantic analysis operations needed for a comprehensive reverse engineering approach. Early results indicate that an extensible but integrated toolkit is required to support the multi-faceted analysis necessary to understand legacy software systems. Such a unified environment is under development, based on the schema and architecture implemented by the group at the University of Toronto. This integration brings the strengths of the diverse research prototypes together.

Acknowledgments

We are very grateful for the efforts of the following people: Morris Bernstein, McGill University; David Lauzon, University of Toronto; and Margaret-Anne Storey, Michael Whitney, Brian Corrie, and Jacek Walkowicz,[1] University of Victoria. Their contributions have been critical to the success of the various research prototypes. We wish to thank the SQL/DS group members at IBM for their participation and the staff at CAS for their support. Finally, we are deeply indebted to Jacob Slonim for his continued guidance

[1] Now at Macdonald-Dettwiler & Associates.

and encouragement in this endeavor.

Trademarks

AIX, IBM, OS/2, RISC System/6000, SQL/DS, System/370, VM/XA, VM/ESA, VSE/XA, and VSE/ESA, are trademarks of International Business Machines Corporation.

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

SAS is a trademark of SAS Institute, Inc.

References

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

[2] P. Selfridge, R. Waters, and E. Chikofsky. Challenges to the field of reverse engineering — a position paper. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 144–150. IEEE Computer Society Press (Order Number 3780-02), May 1993.

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

[4] R. Arnold. *Software Reengineering.* IEEE Computer Society Press, 1993.

[5] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[6] R. Arnold. Tutorial on software reengineering. In *CSM'90: Proceedings of the 1990 Conference on Software Maintenance,* (San Diego, California; November 26-29, 1990). IEEE Computer Society Press (Order Number 2091), November 1990.

[7] A. O'Hare and E. Troan. RE-Analyzer: From source code to structured analysis. *IBM Systems Journal*, 33(1), 1994.

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

[9] M. R. Olsem and C. Sittenauer. Reengineering technology report (Volume I). Technical report, Software Technology Support Center, August 1993.

[10] N. Zvegintzov, editor. *Software Management Technology Reference Guide.* Software Management News Inc., 4.2 edition, 1994.

[11] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.

[12] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance.* PhD thesis, University of Washington, 1991.

[13] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[14] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner. Using function abstraction to understand program behavior. *IEEE Software*, 7(1):55–63, January 1990.

[15] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5–67, Winter 1992.

[16] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. *ACM SIGSOFT Software Engineering Notes*, 14(3):147–150, May 1989. Proceedings of the Fifth International Workshop on Software Specification and Design.

[17] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *ICSE'14: Proceedings of the 14th International Conference on Software Engineering,* (Melbourne, Australia; May 11-15, 1992), pages 138–156, May 1992.

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

[19] E. Buss and J. Henshaw. A software reverse engineering experience. In *Proceedings of CASCON '91,* (Toronto, Ontario; October 28-30, 1991), pages 55–73. IBM Canada Ltd., October 1991.

[20] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software re-engineering. In *COMPSAC '90: Proceedings of the 14th Annual International Computer Software and Applications Conference,* (Chicago, Illinois; October, 1990), pages 314–322, 1990.

[21] J. Troster. Assessing design-quality metrics on legacy software. In *Proceedings of CASCON '92,* (Toronto, Ontario; November 9-11, 1992), pages 113–131, November, 1992.

[22] 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.

[23] E. Buss and J. Henshaw. Experiences in program understanding. In *CASCON'92: Proceedings of the 1992 CAS Conference,* (Toronto, Ontario; November 9-12, 1992), pages 157–189. IBM Canada Ltd., November 1992.

[24] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prentice-Hall, 1990.

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

[26] 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.

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

[28] 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.

[29] 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.

[30] J. K. Ousterhout. *An Introduction to Tcl and Tk.* Addison-Wesley, 1994. To be published.

[31] S. R. Tilley and H. A. Müller. Using virtual subsystems in project management. 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 144–153, July 1993. IEEE Computer Society Press (Order Number 3480-02).

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

[33] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CAS-CON '92,* (Toronto, Ontario; November 9-11, 1992), pages 171–183, November, 1992.

[34] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, March 1987.

[35] Y. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[36] L. Cleveland. PUNS: A program understanding support environment. Technical Report RC 14043, IBM T.J. Watson Research Center, September 1988.

[37] R. Al-Zoubi and A. Prakash. Software change analysis via attributed dependency graphs. Technical Report CSE-TR-95-91, Department of EECS, University of Michigan, May 1991.

[38] 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.

[39] S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, June 1994.

[40] K. Bruce and P. Wegner. An algebraic model of subtype and inheritance. In *Advances in Database Programming Languages*. ACM Press, 1990.

[41] K. Kontogiannis. Toward program representation and program understanding using process algebras. In *CASCON'92: Proceedings of the 1992 CAS Conference,* (Toronto, Ontario; November 9-12, 1992), pages 299–317. IBM Canada Ltd., November 1992.

[42] L. M. Wills. Automated program recognition: A feasibility demonstration. *Artificial Intelligence*, 45(1–2), September 1990.

[43] W. Johnson and E. Soloway. PROUST. *Byte*, 10(4):179–190, April 1985.

[44] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[45] S. Letovsky. *Plan Analysis of Programs.* PhD thesis, Department of Computer Science, Yale University, December 1988.

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

[47] A. Albrecht. Measuring application development productivity. In *Proceedings of the IBM Applications Development Symposium*, pages 83–92, October 1979.

[48] S. Henry, D. Kafura, and K. Harris. On the relationships among the three software metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality*, March 1981.

[49] V. Basili and H. Rombach. Tailoring the software process to project goals and environments. In *ICSE '9: The Ninth International Conference on Software Engineering*, pages 345–359, 1987.

[50] K. Kontogiannis, M. Bernstein, E. Merlo, and R. D. Mori. The development of a partial design recovery environment for legacy systems. In the Proceedings of *CASCON '93,* (Toronto, Ontario; October 25-28, 1993), pages 206–216, October 1993.

[51] A. Corazza, R. De Mori, R. Gretter, and G. Satta. Computation of probabilities for an island-driven parser. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1989.

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

[53] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[54] A. M. Carroll. *ConversationBuilder: A Collaborative Erector Set.* PhD thesis, University of Illinois, 1993.

The Software Refinery is composed of three parts: DIALECT (the parsing system), REFINE (the object-oriented database and programming language), and INTERVISTA (the user interface). The core of the Software Refinery is the REFINE specification and query language, a multi-paradigm high-level programming language. Its syntax is reminiscent of Lisp, but it also includes Prolog-like rules and support for set manipulation. A critical feature of the Software Refinery is its extensibility; it can be integrated into various commercial application domains.

The foundation for software analysis is a tractable representation of the subject system that facilitates its analysis. The DIALECT language model consists of a grammar used for parsing, and a domain model used to store and reference parsed programs as abstract syntax trees (AST). The domain model defines a hierarchy of objects representing the structure of a program. When parsed, programs are represented as an unanotated AST and stored using the domain model's object hierarchy. The objects are then annotated with the rules of the implementation language (such as linking each use of a variable to its declaration) and are then ready for analysis.

Sidebar "The Software Refinery"

The maintenance quality conformance hierarchy begins with low-level defects and climbs upward to broader conceptual characteristics.

Functional defects are errors in a product's function. Usually detected in Product Test or Code Review stages, they are often caused by the mistaken translation of a functional specification to implemented software. An example of a functional defect is a program expression that attempts to divide by zero.

When errors in software do not cause erroneous function but are internally incorrect, we refer to these as *non-functional defects.* These cases of "working incorrect code" often become functional defects when maintainers are making changes in the region of the non-functional defect. An example is a variable that contains an undetermined value and is referenced, but does not cause the program to fail.

Non-portable defects are characteristics that limit the software developer's ability to migrate software from one software environment to another. These environments may be new compilers, new hardware, operating systems, and so on. A familiar example of non-portable software is one that depends on the byte ordering used by the hardware or compiler.

Anti-maintenance defects are program characteristics that make use of unclear, undesirable, or side-effect features in the implementation language. Less experienced maintainers who change the software in regions where these features are present are more likely to inject further defects. Examples of this type are common, such as inconsistent use of variable naming conventions, use of keywords as variable names, and excessive use of `GOTO`s.

Part 1/2 of sidebar "The Conformance Hierarchy"

Minimizing non-portable and anti-maintenance defects means that the risks associated with software maintenance are lowered and that software produced is more "fit for change." When assertions that describe desirable software characteristics are then introduced and enforced, the software's quality is further improved.

Pro-maintenance assertions state desirable attributes of the software that help prevent defects. Many of these assertions are the opposite of anti-maintenance defects, such as the assertion "avoid the use of `GOTO`s." Another example is the inclusion of pseudocode as part of the software's internal documentation.

Design assertions capture the positive aspects of the software's structure that maintain the design quality of the code. For example, a design assertion may be "access to data structure `COMMON_DATA` is controlled by the access variable `COMMON_DATA_LATCH`, which must be set to `1` before accessing `COMMON_DATA` and set to `0` at all other times."

Architectural assertions are broad concepts that apply at a higher level of abstraction than design assertions. They seek to maintain the architectural integrity of a software system. An example is "all access to shared data structures must be controlled by a latch variable for that data structure." Often, architectural assertions are generalizations of design assertions.

Part 2/2 of sidebar "The Conformance Hierarchy"

The follling module-level measures of SQL/DS were performed as part of the D-QMA process:

- the number of lines of code (LOC) per module excluding comments;
- the number of lines of comments per module;
- the number of changed lines of code for a particular release;
- the number of lines of code in each module including `%INCLUDE` structures;
- the software maturity index $SMI(i) = \frac{LOC(i)-CSI(i)}{LOC(i)}$, (where $LOC(i)$ is the number of lines of code for module i and $CSI(i)$ is the number of changed lines of code in module i);
- the number of declared variables used in module;
- the number of declared variables in structures that are superfluous;
- the number of executable statements; and
- McCabe's cyclomatic complexity $V(G) = e - n + 2p$, [46] (where $V(G)$ is the cyclomatic number of graph G, e is the number of edges, n is the number of nodes, and p is the number of unconnected parts).

Figure 1: Module-level measurements of SQL/DS

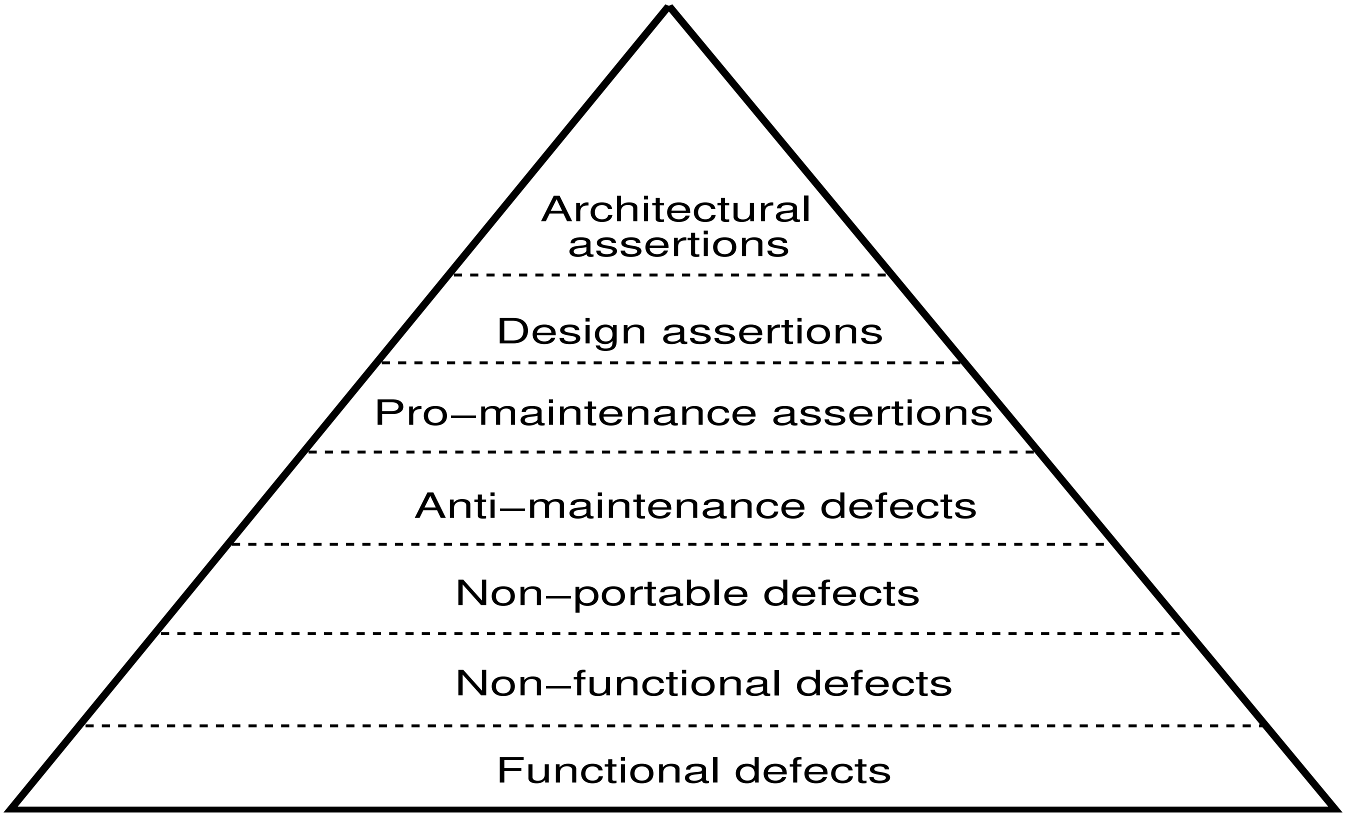

Figure 2: Maintenance quality conformance hierarchy

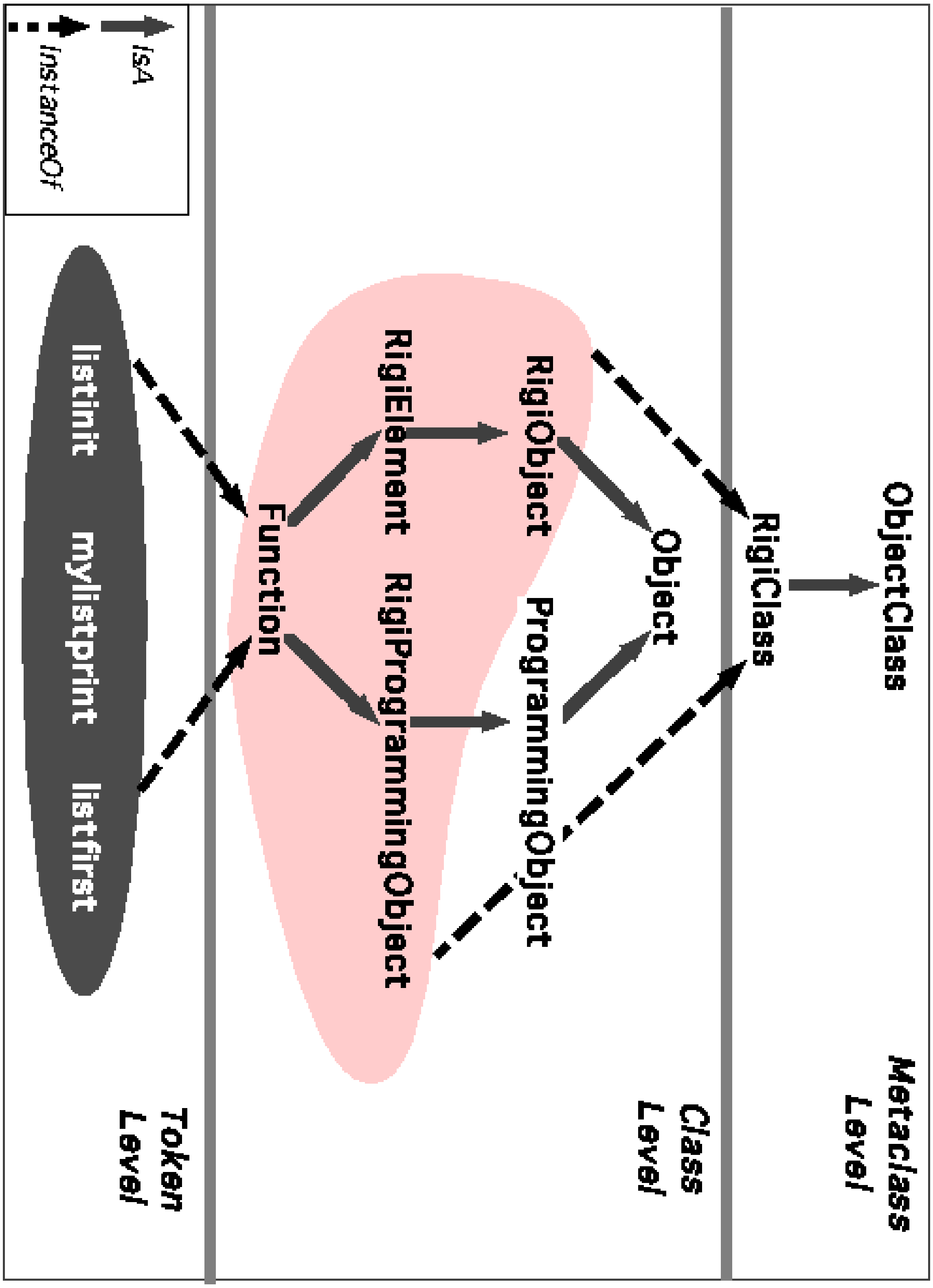

Figure 3: Repository schema

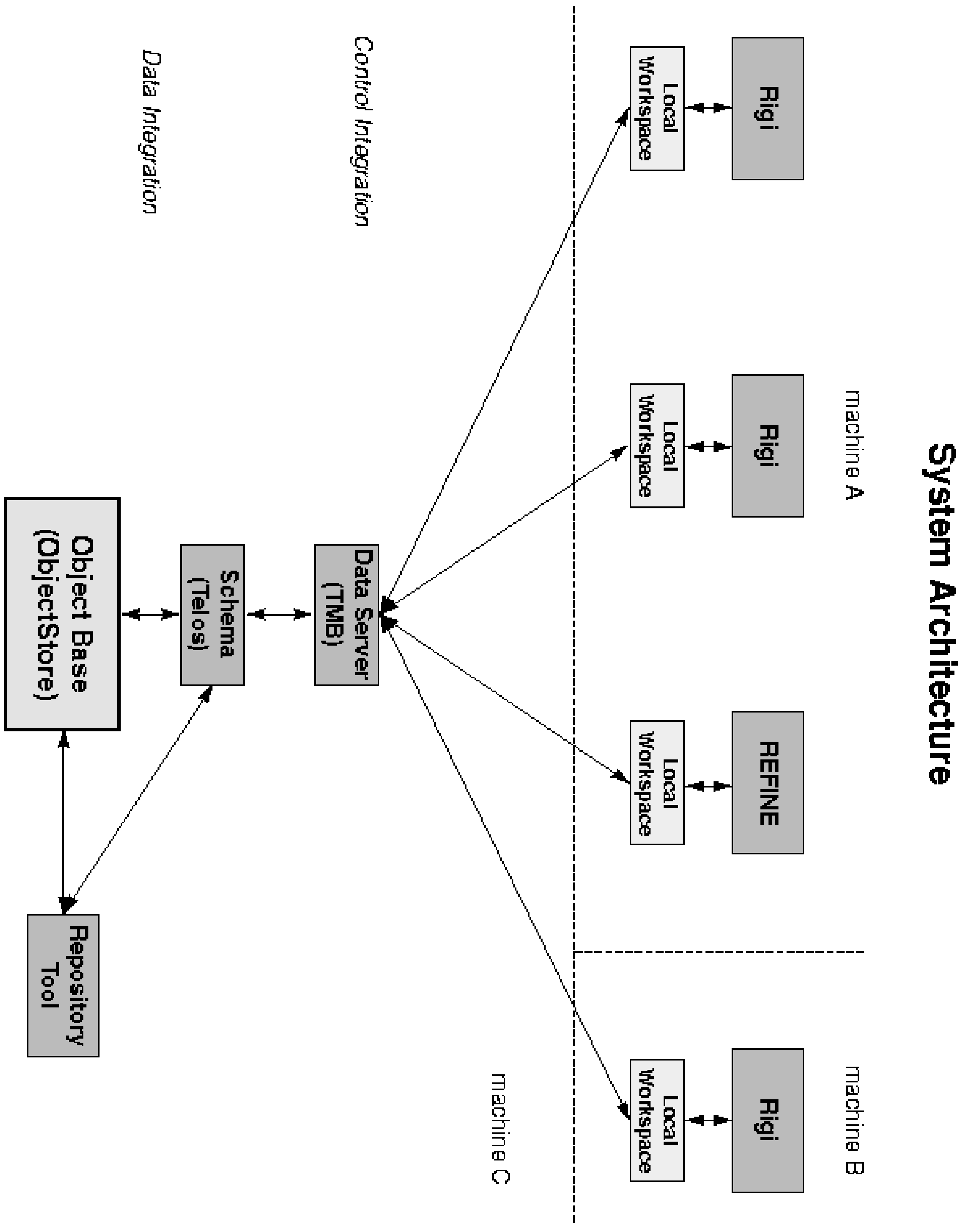

Figure 4: Environment architecture

Author's bios

Erich Buss *IBM SWS Toronto Laboratory, 41/350/895/TOR, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3. (Electronic mail: buss@vnet.ibm.com).* Mr. Buss is an Advisory Software Engineering Process Analyst in the Software Engineering Process Group of the IBM Software Solutions Toronto Laboratory. He graduated with an M.Sc in Computer Science from the University of Western Ontario in 1976. He joined IBM in the SQL/DS data group in 1988 and subsequently moved to the Centre for Advanced Studies (CAS) in 1990. In CAS he was the IBM Principal Investigator for the Program Understanding Project for three years, where he worked on the practical application of reverse engineering technology to real development problems. His current interests are in program analysis, defect filtering and object-oriented development.

Renato De Mori *McGill University, School of Computer Science, 3480 University Street, Room 318, Montréal, Québec, Canada H3A 2A7. (Electronic mail: demori@cs.mcgill.ca).* Dr. De Mori received a doctorate degree in Electronic Engineering from Politecnico di Torino, Torino, Italy, in 1967. He became full professor in Italy in 1975. Since 1986, he has been Professor and the Director of the School of Computer Science at McGill University. In 1991, he became associate of the Canadian Institute for Advanced Research and project leader of the Institute for Robotics and Intelligent Systems, a Canadian Center of Excellence. He is the author of many publications in the areas of computer systems, pattern recognition, artificial intelligence, and connectionist models. His research interests are now stochastic parsing techniques, automatic speech understanding, connectionist models, and reverse engineering. He is a fellow of the IEEE-Computer Society, has been member of various committees in Canada, Europe, and the United States, and is on the board of the following international journals: the *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *Signal Processing*, *Speech Communication*, *Pattern Recognition Letters*, *Computer Speech*, and *Language and Computational Intelligence*.

W. Morven Gentleman *Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario, Canada K1A 0R6. (Electronic mail:gentleman@iit.nrc.ca).* Dr. Gentleman is Head of the Software Engineering Laboratory in the Institute for Information Technology at the National Research Council of Canada. Before going to NRC, he was for fifteen years professor of computer science at the University of Waterloo, and before that member of technical staff at Bell Telephone Laboratories, Murray Hill. His Ph.D. is in Mathematics from Princeton in 1966. His research activities include software engineering, computer architecture, robotics, computer algebra, and numerical analysis. Dr. Gentleman has extensive experience building, supporting, and applying computer systems in research and industrial environments. He has built and supported various commercial software products.

John Henshaw *IBM SWS Toronto Laboratory, 41/350/895/TOR, 895 Don Mills Road, North York, Ontario, Canada, M3C 1W3. (Electronic mail: henshaw@vnet.ibm.com).* Mr. Henshaw is the manager of the Software Engineering Process Group in the IBM Software Solutions Toronto Laboratory. Prior to his current position, he was a staff researcher on the Program Understanding Project at the IBM Centre for

Advanced Studies for about three years. John's interests are in the fields of software engineering, database performance and modelling, and programming languages and environments.

Howard Johnson *Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario, Canada K1A 0R6. (Electronic mail: johnson@iit.nrc.ca).* Dr. Johnson is a Senior Research Officer with the Software Engineering Laboratory of the National Research Council. His current research interest is Software Re-engineering and Design Recovery using full-text approaches. He received his B.Math. and M.Math. in Statistics from the University of Waterloo in 1973 and 1974 respectively. After working as a Survey Methodologist at Statistics Canada for four years, he returned to the University of Waterloo and in 1983 completed a Ph.D. in Computer Science on applications of finite state transducers. Since then, he has been an assistant professor at the University of Waterloo and later a manager of a software development team at Statistics Canada before joining NRC.

Kostas Kontogiannis *McGill University, School of Computer Science, 3480 University Street, Room 318, Montréal, Québec, Canada H3A 2A7. (Electronic mail: kostas@binkley.cs.mcgill.ca).* Mr. Kontogiannis received a B.Sc degree in Mathematics from University of Patras, Greece, and a M.Sc degree in Artificial Intelligence from Katholieke Universiteit Leuven in Belgium. Currently, he is a Ph.D candidate at McGill University, School of Computer Science. His thesis focuses on developing plan localization algorithms and devising code similarity metrics. He is sponsored by IBM Centre for Advanced Studies and Natural Sciences and Engineering Research Council of Canada. His interests include plan localization algorithms, software metrics, artificial intelligence, and expert systems.

Ettore Merlo *DGEGI, Departement de Genie Electrique, École Polytechnique, C.P. 6079, Succ. Centre Ville, Montréal, Québec, Canada H3C 3A7. (Electronic mail: merlo@rgl.polymtl.ca).* Dr. Merlo graduated in Computer Science from the University of Turin (Italy) in 1983 and obtained the Ph.D. degree in Computer Science from McGill University in 1989. From 1989 until 1993 he was the lead researcher of the software engineering group at Computer Research Institute of Montreal (CRIM). He is currently an Assistant Professor of Computer Engineering at École Polytechnique de Montréal where his research interests include software reengineering, software analysis, and artificial intellligence. He is a member of IEEE Computer Society.

Hausi A. Müller *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6. (Electronic mail: hausi@csr.uvic.ca).* Dr. Müller is an Associate Professor of Computer Science at the University of Victoria, where he has been since 1986. From 1979 to 1982 he worked as a software engineer for Brown Boveri & Cie in Baden, Switzerland (now called ASEA Brown Boveri). He received his Ph.D. in Computer Science from Rice University in 1986. In 1992/93 he was on sabbatical at the IBM Centre for Advanced Studies in the Toronto Laboratory working with the program understanding group. His research interests include software engineering, software analysis, reverse engineering, re-engineering, programming-in-the-large, software metrics, and computational geometry. He is currently a Program Co-Chair of the *International Conference on Software Maintenance*—ICSM '94 in Victoria, a

Program Co-Chair of the *International Workshop on Computer Aided Software Engineering*—CASE '95 in Toronto, a member of the editorial board of *IEEE Transactions on Software Engineering*, and has been a Co-Chair of a *National Workshop on Software Engineering Education*—NWSEE '93 in Toronto.

John Mylopoulos *Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 1A4. (Electronic mail: jm@ai.utoronto.ca).* Dr. Mylopoulus is Professor of Computer Science at the University of Toronto. His research interests include knowledge representation and conceptual modelling, covering languages, implementations and applications. His past research accoplishments include requirements and design languages for information systems, the adoption of database implementation techniques for large knowledge bases and the application of knowledge bases to software repositories. He is currently leading a number of research projects and is principal investigator of both a national and a provincial Centre of Excellence for Information Technology. Mylopoulos received his Ph.D. degree from Princeton University in 1970. His publication list includes more than 120 refereed journal and conference proceedings papers and three edited books. He is the recipient of the first ever Outstanding Services Award given out by the Canadian AI Society (1992), also a co-recipient of a best paper award given out by the *16th International Conference on Software Engineering.*

Santanu Paul *Software Systems Research Laboratory, Dept. of EECS, University of Michigan, Ann Arbor, MI 48109. (Electronic mail: santanu@eecs.umich.edu).* Mr. Paul received his B.Tech. degree in Computer Science from the Indian Institute of Technology, Madras, in 1990 and an M.S. in Computer Science and Engineering from the University of Michigan in 1992. At present, he is a Ph.D. candidate at the University of Michigan, Ann Arbor. His thesis focuses on the design of algebraic languages to query source code. His research interests include databases, reverse engineering, and multimedia systems. He was the recipient of an IBM Canada Graduate Research Fellowship during 1991-93. He is a student member of the IEEE Computer Society.

Atul Prakash *Software Systems Research Laboratory, Dept. of EECS, University of Michigan, Ann Arbor, MI 48109. (Electronic mail: aprakash@eecs.umich.edu).* Dr. Prakash received his B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, New Delhi in 1982, and M.S. and Ph.D. degrees in Computer Science from the University of California at Berkeley in 1984 and 1989, respectively. Since 1989, he has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, where he is currently an Assistant Professor. His research interests include toolkits and architectures for supporting computer-supported cooperative work, support for re-engineering of software, and parallel simulation. His primary research focus at present is on providing distributed systems and multimedia support for carrying out computer-supported cooperative work over wide-area networks. He is a member of the ACM and the IEEE Computer Society.

Martin Stanley *Department of Computer Science, University of Toronto, 6 King's College Road, Toronto, Ontario, Canada M5S 1A4. (Electronic mail: mts@ai.utoronto.ca).* Mr. Stanley received his M.S. degree in Computer Science from the University of Toronto in 1987. His research interests include knowledge representation and conceptual modeling, with particular application to the building of software repositories. He is currently is a Research Associate in the Computer Science Department at the University of Toronto, with primary responsibilty for the Reverse Engineering project at Toronto.

Scott R. Tilley *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6. (Electronic mail: stilley@csr.uvic.ca).* Mr. Tilley is currently on leave from the IBM Software Solutions Toronto Laboratory, and is a Ph.D. candidate in the Department of Computer Science at the University of Victoria. His first book on home computing was published in 1993. His research interests include end-user programming, hypertext, program understanding, reverse engineering, and user interfaces. He is a member of the ACM and the IEEE.

Joel Troster *IBM SWS Toronto Laboratory, 41/350/895/TOR, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3. (Electronic mail: jtroster@vnet.ibm.com).* Mr. Troster is an Software Engineering Process Analyst in the Software Engineering Process Group of the IBM Software Solutions Toronto Laboratory. Joel obtained his Bachelor of Applied Sciences in Electrical Engineering in 1972 and his Masters of Applied Sciences in Biomedical Engineering in 1975, both from the University of Toronto. Joel interests include software complexity metrics, technology propagation, software development process benchmarking, enjoying family life, and growing orchids. He is a member of the IEEE Computer Society.

Kenny Wong *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC, Canada V8W 3P6. (Electronic mail: kenw@csr.uvic.ca).* Mr. Wong is a Ph.D. candidate in the Department of Computer Science at the University of Victoria. His research interests include program understanding, user interfaces, and software design. He is a member of the ACM, USENIX, and the Planetary Society.

END