Investigating reverse engineering technologies for the CAS program understanding project

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

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 the problem of program understanding, software engineers are spending an ever-growing amount of effort on reverse engineering technologies. This paper describes the scope and results of an ongoing research project on program understanding undertaken by the IBM Toronto Software Solutions Laboratory Centre for Advanced Studies (CAS). The project involves a team from CAS and five research groups working cooperatively on complementary reverse engineering approaches. All the groups are using the source code of SQL/DS™ (a multimillion-line relational database system) as the reference legacy system. Also discussed is an approach adopted to integrate the various tools under a single reverse engineering environment.

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 10 percent each year just in enhancements, updates, and normal maintenance. As a result of maintenance alone, software inventories will double in size every seven years. Since these systems cannot easily be replaced without re-

viewing their entire history, managing long-term software evolution is critical. It has been estimated that 50 to 90 percent of work each year is devoted to program understanding;[1] hence, facilitating 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, which has been proposed to help refurbish and maintain software systems. The importance of reverse engineering[2] will grow accordingly as maintenance and reengineering costs for large legacy software systems increase. 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.

This paper describes the use of several complementary reverse engineering technologies applied to a real-world software system: Structured Query Language/Data System (SQL/DS*). The goal was to aid the maintainers of SQL/DS to improve product quality by enhancing their understanding of the three million lines of source code. The background on the genesis of the program understanding project and its focus on the SQL/DS product is described and subsequent sections detail the individual research programs. Defect filtering is discussed 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; thus the section on structural redocumentation discusses a system to reconstruct and present high-level documentation for software understanding. A comprehensive approach to reverse engineering requires many different techniques, and three techniques are outlined that analyze source code at textual, syntactic, and semantic levels. Finally the convergence of the separate research prototypes into an integrated reverse engineering environment is reported. The paper concludes with the important lessons learned in this endeavor.

Background

Faced with demanding and ambitious quality-related objectives, the SQL/DS product group at IBM 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) systems, and to utilize program understanding analysis to improve the quality of the SQL/DS product and to improve the productivity of the software organization.

The philosophy of the IBM Centre for Advanced Studies (CAS) in Toronto 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.[3] 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 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 Toronto Software Solutions 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.

The reference system: SQL/DS. SQL/DS 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 by IBM in 1982. Originally written in PL/I to run on the Virtual Machine System (VM), SQL/DS now consists of over 3 000 000 lines of PL/AS code. PL/AS (as PL/I) is a proprietary IBM systems programming language that is PL/I-like and allows embedded System/370* assembler language code to be part of the instruction stream. Because PL/AS is a proprietary language, commercial off-the-shelf analysis tools are unsuitable.

SQL/DS consists of about 1300 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 that is current and accurate. 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 CAS 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: (1) detecting uninitialized data, pointer errors, and memory leaks, (2) detecting data type mismatches, (3) finding incomplete uses of record fields, (4) finding similar code fragments, (5) localizing algorithmic plans, (6) recognizing inefficient or high-complexity code, and (7) predicting the impact of change.

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[4] 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 match patterns 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.[5]

The *reverse engineering process* involves two distinct phases:[6] (1) the identification of the current components of the system and their dependencies, and (2) the discovery of system abstractions and design information.[7] During this process, the source code is not altered, although additional information about the system is generated. In contrast, the entire *re-engineering process* typically consists of a reverse engineering phase, followed by a forward engineering or reimplementation phase that alters the source code of the subject system. Definitions of related concepts may be found in Reference 8.

The focus was on the source code of the SQL/DS product.

The discovery phase of reverse engineering 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.[9] 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.

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 the need for the focus on features such as control flows, global variables, data structures, and resource exchanges. At a higher semantic level, research 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 de-

scribe several hundred such packages.[10,11] Most commercial systems focus on source code analysis and simple code restructuring, and use information abstraction via program analysis, the most common form of reverse engineering. Research in reverse engineering consists of many diverse approaches, including formal transformations,[12] meaning-preserving restructuring,[13] plan recognition,[14] function abstraction,[15] information abstraction,[16] maverick identification,[17] graph queries,[18] and reuse-oriented methods.[19]

Defect filtering, structural documentation, and pattern-matching analyses are used.

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 toward 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 team 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 overall CAS project is to integrate the results of the complementary (but sometimes overlapping) research efforts to produce a more comprehensive reverse engineering set of tools. This integration process is described more fully in the section "Steps toward integration." The sections that now follow describe the program understanding research results on defect filtering, structural redocumentation, and pattern matching.

Defect filtering

The IBM team[20] performed defect filtering[21] using the commercial product Software Refinery** (REFINE**)[22] to parse the source code of SQL/DS into a form suitable for analysis. This work applied the experience of domain experts to create REFINE "rules" to find certain families of defects in the subject software. These defects included 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 focused on detecting specific errors in the reference system.

A design-quality metrics analysis (D-QMA) was also performed[23] on SQL/DS.[24] This analysis included measurements that 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 proved to be a fruitful approach to improve the quality of the reference system.[25] We next describe the evolution of the defect filtering process that consists of 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 filtering for quality.

Building a reverse engineering toolkit. Most application problem domains have unique and specialized characteristics; therefore, the expectations and requirements for reverse engineering tools vary, and the tools must be extensible and versatile. It is unlikely that a turnkey reverse engineering package will suffice for most users. This is especially true for analyzing systems of a proprietary nature such as SQL/DS. Unless it is known exactly what is to be accomplished, a priority should be placed on toolkit flexibility. Because of these considerations, the Software Refinery product was chosen as the basis upon which to build a PL/AS reverse engineering toolkit for the defect filtering process.

The Software Refinery product is composed of three parts: DIALECT (the parsing system), REFINE (the object-oriented database and pro-

gramming language), and INTERVISTA (the user interface). The core of Software Refinery is the REFINE specification and query language, a multiparadigm 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 product 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 unannotated AST and stored using the object hierarchy of the domain model. 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.

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 product was used to convert the SQL/DS source code into a more tractable form, or a form more easily analyzed. 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 the PL/AS functions. 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 source code for the reference system. 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 apply to similar programming languages, and can be evolved as implementation rules change.

Once the SQL/DS source code was put in this tractable form, the customers (the SQL/DS maintainers) were consulted to determine how best to utilize this technology for them. The answer was to help remove defects from the code. The challenge was how to do it effectively. The solution was to apply the power of the prototype environment to analyze 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 semiautomatically.

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 toward defect detection, due in part to the quality-related objectives of the SQL/DS product group. 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 were 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

Table 1 Module-level measurements of SQL/DS

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

- Number of lines of code (LOC) per module excluding comments
- Number of lines of comments per module
- Number of changed lines of code for a particular release
- Number of lines of code in each module including %INCLUDE structures
- 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
$CSI(i)$ is the number of changed lines of code in module i

- Number of declared variables used in module
- Number of declared variables in structures that are superfluous
- Number of executable statements
- McCabe's cyclomatic complexity

$$V(G) = e - n + 2p$$

where $V(G)$ is the cyclomatic number of graph G
e is the number of edges
n is the number of nodes
p is the number of unconnected parts

have a similar 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 were specifically built to demonstrate the capability for analysis in all of these domains. Some of the prototypes are documented in Reference 26. 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.

Design-quality metrics analyses. While maintenance goals continue to focus on 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, the design-quality metrics analysis (D-QMA) project was initiated.

The purpose of assessing design-quality metrics[27] is to examine the design process by examining the end product (source code) to predict the quality of a product 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
- 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 defect history of the product.

Intermodule metrics for module-level design measure intermodule coupling and cohesion, data flow between modules, and so on. These "black-box" measures require no knowledge of the inner workings of the module. Intramodule design metrics include measures of control flow, data flow, and logic within a module. These "clear-box" measures do require knowledge of the inner working of the module. Both the intermodule and intramodule versions of structural complexity, data complexity, and system complexity[28] were measured. Other module-level measurements are shown in Table 1.

The experiment applied the reverse engineering toolkit (previously discussed) to extract the metrics from the reference system. Defect data were gathered from the defect database (which existed on the fast system) and were then correlated using the SAS** statistical package running on the Operating System/2* (OS/2*) workstation. For SQL/DS Version 3, Release 3, about nine hours of machine time (on a RISC System/6000* Model 550) were required to analyze all 1303 PL/AS modules. This time does not include the 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 occurred when the smallest change was made. This counterintuitive 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 analysis work is continuing using other IBM products written in PL/AS, PL/MI, C, and C++.

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 is concerned with improved methods for managing risk, increased productivity, and reduced 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 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.

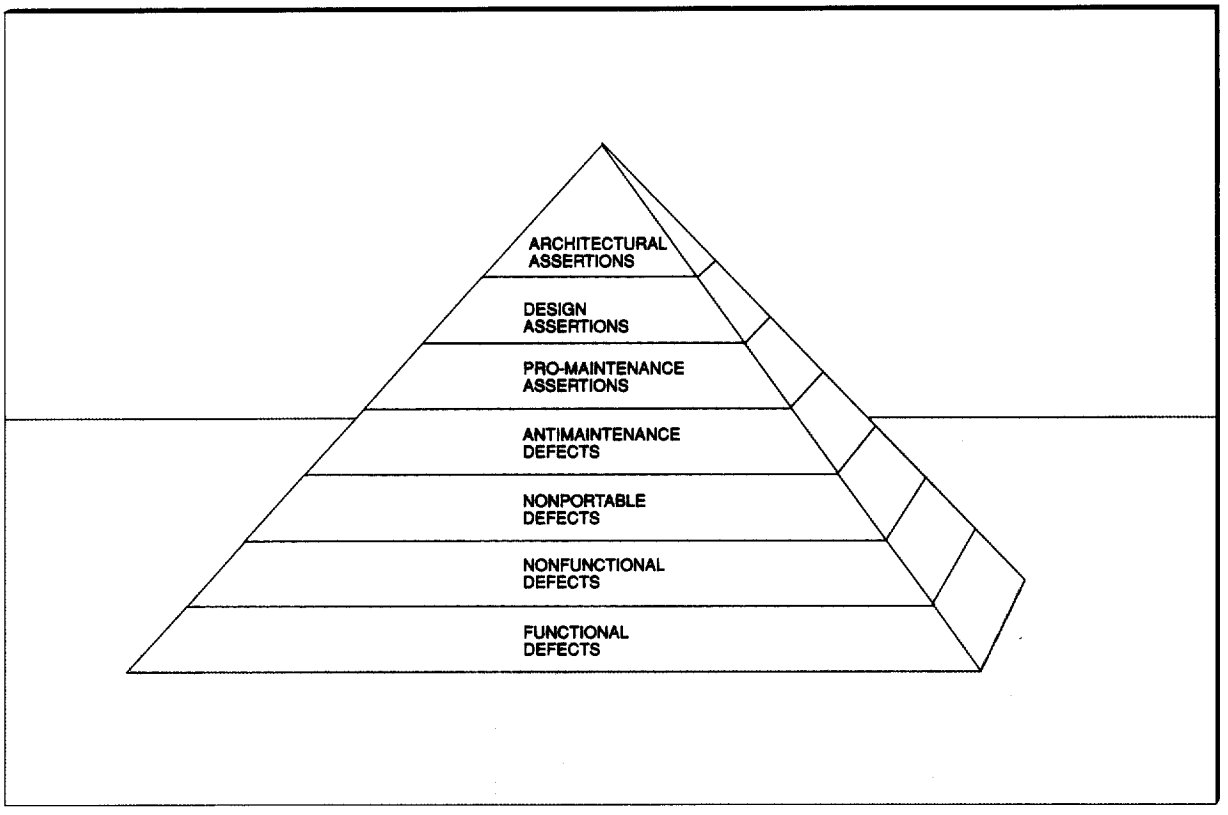
By extending the meaning of what constitutes a defect, one can expand the definition of quality. For example, the recognition of defects caused by coding standard violations means that quality is no longer bound to purely functional characteristics; quality attributes can be extended to include indirect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

Further extension to the quality framework may include assertions that must be adhered to; assertion nonconformance 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 1 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 limiting the definition of correctness, one can build higher quality software.

Functional defects are function errors in a product. 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 *nonfunctional defects*. These cases of "working incorrect code" often become functional defects when maintainers are making changes in the region of the nonfunctional defect. An example is a variable that contains an undetermined value and is referenced, but does not cause the program to fail.

Figure 1 Maintenance quality conformance hierarchy



Nonportable 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 nonportable software is one that depends on the byte ordering used by the hardware or compiler.

Antimaintenance 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 instructions.

Minimizing nonportable and antimaintenance 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 quality of the software is further improved.

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

Design assertions capture the positive aspects of the software 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 the data structure." Often, architectural assertions are generalizations of design assertions.

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 the earlier section "Building a reverse engineering toolkit." 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 some tools.

There are many benefits of automation to the filtering for quality 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 domain of the filter. 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.

Summary of defect filtering. 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 potentially 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 the earlier sections have been partially integrated into the mainstream SQL/DS maintenance process.

Measurable results come from measurable problems. Defect filtering can produce 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.

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 1000 lines of code versus those of 1000000 lines. Typical software documentation describes the program in terms of isolated algorithms and data structures. More-

over, 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 process that is needed to expose the overall architecture of large software systems.

There are trade-offs between what can be and should be automated.

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.[29] 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 work at the University of Victoria is centered around Rigi,[30] an environment for understanding evolving software systems. Output from this environment can also serve as input to conceptual modeling, 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, multiuser repository to store the extracted information; and an interactive, window-oriented graph editor to manipulate structural representations.

Scalability. Effective approaches to program understanding must be applicable to huge, multimillion-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 upward to large systems. 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.[31]

Redocumentation strategy. There are trade-offs in program understanding environments between what can be automated and what should (or must) be left for processing by 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 semiautomatic 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.[32] 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 and 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.[32] 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.

Multiple dynamic views. Visual representations enhance the human ability to recognize patterns. Using the graph editor of Rigi, 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 semiautomatic 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 coexist under the structural representation supported by Rigi. These decompositions provide many possible perspectives for later review. In effect, multiple, logical representations of the software architecture can be created, manipulated, and saved.

Multiple domains. 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 allow the user to fully customize the user interface. This approach permits analysts to tailor the environment to better suit their needs, providing a smooth transition between automatic and semiautomatic reverse engineering. The goal to have 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 easier to program and to enhance, the user interface and editor engine were decoupled to make room for an intermediate scripting layer based on embedded Tcl and Tk libraries.[33] 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,[34] personalized hypertext,[35] and redocumenting legacy software systems.

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 multimillion-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 a mental model of its structure, based on development logbooks and experience. This model was recreated using the Rigi 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.

Summary of structural redocumentation. 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.

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 that are similar to each other.

Program understanding techniques may use source code in increasingly abstract forms, including: raw text, preprocessed text, lexical tokens, syntax trees, annotated abstract syntax trees with symbol tables, and control or 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, in-line functions, and file inclusions. Three research groups affiliated with the program understanding project focused on textual, syntactic, and semantic pattern-matching approaches.

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 amounts of source code have a large internal structure as a result of their evolution. The NRC (one member group of the programming understanding project) 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 at NRC[36] identifies the 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, white space, 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-dependent and environmentally dependent; slight changes in these aspects can make the tools inapplicable. For example, C language 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 the determination of the ef-

fects of cut-and-paste, discovery of the effects of preprocessing, measurement of the changes between versions, and the 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.

The several issues to be addressed are the discovery of efficient algorithms for computing fingerprints, determination of the appropriate set of substrings, and the devising of postprocessing techniques to make the generated fingerprint file more useful. Karp and Rabin[37] 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), and 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 Model 550. 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.

Measures of repetition are a useful basis for building program understanding tools.

Syntactic analysis. The effort at the University of Michigan[38] 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 a 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.

Deficiencies with current approaches. Despite the critical nature of the task, good source code search systems do not exist. General string-searching tools 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 also contains numerous syntactic, structural, and spatial relationships that are not

fully captured by the entity-relation-attribute model of a relational database.

For example, systems such as the C Information Abstraction system (CIA)[39] and PUNS[40] 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[41] 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 (previously discussed in the section "Defect filtering"), 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.

Syntactic, structural, and spatial relationships are not captured by models.

SCRUPLE. The University of Michigan group has developed the SCRUPLE source code search system (Source Code Retrieval Using Pattern LanguagEs).[42] 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 that 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.

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, a source code algebra (SCA)[43] was designed 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. Fourth, the 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[44] 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 a parser generator.

Semantic analysis. The McGill research[45] 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.

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 lambda calculus. The McGill representation scheme is an object-oriented annotated AST.

A grammar and a domain model for the language of the subject system is constructed using REFINE. 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, file names, and links between identifier references and corresponding variable and data type definitions. Annotations produced by analysis routines include variables used and updated, functions called, variable scope information, input/output operations, and complexity and quality metrics. Annotations stored in the AST may be used by other analysis routines.

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[46] compares the attributed data flow subgraphs of code fragments and algorithmic plans, and uses control dependencies as additional constraints. PROUST[47,48] 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[49] 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 focuses 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 the McGill 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.

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 previously discussed in the section "Textual analysis." 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 cannot be made. 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 $c1$ and $c2$ are similar under a set of features measured by metric M, then their metric values $M(c1)$ and $M(c2)$ 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 input/output variables to the fan-out
3. McCabe's cyclomatic complexity[50]
4. Albrecht's Function Point quality metric[51]
5. Henry-Kafura's information flow quality metric[52]

Similarity is gauged by a distance measure on the tuples. The distances currently used are based on two measures: (1) on the Euclidean distance defined in the five-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 track 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.

Goal-driven program understanding. Another design recovery strategy that has been explored by the McGill group is a variation of the GQM[53] model, which is a goal, question, analysis, and action model.[54] 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[55] 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.

Summary of pattern matching. 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 and research is continuing on improving these tools. The NRC group is also focusing on better visualization techniques. Michigan is investigating better program representations and pattern-matching engines, and McGill is exploring techniques for plan recognition and similarity distances between source code features.

The common themes that have emerged from this research are: (1) domain-specific knowledge is critical in easing the interpretation of large software systems, (2) program representations for efficient queries are essential, (3) many kinds of analyses are needed in a comprehensive reverse engineering approach, and (4) an extensible environment is needed to consolidate these diverse approaches into a unified framework. An architecture for a multifaceted reverse engineering environment to address these requirements is presented in the next section.

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 the IBM team is used daily by several development groups, including SQL/DS and DB2. The second phase of the program understanding project focuses 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, in the implementation of such descriptions in terms of physical data structures, and in the mechanisms deployed 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.[56] Integrating the representations employed by REFINE and Rigi is a nontrivial 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:

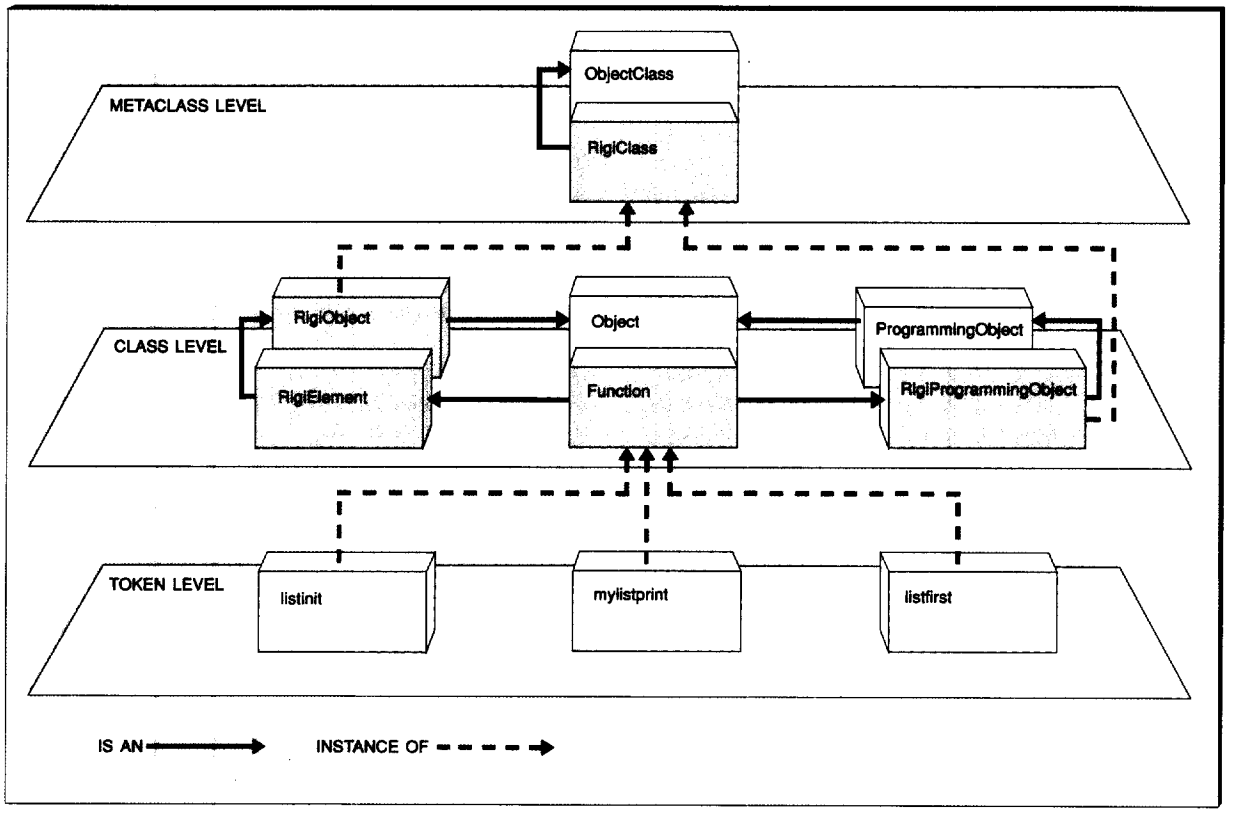
- Smooth data, control, and presentation integration among components of the environment
- Extensible data model and interfaces to support new tools and user-defined objects, dependencies, and functions
- Domain-specific, semantic pattern matching to complement the facilities developed during the first phase of the project
- The representation and support of processes and methodologies for reverse engineering
- 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.

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. A set of requirements was created for the repository. 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 subschemas 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 named Module. By contrast, since this concept is not supported by REFINE, the programming language construct of an arithmetic expression is captured in the REFINE subschema using the class Expression. As an example of a shared concept, the notation of a function is common to both tools and is captured in the shared class Function. Each tool has a different view of this class, where only the common portions and the information perti-

Figure 2 The repository schema

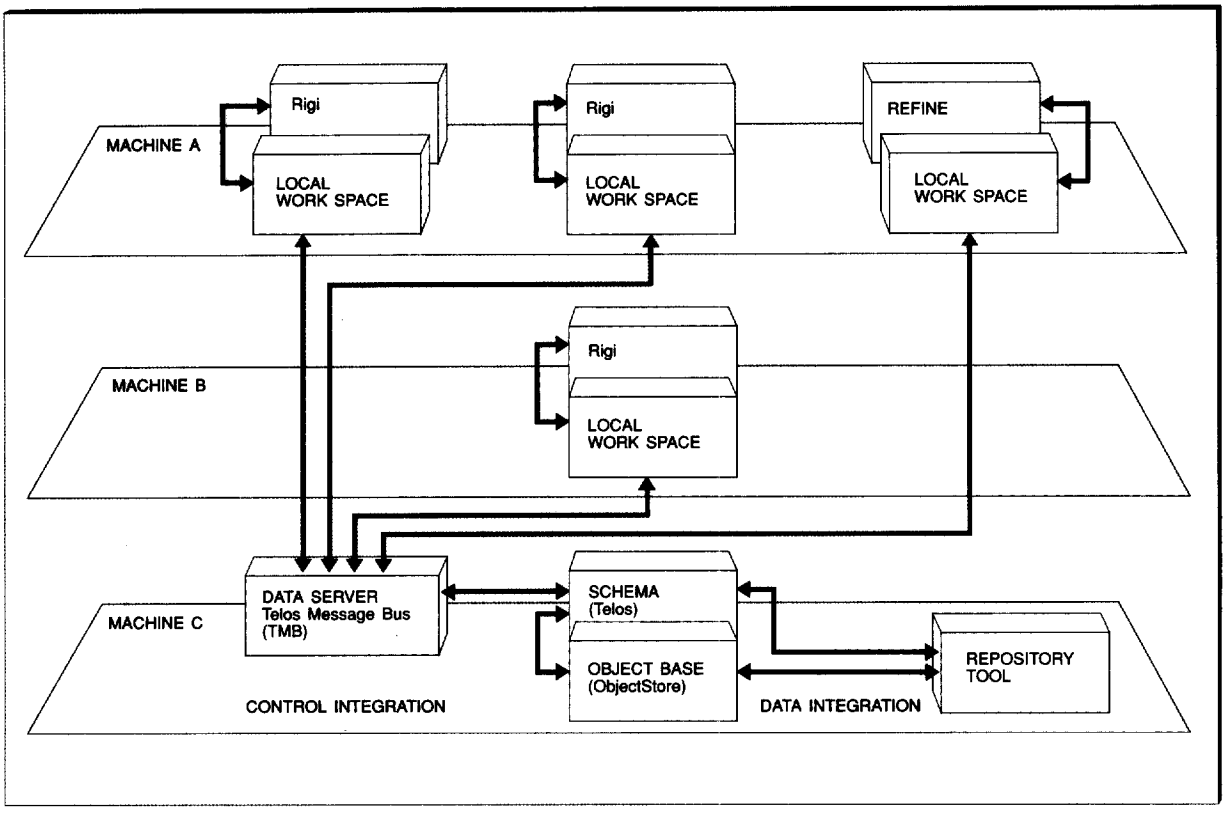


nent to that tool are accessible. The second phase classifies the patterns used and captures the analysis results generated from each tool. The third phase will record other information relevant to reverse engineering, such as designs, system requirements, domain modeling, and process information. The 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.[57] Features of Telos include: an object-oriented framework that supports generalization, classification and attribution, a metamodeling 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**, or C++-based models) 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 2, the schema consists of three tiers. The top level (Metaclass Level) exploits meta modeling facilities to define the types of attribute values that the repository supports, and 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 pertain 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, using the metaclasses and attributes defined in the top level. For instance,

Figure 3 System architecture



RigiObject, RigiElement, RigiProgrammingObject, and Function (grouped in the shaded area in Figure 2), 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 2 shows three function objects: listinit, mylistprint, and listfirst corresponding to the actual function definitions. These are created when Rigi parses the target source code.

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 involve 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 named the Telos Message Bus (TMB), as shown in Figure 3. This message server allows all tools to communicate both with the repository and with each other, using the common schema. 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[58] 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 project. This earlier 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.

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.

The IBM team built several prototype toolkits in REFINE, each focusing on detecting specific errors in SQL/DS. A flexible approach was also developed that applies defect filters to the source code to improve the quality. Defect filtering produces measurable results in software quality.

The University of Victoria group 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.

The National Research Council studied 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.

The University of Michigan group matched 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.

The McGill University group studied 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 an 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 multifaceted 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 (now at Macdonald-Dettwiler & Associates), 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 and encouragement in this endeavor.

Cited references and notes

1. T. A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering* **SE-10**, No. 5, 494–497 (September 1984).
2. In this paper, re-engineering means the authorized logical conversion of a customized architecture (implemented in SQL/DS) to a commercial architecture (implemented in a compiler source language). Re-engineering does not include unauthorized reverse compilation of object code to

form source code as the basis for the derivation of a substitute product, generally referred to as *reverse engineering*.
3. P. Selfridge, R. Waters, and E. Chikofsky, "Challenges to the Field of Reverse Engineering—A Position Paper," *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering*, Baltimore, MD; IEEE Computer Society Press, Order Number 3780-02 (May 1993), pp. 144–150.
4. R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies* **18**, 543–554 (1983).
5. R. Arnold, *Software Reengineering*, IEEE Computer Society Press (1993).
6. R. Arnold, "Tutorial on Software Reengineering," *CSM'90: Proceedings of the 1990 Conference on Software Maintenance*, San Diego, CA; IEEE Computer Society Press, Order Number 2091 (November 1990).
7. E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software* **7**, No. 1, 13–17 (January 1990).
8. A. B. O'Hare and E. W. Troan, "RE-Analyzer: From Source Code to Structured Analysis," *IBM Systems Journal* **33**, No. 1, 110–130 (1994).
9. G. Myers, *Reliable Software Through Composite Design*, Petrocelli/Charter (1975).
10. M. R. Olsem and C. Sittenauer, *Reengineering Technology Report, Volume I*, Technical Report, Software Technology Support Center (August 1993).
11. *Software Management Technology Reference Guide*, N. Zvegintzov, Editor, Software Management News Inc., 4.2 Edition (1994).
12. G. Arango, I. Baxter, P. Freeman, and C. Pidgeon, "TMM: Software Maintenance by Transformation," *IEEE Software* **3**, No. 3, 27–39 (May 1986).
13. W. G. Griswold, *Program Restructuring as an Aid to Software Maintenance*, Ph.D. thesis, University of Washington, Seattle, WA (1991).
14. C. Rich and L. M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software* **7**, No. 1, 82–89 (January 1990).
15. P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software* **7**, No. 1, 55–63 (January 1990).
16. J. E. Grass, "Object-Oriented Design Archaeology with CIA++," *Computing Systems* **5**, No. 1, 5–67 (Winter 1992).
17. R. Schwanke, R. Altucher, and M. Platoff, "Discovering, Visualizing, and Controlling Software Structure," *ACM SIGSOFT Software Engineering Notes* **14**, No. 3, 147–150 (May 1989).
18. M. Consens, A. Mendelzon, and A. Ryman, "Visualizing and Querying Software Structures," *Proceedings of the 14th International Conference on Software Engineering*, Melbourne, Australia, May 11–15, 1992, pp. 138–156 (May 1992).
19. T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The Concept Assignment Problem in Program Understanding," *Proceedings of the 1993 Working Conference on Reverse Engineering*, Baltimore, Maryland, May 21–23, 1993, pp. 27–43; IEEE Computer Society Press, Order Number 3780-02 (May 1993).
20. The IBM team was led by authors E. Buss and J. Henshaw.
21. E. Buss and J. Henshaw, "A Software Reverse Engineering Experience," *Proceedings of CASCON '91*, Toronto, Ontario, October 28–30, 1991, pp. 55–73; IBM Canada Ltd. (October 1991).
22. S. Burson, G. B. Kotik, and L. Z. Markosian, "A Program Transformation Approach to Automating Software Re-engineering," *Proceedings of the 14th Annual International Computer Software and Applications Conference*, Chicago, IL, October, 1990, pp. 314–322 (1990).
23. J. Troster, a member of the IBM team, performed the design-quality metrics analysis.
24. J. Troster, "Assessing Design-Quality Metrics on Legacy Software," *Proceedings of CASCON '92*, Toronto, Ontario, November 9–11, 1992), pp. 113–131 (November 1992).
25. J. Troster, J. Henshaw, and E. Buss, "Filtering for Quality," *Proceedings of CASCON '93*, Toronto, Ontario, October 25–28, 1993, pp. 429–449 (October 1993).
26. E. Buss and J. Henshaw, "Experiences in Program Understanding," *Proceedings of the 1992 CAS Conference*, Toronto, Ontario, November 9–12, 1992, pp. 157–189; IBM Canada Ltd. (November 1992).
27. D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1990).
28. D. N. Card, "Designing Software for Producibility," *Journal of Systems and Software* **17**, No. 3, 219–225 (March 1992).
29. H. L. Ossher, "A Mechanism for Specifying the Structure of Large, Layered Systems," *Research Directions in Object-Oriented Programming*, B. D. Shriver and P. Wegner, Editors, The MIT Press (1987), pp. 219–252.
30. H. A. Müller, *Rigi—A Model for Software System Construction, Integration, and Evolution Based on Module Interface Specifications*, Ph.D. thesis, Rice University (August 1986).
31. M. Shaw, "Larger-Scale Systems Require Higher-Level Abstractions," *ACM SIGSOFT Software Engineering Notes* **14**, No. 3, 143–146 (May 1989).
32. 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**, No. 4, 181–204 (December 1993).
33. J. K. Ousterhout, *Tcl and Tk Toolkit*, Addison-Wesley Publishing Co., Reading, MA (1994).
34. S. R. Tilley and H. A. Müller, "Using Virtual Subsystems in Project Management," *The Sixth International Conference on Computer-Aided Software Engineering*, Institute of Systems Science, National University of Singapore, Singapore, July 19–23, 1993; IEEE Computer Society Press, Order Number 3480-02 (July 1993), pp. 144–153.
35. S. R. Tilley, M. J. Whitney, H. A. Müller, and M.-A. D. Storey, "Personalized Information Structures," *The 11th Annual International Conference on Systems Documentation*, Waterloo, Ontario, October 5–8, 1993; ACM Order Number 6139330 (October 1993), pp. 325–337.
36. J. H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proceedings of 1992 CAS Conference*, Toronto, Ontario, November 9–12, 1992, pp. 171–183; IBM Canada Ltd. (November 1992).
37. R. M. Karp and M. O. Rabin, "Efficient Randomized Pattern-Matching Algorithms," *IBM Journal of Research and Development* **31**, No. 2, 249–260 (March 1987).

38. The University of Michigan team was lead by authors S. Paul and A. Prakash.
39. Y. Chen, M. Nishimoto, and C. Ramamoorthy, "The C Information Abstraction System," *IEEE Transactions on Software Engineering* **16**, No. 3, 325–334 (March 1990).
40. L. Cleveland, *PUNS: A Program Understanding Support Environment*, Technical Report RC 14043, IBM T. J. Watson Research Center (September 1988).
41. 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).
42. S. Paul and A. Prakash, "Source Code Retrieval Using Program Patterns," *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, Montreal, Quebec, July 6–10, 1992 (July 1992), pp. 95–105.
43. S. Paul and A. Prakash, "A Framework for Source Code Search Using Program Patterns," *IEEE Transactions on Software Engineering* **20**, No. 6 (June 1994).
44. K. Bruce and P. Wegner, "An Algebraic Model of Subtype and Inheritance," *Advances in Database Programming Languages*, ACM Press (1990).
45. K. Kontogiannis, "Toward Program Representation and Program Understanding Using Process Algebras," *Proceedings of the 1992 CAS Conference*, Toronto, Ontario, November 9–12, 1992; IBM Canada Ltd. (November 1992), pp. 299–317.
46. L. M. Wills, "Automated Program Recognition: A Feasibility Demonstration," *Artificial Intelligence* **45**, 1–2 (September 1990).
47. W. Johnson and E. Soloway, "PROUST," *Byte* **10**, No. 4, 179–190 (April 1985).
48. W. Kozaczynski, J. Ning, and A. Engberts, "Program Concept Recognition and Transformation," *IEEE Transactions on Software Engineering* **18**, No. 12, 1065–1075 (December 1992).
49. S. Letovsky, *Plan Analysis of Programs*, Ph.D. thesis, Department of Computer Science, Yale University (December 1988).
50. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* **7**, No. 4, 308–320 (September 1976).
51. A. J. Albrecht, "Measuring Application Development Productivity," *Proceedings of IBM Applications Development Symposium*, Monterey, CA (October, 1979), pp. 83–92.
52. S. Henry, D. Kafura, and K. Harris, "On the Relationships among the Three Software Metrics," *Proceedings of 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality* (March 1981).
53. V. Basili and H. Rombach, "Tailoring the Software Process to Project Goals and Environments" *The Ninth International Conference on Software Engineering* (1987), pp. 345–359.
54. K. Kontogiannis, M. Bernstein, E. Merlo, and R. D. Mori, "The Development of a Partial Design Recovery Environment for Legacy Systems," *Proceedings of CASCON '93*, Toronto, Ontario, October 25–28, 1993 (October 1993), pp. 206–216.
55. 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* **13**, No. 9, 36–50 (Sept. 1991).
56. N. Kiesel, A. Schürr, and B. Westfechtel, "GRAS: A Graph-Oriented Database System for (Software) Engineering Applications, *The Sixth International Conference on Computer-Aided Software Engineering*, Institute of Systems Science, National University of Singapore, Singapore, July 19–23, 1993; IEEE Computer Society Press, Order Number 3480-02 (July 1993), pp. 272–286.
57. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing Knowledge about Information Systems," *ACM Transactions on Information Systems* **8**, No. 4, 325–362 (October 1990).
58. A. M. Carroll, *ConversationBuilder: A Collaborative Erector Set*, Ph.D. thesis, University of Illinois (1993).

Accepted for publication April 20, 1994.

Erich Buss *IBM Software Solutions Division, Toronto Laboratory, IBM Canada Ltd., 895 Don Mills Road, North York, Ontario M3C 1W3, Canada (electronic mail: buss@vnet.ibm.com).* Mr. Buss is an advisory software engineering process analyst in the Software Engineering Process Group of the IBM Toronto Software Solutions 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 IBM Centre for Advanced Studies (CAS) in 1990. In CAS he was the 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 H3A 2A7, Canada (electronic mail:demori@cs.mcgill.ca).* Dr. De Mori received a doctoral 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 a professor and the Director of the School of Computer Science at McGill University. In 1991, he became an 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 many international journals.

W. Morven Gentleman *Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario K1A 0R6, Canada (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 a member of the technical staff at Bell Telephone Laboratories, Murray Hill, New Jersey, and for 15 years a professor of computer science at the University of Waterloo. 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 Software Solutions Division, Toronto Laboratory, IBM Canada Ltd., 895 Don Mills Road, North York, Ontario M3C 1W3, Canada (electronic mail: henshaw@vnet.ibm.com).* Mr. Henshaw is the manager of the Software Engineering Process Group in the IBM Toronto Software Solutions 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. Mr. Henshaw's interests are in the fields of software engineering, database performance and modeling, and programming languages and environments.

Howard Johnson *Institute for Information Technology, National Research Council Canada, Montreal Road, Building M-50, Ottawa, Ontario K1A 0R6, Canada (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 the National Research Council.

Kostas Kontogiannis *McGill University, School of Computer Science, 3480 University Street, Room 318, Montréal, Québec H3A 2A7, Canada (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 the IBM Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada. His interests include plan localization algorithms, software metrics, artificial intelligence, and expert systems.

Ettore Merlo *Departement de Genie Electrique, École Polytechnique, C.P. 6079, Succ. Centre Ville, Montréal, Québec H3C 3A7, Canada (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 the 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 intelligence. He is a member of the IEEE Computer Society.

Hausi A. Müller *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC V8W 3P6, Canada (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 and 1993 he was on sabbatical leave 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; and a member of the editorial board of *IEEE Transactions on Software Engineering*. He was previously co-chair of the *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 M5S 1A4, Canada (electronic mail:jm@ai.utoronto.ca).* Dr. Mylopoulos is Professor of Computer Science at the University of Toronto. His research interests include knowledge representation and conceptual modeling, covering languages, implementations, and applications. His past research accomplishments 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. Dr. 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 by the Canadian AI Society (1992), and is also a co-recipient of a best paper award given by the *16th International Conference on Software Engineering*.

Santanu Paul *Software Systems Research Laboratory, Department of EECS, University of Michigan, Ann Arbor, Michigan 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 re-

search 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, Department of EECS, University of Michigan, Ann Arbor, Michigan 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 M5S 1A4, Canada (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 a research associate in the Computer Science Department at the University of Toronto, with primary responsibility for the reverse engineering project at Toronto.

Scott R. Tilley *Department of Computer Science, University of Victoria, P.O. Box 3055, Victoria, BC V8W 3P6, Canada (electronic mail:stilley@csr.uvic.ca).* Mr. Tilley is currently on leave from the IBM Toronto Software Solutions 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 Software Solutions Division, Toronto Laboratory, IBM Canada Ltd., 895 Don Mills Road, North York, Ontario M3C 1W3, Canada (electronic mail: jtroster@vnet.ibm.com).* Mr. Troster is a software engineering process analyst in the Software Engineering Process Group of the IBM Toronto Software Solutions Laboratory. Mr. Troster obtained his Bachelor of Applied Sciences degree in electrical engineering in 1972 and his Master of Applied Sciences degree in biomedical engineering in 1975, both from the University of Toronto. His 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 V8W 3P6, Canada (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.

Reprint Order No. G321-5552.