Reverse Engineering: A Roadmap

Hausi A. Müller
Dept. of Computer Science
University of Victoria, Canada
hausi@csr.uvic.ca

Jens H. Jahnke
Dept. of Computer Science
University of Victoria, Canada
jens@csr.uvic.ca

Dennis B. Smith
Software Engineering Institute
Carnegie Mellon University, USA
dbs@sei.cmu.edu

Margaret-Anne Storey
Dept. of Computer Science
University of Victoria, Canada
mstorey@csr.uvic.ca

Scott R. Tilley
Dept. of Computer Science
University of California,
Riverside, USA
stilley@cs.ucr.edu

Kenny Wong
Dept. of Computing Science
University of Alberta, Canada
kenw@cs.ualberta.ca

ABSTRACT
By the early 1990s the need for reengineering legacy systems was already acute, but recently the demand has increased significantly with the shift toward web-based user interfaces. The demand by all business sectors to adapt their information systems to the Web has created a tremendous need for methods, tools, and infrastructures to evolve and exploit existing applications efficiently and cost-effectively. Reverse engineering has been heralded as one of the most promising technologies to combat this legacy systems problem.

This paper presents a roadmap for reverse engineering research for the first decade of the new millennium, building on the program comprehension theories of the 1980s and the reverse engineering technology of the 1990s.

Keywords
Software engineering, reverse engineering, data reverse engineering, program understanding, program comprehension, software analysis, software evolution, software maintenance, software reengineering, software migration, software tools, tool adoption, tool evaluation.

1 INTRODUCTION

The notion of computers automatically finding useful information is an exciting and promising aspect of just about any application intended to be of practical use [55]. A decade ago, following up on the successes of the early CASE tools, Chikofsky and Cross introduced a taxonomy for reverse engineering and design recovery [20]. They defined reverse engineering to be "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information."

Over the past ten years, researchers have produced a number of capabilities to explore, manipulate, analyze, summarize, hyperlink, synthesize, componentize, and visualize software artifacts. These capabilities include documentation in many forms and intermediate representations for code, data, and architecture. Many reverse engineering tools focus on extracting the structure of a legacy system with the goal of transferring this information into the minds of the software engineers trying to reengineer or reuse it. In corporate settings, reverse engineering tools still have a long way to go before becoming an effective and integral part of the standard toolset that a typical software engineer uses day-to-day.

The vitality of the field has been demonstrated by three annual conferences that helped to spark interest in the field and shape its ideas and focus: the *Working Conference on Reverse Engineering (WCRE)*, the *International Workshop on Program Comprehension (IWPC)*, and the *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*.

This paper presents a roadmap for reverse engineering research for the first decade of the new millennium, building on the program comprehension theories of the 1980s and the reverse engineering technology of the 1990s. We describe selected research agendas for code and data reverse engineering, as well as research strategies for tool development and evaluation. Investing in program understanding technology is critical for the software and information technology industry to control the inherent high costs and risks of legacy system evolution. Reverse engineering is a truly exciting field of research that is ready to be taught in computer science, computer engineering, and software engineering curricula [68].

In summarizing the major research trends, accomplishments, and unanswered needs, this paper is divided into four major parts. Section 2 concentrates on code reverse engineering, which has been the main focus of attention in this field over the past decade. In contrast, data reverse engineering, the topic of Section 3, is not as well established, but is expected to gain prominence in the new millennium. Section 4 explores the spectrum of reverse engineering tools. Section 5 deals with the question of why software reverse engineering tools are not more widely used, and Section 6 concludes the

paper.

2 CODE REVERSE ENGINEERING

In current research and practice, the focus of both forward and reverse engineering is at the code level. Forward engineering processes are geared toward producing quality code. The importance of the code level is underscored in legacy systems where important business rules are actually buried in the code [86]. During the evolution of software, change is applied to the source code, to add function, fix defects, and enhance quality. In systems with poor documentation, the code is the only reliable source of information about the system. As a result, the process of reverse engineering has focused on understanding the code.

Over the past ten years, reverse engineering research has produced a number of capabilities for analyzing code, including subsystem decomposition [13, 86], concept synthesis [8], design, program and change pattern matching [16, 31, 59, 76], program slicing and dicing [89], analysis of static and dynamic dependencies [80], object-oriented metrics [19], and software exploration and visualization [65]. In general, these analyses have been successful at treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps.

However, the code does not contain all the information that is needed. Typically, knowledge about architecture and design tradeoffs, engineering constraints, and the application domain only exists in the minds of the software engineers [3]. Over time, memories fade, people leave, documents decay, and complexity increases [46]. Consequently, an understanding gap arises between known, useful information and the required information needed to enable software change. At some point, the gap may become too wide to be easily spanned by the syntactic, semantic, and dynamic analyses provided by traditional programming tools.

Thus when we focus only at the low levels of abstraction, we miss the big picture behind the evolution of a software system [42]. There is a need to focus future research on the more significant levels of the business processes and the software architecture. For example, knowledge of software architecture from multiple user perspectives is needed to make large-scale, structural changes [91], and the capability to perform architecture reconstruction is becoming increasingly important [3]. Developers need information about the impacts of potential changes. Managers need information to assign and coordinate their personnel. If the information to create this knowledge can be maintained continuously, we could generate the required perspectives on a continuous basis without costly reverse engineering efforts.

Because such analyses are rarely performed today, current system evolution efforts often experience a time of crisis at which the gap between desired information and available information becomes critical. At that point reverse engineering techniques are inserted in a "big bang" attempt to regain useful understanding and insight. The structural, functional, and behavioral code analyses [58], however, require intensive human input to construct from scratch. These analyses are difficult to interpret, and are costly efforts with high risk.

Continuous Program Understanding

To avoid a crisis, it is important to address information needs more effectively throughout the software lifecycle. We need to better support the forward and backward traceability of software artifacts. For example, in the forward direction, given a design module, it is important to be able to obtain the code elements that implement it. In the backward direction, given a source or object file, we need to be able to obtain the business rule to which it contributes. In addition it is important to determine when it is most appropriate to focus the analysis at different levels of abstraction [7, 43].

For understanding purposes, traceability is especially important. We need to be able to take a pattern of change, such as updating a tax law, and map this law explicitly into software structures. Part of program comprehension is to reconstruct mappings between the application and implementation domains [14]. Thus, to ease long-term understanding, these mappings must be made explicit, recorded, reused, and updated continuously. The vision is that reverse engineering would be applied incrementally, in small loops with forward engineering, rather than as a desperate attempt at resurrecting a poorly understood system.

Several research issues, formulated as questions, need to be addressed to enable this capability for "continuous program understanding" [90].

- What are the long-term information needs of a software system?
- What patterns of change do software systems undergo?
- What mappings need to be explicitly recorded?
- What kind of software repository could represent the required information?
- What are the requirements of tool support to produce and manipulate the mappings?
- How can this support coexist with traditional, code-dominated tools, users, and processes?

Reverse Engineering Process

In addition to an emphasis on "continuous program understanding," it is important to focus efforts on a better definition of the reverse engineering process. Reverse engineering has typically been performed in an ad hoc manner. To address the technical issues effectively, the process must become more mature and repeatable, and more of its elements need to be supported by automated tools.

For example, a developer might require the software components that contribute to a specific system responsibility. The subsystem view to present this information should not require tedious manual manipulation. Instead, the mapping between responsibility and components should be consulted

and a script would then generate the required view, with the option for minor, personal customization by the user.

Such a script is an instance of a reverse engineering pattern [90], a commonly used task or solution to produce understanding in a particular situation. By cataloging such patterns and automating them through tool support, we would improve the maturity of the reverse engineering process. Thus, the insights of the SEI Capability Maturity Model® (CMM®) framework [36, 37] ought to apply to reverse engineering as well as forward engineering. Future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized.

Increased process maturity would enable better assessment of the risks, costs, and economics of reengineering activities. With poorly understood processes, the success of a reengineering project rests solely on the ingenuity of the people involved—ingenuity that disappears when the project ends. For evolving large software systems over long periods of time, an appreciation of both product and process improvement is needed.

Research Direction

In summary, for future research in reverse engineering, it is important to understand software at various levels of abstraction and maintain mappings between these levels. Catalogs of information, tool, and process requirements are needed as a prelude to enabling continuous program understanding. Useful reverse engineering processes need to be identified and better supported, as an important step to make the discipline of reengineering more rational. Reverse engineering tools and processes need to evolve with the development environment that stresses components, the Web, and distributed systems [6].

3 DATA REVERSE ENGINEERING

Most software systems for business and industry are information systems, that is, they maintain and process vast amounts of persistent business data. While the main focus of code reverse engineering is on improving human understanding about how this information is processed, data reverse engineering tackles the question of what information is stored and how this information can be used in a different context.

Research in data reverse engineering has been underrepresented in the software reverse engineering arena for two main reasons. First, there is a traditional partition between the database systems and software engineering communities. Second, code reverse engineering appears at first sight to be more challenging and interesting than data reverse engineering for academic researchers.

Recently, data reverse engineering concepts and techniques have gained increasing attention in the reverse engineering arena. This has been driven by requirements for data-oriented mass software changes resulting from needs such as the Y2K problem, the European currency conversion, or the migration of information systems to the Web and towards electronic commerce.

Researchers now recognize that the quality of a legacy system's recovered data documentation can make or break strategic information technology goals. For example data analysis is crucial in identifying the central business objects needed for migrating software systems to object-oriented platforms. A negative example can be seen from the fact that difficulties in comprehending the data structure of legacy systems have been cited as barriers in replacing legacy software with modern business solutions (e.g., SAP, Baan, or PEOPLESOFT [22]).

The increased use of data warehouses and data mining techniques for strategic decision support systems [86] have also motivated an interest in data reverse engineering technology. Incorporating data from various legacy systems in data warehouses requires a consistent mapping of legacy data structures on a common business object model. Similar challenges also occur with the web-based integration of formerly autonomous legacy information systems into cooperative, net-centric infrastructures.

Data reverse engineering techniques can also be used to assess the overall quality of software systems. An implemented persistent data structure with significant design flaws indicates a poorly implemented software system. An analysis of the data structures can help companies make decisions on whether to purchase (and maintain) commercial-off-the-shelf software packages. Data reverse engineering can also be used to assess the quality of the DBMS schema catalog of vendor software, and thus it can represent one of the evaluation criteria for a potential software product [10].

In general, reverse engineering the persistent data structure of software systems using a DBMS is more specifically referred to as database reverse engineering. Since most DBMSs provide the functionality to extract initial information about the implemented physical data structure, database reverse engineering has a higher potential for automation than data reverse engineering [1]. Consequently, most existing reverse engineering tools in this area consider information systems that employ a database platform. Many of these approaches are specifically targeted to relational systems [4, 26, 33, 40, 51, 64, 70].

Data Reverse Engineering Process and the Role of Tools

Figure 1 shows that the data (base) reverse engineering process consists of two major activities, referred to as analysis and abstraction, respectively.

Data Analysis

The analysis activity aims to recover an up-to-date logical data model that is structurally complete and semantically annotated. In most cases, important information about the data model is missing in the physical schema catalog extracted from the DBMS. However, indicators for structural and se-

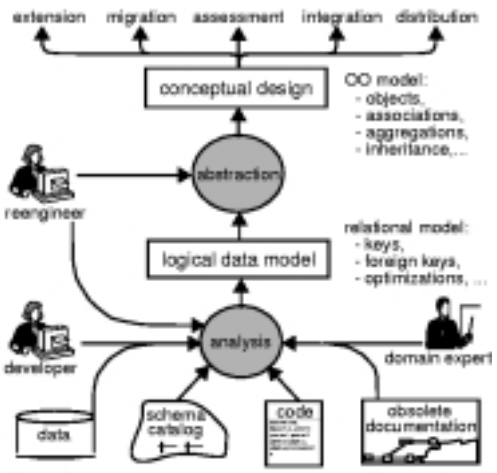

Figure 1: Data reverse engineering process

mantic schema constraints can be found in various parts of the legacy information system, including its data, procedural code, and documentation. Developers, users, and domain experts can often contribute valuable knowledge. In general, data analysis is an exploratory and human-intensive activity that requires a significant amount of experience and skills. Current tools provide only minimal support in this activity beyond visualizing the structure of an extracted schema catalog.

Even though it is unlikely that the cognitive task of data analysis can ever be fully automated, computer-aided reverse engineering tools have the potential to dramatically reduce the effort spent in this phase. They could be a major aid in searching, collecting, and combining indicators for structural and semantic schema constraints and guiding the reengineer from an initially incomplete data model to a complete and consistent result. However, to achieve this kind of support, current data reverse engineering tools need to overcome the following two significant problems:

- **Imperfect knowledge.** Data analysis inherently deals with uncertain assumptions and heuristics about legacy data models [39]. Combining detected semantic indicators (e.g., stereotypical code patterns or instances of hypothetical naming conventions in the schema catalog) often leads to uncertain and/or contradicting analysis results. Data reverse engineering tools have to tolerate imperfect knowledge to support this interactive process and to incrementally guide the reengineer to a consistent data model.

- **Customizability.** Legacy information systems are based on many different hardware and software platforms and programming languages. Their data models have been developed using various design conventions and idiosyncratic optimization patterns [11]. Most existing tools do not provide the necessary customizability to be applicable to this variety of application contexts. Some approaches address this problem by providing mechanisms for end-user programming with scripting languages [33]. In principle such tools provide a high amount of flexibility. However, coding analysis operations and heuristics with scripting languages often require significant skills and experience. To address this problem, a number of dedicated, more abstract formalisms have been proposed to specify and customize reverse engineering processes [40, 70]. Due to their high level of abstraction these approaches facilitate the customization process. However, they do not provide the same amount of flexibility as scripting languages. Consequently, a hybrid solution that combines high-level (e.g., rule-based) formalisms with low-level (e.g., programming scripts)is a fruitful area for exploration.

Conceptual Abstraction

Conceptual abstraction aims to map the logical data model derived from data analysis to an equivalent conceptual design. This design is usually represented by an entity-relationship or object-oriented model and provides the necessary level of abstraction required by most subsequent reengineering activities (cf. Figure 1). Currently, several tools support data abstraction. However, in practice, most of them are of limited use because they fail to fulfill at least one of the following two requirements:

- **Iteration.** The data reverse engineering process involves a sequence of analysis and abstraction activities with several cycles of iteration. After an initial analysis phase, the reengineer produces an initial abstract design that serves as the basis for discussion with domain experts and further investigations. This first abstract design needs to be altered as new knowledge about the legacy system becomes available. Although iteration is not well supported by current tools, an incremental change propagation mechanism is presented by Jahnke and Wadsack [41].

- **Bidirectional mapping process.** Current data reverse engineering tools follow a strictly bottom-up data abstraction process, that is, the abstraction is produced through a transformation of the analyzed logical data model. This approach is less adequate if a pre-existing partial design for the data structure is available from documentation or the knowledge of domain experts or developers. Using such information efficiently in reverse engineering legacy information systems would require a hybrid bottom-up/top-down abstraction process. Furthermore, such a process is required when more than one legacy data structure has to be mapped to a common

abstract data model (e.g., when several information systems are federated or integrated with a data warehouse).

Research Direction

Based on this discussion, the reverse engineering community needs to develop tools that provide more adequate support for human reasoning in an incremental and evolutionary reverse engineering process that can be customized to different application contexts.

4 REVERSE ENGINEERING TOOLS

Techniques used to aid program understanding can be grouped into three categories: *unaided browsing*, *leveraging corporate knowledge and experience*, and *computer-aided techniques like reverse engineering* [83].

Unaided browsing is essentially "humanware": the software engineer manually flips through source code in printed form or browses it online, perhaps using the file system as a navigation aid. This approach has inherent limitations based on the amount of information that a software engineer may be able to keep track of in his or her head.

Leveraging corporate knowledge and experience can be accomplished through mentoring or by conducting informal interviews with personnel knowledgeable about the subject system. This approach can be very valuable if there are people available who have been associated with the system as it has evolved over time. They carry important information in their heads about design decisions, major changes over time, and troublesome subsystems.

For example, corporate memory may be able to provide guidance on where to look when carrying out a new maintenance activity if it is similar to another change that took place in the past. This approach is useful both for gaining a big- picture understanding of the system and for learning about selected subsystems in detail.

However, leveraging corporate knowledge and experience is not always possible. The original designers may have left the company. The software system may have been acquired from another company. Or the system may have had its maintenance out-sourced. In these situations, computer-aided reverse engineering is necessary. A reverse-engineering environment can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level artifacts, such as source code. This frees software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern matching by inspection.

Current Tool Effectiveness

Given that reverse engineering tools seem to be a key to aiding program understanding, how effective are today's offerings in meeting this goal? In both academic and corporate settings, reverse engineering tools have a long way to go before becoming an effective and integral part of the standard toolset a typical software engineer calls upon in day-to-day usage [82]. Perhaps the biggest challenge to increased effectiveness of reverse engineering tools is wider adoption: tools can't be effective if they aren't used, and most software engineers have little knowledge of current tools and their capabilities. While there is a relatively healthy market for unit-testing tools, code debugging utilities, and integrated development environments, the market for reverse engineering tools remains quite limited.

In addition to awareness, adoption represents a critical barrier. Most people lack the necessary skills needed to make proper use of reverse engineering tools. The root of the adoption problem is really two-fold: a lack of software analysis skills on the part of today's software engineers, and a lack of integration between advanced reverse engineering tools and more commonplace software utilities such as those mentioned above. The art of program understanding requires knowledge of program analysis techniques that are essentially tool-independent. Since most programmers lack this type of foundational knowledge, even the best of tools won't be of much help.

From an integration perspective, most reverse engineering tools attempt to create a completely integrated environment in which the reverse engineering tool assumes it has overall control. However, such an approach precludes the easy integration of reverse engineering tools into toolsets commonly used in both academic research and in industry. In a UNIX-like environment, the established troika of edit/compile/debug tools are common [34]. Representative tools in this group include *emacs* and *vi* for editing, *gcc* for compiling, and *gdb* for debugging. In a Windows NT environment, the tools may have different names, but they serve similar purposes. The only real difference is cost and choice. A recent case study [84] illustrates the challenges facing students in a short-term project and the difficulties they face in solving the problem. Learning how to effectively use a reverse engineering tool is low on their list of priorities, even when such a tool is available.

In a corporate setting, the situation is not so very different. A relatively short project often means little time to learn new tools. The tools used in a commercial software development firm may be slightly richer than those in the academic setting. However, displacing an existing tool with a new tool—even if that tool is arguably better—is an extremely difficult task.

What Can Be Done

To address the challenges of reverse engineering tool effectiveness, there are several possible avenues to explore. These candidate solutions should address the two primary issues identified above: awareness and adoption. First, computer science and software engineering curriculums can encourage greater use of reverse engineering tools. They can carefully balance code synthesis (which is commonly taught) with program analysis (which is rarely taught). By learning the analy-

sis techniques used in the art of program understanding, students would be in a better position to leverage the capabilities of reverse engineering tools that can automate many of the analysis tasks.

To increase the adoption rate of reverse engineering tools, vendors need to address several issues. The tools need to be better integrated with common development environments on the popular platforms. They also need to be easier to use. A lengthy training period is a strong disincentive to tool adoption.

An issue related to both integration and ease-of-use is "good enough" or "just in time" understanding. If one watches how a software engineer uses other tools, they rarely exercise all of the tool's functionality. Indeed, the 80/20 rule seems to apply: 80% of the time they use less than 20% of the tool's capabilities. If the critical capabilities that constitute the 20% of commonly used functions were identified, vendors might be better able to integrate at least this level of support into other vendors' environments. For example, the use of simple tools such as *grep* to look for patterns in source code is inefficient. These inefficiencies are the result of inexactness of regular expressions versus programming language syntax and semantics, as well as the large number of false positive matches. Yet *grep* is still widely used because of cost, availability and ease of use. Perhaps simply augmenting *grep* with more context-dependent or domain-aware capabilities would be a better approach than a full-fledged search engine, with a new pattern language, a proprietary repository, and tangential capabilities.

5 EVALUATING REVERSE ENGINEERING TOOLS

This paper includes many references to tools and techniques to support reverse engineering. But an important consideration when choosing a path through these technologies, is how to measure the success of the tools or theories that may be selected. Many reverse engineering tools concentrate on extracting the structure or architecture of a legacy system with the goal of transferring this information into the minds of the software engineers trying to maintain or reuse it. That is, the tool's purpose is to increase the understanding that software engineers or/and managers have of the system being reverse engineered. But, since there is no agreed-upon definition or test of understanding [21], it is difficult to claim that program comprehension has been improved when program comprehension itself cannot be measured.

Despite such difficulty, it is generally agreed that more effective tools could reduce the amount of time that maintainers need to spend understanding software or that these tools could improve the quality of the programs that are being maintained. Coarse-grained analyses of these types of results can be attempted. There are several investigative techniques and empirical studies that may be appropriate for studying the benefits of reverse engineering tools [62]. These include:

- expert reviews,
- user studies,
- field observations,
- case studies, and
- surveys.

In general, there has been a lack of evaluation of reverse engineering tools [47], but there are some examples where the investigative techniques listed above have been used for evaluating tools. In this section, we describe these techniques and give examples of when these techniques have been applied to the evaluation of reverse engineering tools.

Expert reviews

Expert reviews are a set of informal investigative techniques that are very effective for evaluating tools in the area of human-computer interaction [69]. One of these techniques, heuristic evaluation, involves a set of expert reviewers critiquing the interface using a short list of design criteria [57]. Cognitive walkthroughs, another expert review technique, involve experts simulating users walking through the interface to carry out typical tasks.

Expert reviews can be applied at any stage in the tool's design life cycle, and are normally not as expensive or as time-consuming as more formal methods. For example, a reverse engineering tool developer could use the Technology Delta Framework developed by Brown and Wallnau [15] to do an introspective evaluation of their own tool in the early stages of development. This framework supports technology evaluation in two ways: understanding how the technology differs from other technologies and then considering how these differences will support the users' needs. This type of evaluation is very useful but is often overlooked for sophisticated research tools such as reverse engineering tools.

User studies

User studies are formal experiments where key factors (the independent variables) are identified and manipulated to measure their effects on other factors (the dependent variables). Experiments can be conducted either in a laboratory or in the field. In a laboratory setting, there is more control over the independent variables in the experiment. However, other factors are introduced which may not be applicable in more realistic situations. For example, students are often used to act as subjects, but students probably do not comprehend programs in the same way that industrial programmers do [73]. Fenton and Pfleeger refer to formal experiments as research in the small [27]. User studies are more appropriate for fine-grained analyses of software engineering activities or processes.

In general, there have been relatively few formal experiments to evaluate reverse engineering tools. However there are a few exceptions, most notably [12, 49, 78, 79].

Field observations

Formal user studies in the field can be more difficult to execute than those in a laboratory setting, because they tend to

be more expensive and time consuming. However, informal user studies where one or two programmers are observed in their natural setting can be very insightful. Often a researcher will only have the opportunity to observe one or two programmers. Although the observation may be intrusive on the programmers, this technique gives the researcher the opportunity to observe maintainers using tools in more realistic settings. However, the results from field observations may also be difficult to generalize because of the small number of subjects normally involved.

Von Mayrhauser and Vans observed programmers in an industrial setting performing a variety of maintenance activities [87]. The goal of their study was to validate their integrated code comprehension model. They derived reverse engineering tool capabilities from an analysis of audio-taped, think-aloud reports of the programmers' information needs during maintenance activities.

Singer and Lethbridge describe a field experiment to study the work practices of software engineers working at a large telecommunications company [73]. They combined various investigative techniques to gather information on software engineers' work practices, such as questionnaires issued on the Web, longitudinal observations of several software engineers, and company wide tool usage statistics. They used the results from their studies to motivate the design of a software exploration tool called TkSEE (Software Exploration Environment) [73].

Case studies
Case studies occur when a particular tool is applied to a specific system, and the experimenter, often introspectively, documents the activities involved. Case studies are particularly useful when the experimenter has very little control over the factors to be studied. Expert reviews can be combined with specific case studies as a more powerful evaluation technique.

Bellay and Gall report an evaluation of four reverse engineering tools that analyze C source code [5]: Refine/C [85], Imagix 4D [38], SNiFF+ [74], and Rigi [53]. They investigated the capabilities of these tools by applying them to a real-world embedded software system which implements part of a train control system. They used a number of assessment criteria derived from Brown and Wallnau's Technology Delta Framework [15]. The main focus of their case study was on the tool capabilities to generate graphical reports such as call trees, control-flow graphs, and data-flow graphs [5]. They concluded that there is no single tool that is the 'best' as the four tools differ considerably in their respective functionalities.

Armstrong and Trudeau also evaluated several reverse engineering tools. They based their evaluation on the abilities of the tools to extract an architectural design from the source code of CLIPS (C-Language Interface processing System) and for browsing the Linux operating system [2]. The five tools they examined were: Rigi [53], the Dali workbench [42], the Software Bookshelf [28], CIA [18], and SNiFF+ [74]. Their investigations focused on the abstraction and visualization of system components and interactions.

Surveys
Surveys are normally used as a retrospective investigative technique. For example, surveys can ask questions of the nature: Did the use of tool A reduce the amount of time you had to spend doing maintenance changes? Although infrequently used in the field of psychology of programming, surveys can be useful as a form of exploratory research [9].

Cross *et al.* designed a preference survey to informally evaluate the GRASP software visualization tool [24]. GRASP uses a Control Structure Diagram (CSD), an algorithmic level graphical representation of the software. The CSD was compared to four other graphical diagrams [25].

Sim *et al.* conducted a survey using a web-based questionnaire to find archetypes (i.e., typical or standard examples) of source code searching by maintainers [71]. Their results found that the most commonly used tools for searching were (by increasing usage): editors, *grep*, *find*, and integrated development environments. Administering the questionnaire over the Web was found to be very effective for information gathering.

Summary
This section reviewed various experimental techniques for evaluating and comparing software exploration tools, an important category of reverse engineering tools. Each of the investigative techniques just described has certain advantages and disadvantages. However, combining these techniques (as Singer and Lethbridge have done [73]) should produce stronger results. Moreover, sharing results among research groups is also very important. For example, Sim and Storey chaired a workshop where several reverse engineering tools were compared in a live demonstration [72]. The tools were applied to a significant case study where each team had to complete a series of software maintenance and documentation tasks and collaboration between teams was emphasized.

Adoption of reverse engineering technology in industry has been very slow [90]. However, we observed in our user studies [78, 79] that usability is often a major concern. If the tool is difficult to use, it will affect its adoption rate, no matter how useful it may be.

6 CONCLUSIONS
The 1980s produced a solid foundation for our field with the *Laws of Software Evolution* [46], theories for the fundamental strategies of program comprehension [14, 48, 60], and a taxonomy for reverse engineering [20]. We also realized that fifty to ninety percent of evolution effort involves program understanding [75].

The 1990s began with a series of papers that outlined challenges and research directions for the decade [20, 35, 66, 67,

63, 88]. During that decade, the reverse engineering community developed infrastructures and tools for the three major components of a reverse engineering system: parsers, a repository, and a visualization engine. Researchers developed strategies for specific reengineering scenarios [13, 30, 32, 45], and as a result investigated program understanding technology for these scenarios using industrial-strength reverse engineering and transformation tools [17].

Even though the theory of parsing and its technology has been around since the 1960s, robust parsers for legacy languages and their dialects are still not readily available [56]. A notable exception is the IBM VisualAge C++ environment, which features an API to access the complete abstract syntax tree [50]. Fortunately, the urgency of the Year 2000 problem has made the availability of stand-alone parsers a top priority. But there is more research needed to produce parsing components that can be easily integrated with reverse engineering tools.

With the proliferation of object technology, the expectations were high during the early 1990s for a common object-oriented repository to store all the artifacts being accumulated during the evolution of a software system. The research community made great strides in modelling collections of software artifacts at various levels of abstraction using graphs and developing object-oriented schemas for these models, but in most cases the artifacts for multi million-line software systems were stored in relational databases and file systems.

The past decade produced many software exploration tools [12, 18, 23, 29, 42, 52, 53, 54, 61, 65, 73, 77]. We finally have enough desktop computing power to manipulate huge graphs of software artifacts effectively. Some software exploration tools are now built using web browsers to exploit the fact that the users intimately know these tools for exploring dependencies [29].

This paper presented four perspectives on the field of reverse engineering to provide a roadmap for the first decade of the new millennium. Researchers will continue to develop technology and tools for generic reverse engineering tasks, particularly for data reverse engineering (e.g., the recovery of logical and conceptual schemas), but future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized [90]. We need to integrate forward and reverse engineering processes for large evolving software systems and achieve the same appreciation for product and process improvement for long-term evolution as for the initial development phases [44].

The most promising direction in this area is the continuous program understanding approach [90]. The premise that software reverse engineering needs to be applied continuously throughout the lifetime of the software and that it is important to understand and potentially reconstruct the earliest design and architectural decisions [42] has major tool design implications. Tool integration and adoption should be central issues for the next decade. For the future, it is critical that we can effectively answer questions, such as "How much knowledge, at what level of abstraction, do we need to extract from a subject system, to make informed decisions about reengineering it?" Thus, we need to tailor and adapt the program understanding tasks to specific reengineering objectives.

We will never be able to predict all needs of the reverse engineers and, therefore, must develop tools that are end-user programmable [81]. Pervasive scripting is one successful strategy to allow the user to codify, customize, and automate continuous understanding activities and, at the same time, integrate the reverse engineering tools into his or her personal software development process and environment. Infrastructures for tool integration have evolved dramatically in recent years. We expect that control, data, and presentation integration technology will continue to advance at amazing rates. Finally, we need to evaluate reverse engineering tools and technology in industrial settings with concrete reengineering tasks at hand.

Even if we perfect reverse engineering technology, there are inherent high costs and risks in evolving legacy software systems. Developing strategies to control these costs and risks is a key research direction for the future. Practitioners need a reengineering economics book, which would serve as a guide to determine reengineering costs and to use economic analyses for making improved reengineering decisions.

Probably the most critical issue for the next decade is to teach students about software evolution. Computer science, computer engineering, and software engineering curricula, by and large, teach software construction from scratch and neglect to teach software maintenance and evolution. Contrast this situation with electrical or civil engineering, where the study of existing systems and architectures constitutes a major part of the curriculum. Concepts such as architecture, abstraction, consistency, completeness, efficiency, or robustness should be taught from both a software design and a software analysis perspective. Software architecture courses are now established in many computer science programs, but topics such as software evolution, reverse engineering, program understanding, software reengineering, or software migration are rare. We must aim for a balance between software analysis and software construction in software engineering curricula.

ACKNOWLEDGEMENTS

This research was supported in part by *NSERC*, the *National Sciences and Engineering Research Council of Canada*, by *CAS*, the *IBM Toronto Centre for Advanced Studies*, by *CSER*, the *Canadian Consortium for Software Engineering Research*, by *IRIS*, the *Institute for Robotics and Intelligent Systems Network of Centres for Excellence*, by *ASI*, the *British Columbia Advanced Systems Institute*, by the *Carnegie Mellon Software Engineering Institute*, and the *Universities of Alberta*, *Paderborn*, *Riverside*, and *Victoria*.

REFERENCES

[1] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[2] M. Armstrong and C. Trudeau. Evaluating architectural extractors. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 30–39, October 1998.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1997.

[4] A. Behm, A. Geppert, and K. R. Dittrich. On the migration of relational schemas and data to object-oriented database systems. In *Proceedings 5th International Conference on Re-Technologies for Information Systems*, Klagenfurt, Austria, pages 13–33. Österreichische Computer Gesellschaft, December 1997.

[5] B. Bellay and H. Gall. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance: Research and Practice*, 10:305–331, 1998.

[6] K. Bennett and V. Rajlich. Software maintenance and evolution: A roadmap. In *this volume*, June 2000.

[7] J. Bergey, D. Smith, N. Weiderman, and S. Woods. Options analysis for reengineering (OAR): Issues and conceptual approach. Technical Report CMU/SEI-99-TN-014, Carnegie Mellon Software Engineering Institute, 1999.

[8] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[9] A. Blackwell. Questionable practices: The use of questionnaire in psychology of programming research. *The Psychology of Programming Interest Group Newsletter*, 22, July 1998.

[10] M. Blaha. On reverse engineering of vendor databases. In *Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 183–190. IEEE Computer Society Press, October 1998.

[11] M. Blaha and W. Premerlani. Observed idiosyncracies of relational database designs. In *Second Working Conference on Reverse Engineering (WCRE-95)*, Toronto, Ontario, Canada. IEEE Computer Society Press, 1995.

[12] K. Brade, M. Guzdial, M. Steckel, and E. Soloway. Whorf: A visualization tool for software maintenance. In *Proceedings 1992 IEEE Workshop on Visual Languages*, Seattle, Washington, pages 148–154, September 1992.

[13] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kauffman, 1995.

[14] R. Brooks. Towards a theory of comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:86–98, 1983.

[15] A. Brown and K. Wallnau. A framework for evaluating software technology. *IEEE Software*, pages 39–49, September 1996.

[16] B. Brown, X. Malveau, X. M. III, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

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

[18] Y.-F. Chen, M. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(1):325–334, March 1990.

[19] S. Chidamber and C. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions Software Engineering*, 20(6):476–493, 1994.

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

[21] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 69–78, October 1998.

[22] A. Clewett, D. Franklin, and A. McCown. *Network Resource Planning For SAP R/3, BAAN IV, and PEOPLESOFT: A Guide to Planning Enterprise Applications*. McGraw-Hill, 1998.

[23] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering (ICSE)*, Melbourne, Australia, pages 138–156. IEEE Computer Society Press, 1992.

[24] J. Cross II, T. Hendrix, L. Barowski, and K. Mathias. Scalable visualizations to support reverse engineering: A framework for evaluation. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 201–209, October 1998.

[25] J. Cross II, S. Maghsoodloo, and T. Hendrix. The control structure diagram: An initial evaluation. *Empirical Software Engineering*, 3(2):131–156, 1998.

[26] C. Fahrner and G. Vossen. Transforming relational database schemas into object-oriented schemas according to ODMG-93. In *Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases*, 1995.

[27] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[28] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.

[29] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[30] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[32] I. Graham. *Migrating to Object Technology*. Addison-Wesley, 1994.

[33] J.-L. Hainaut, J. Henrard, J.-M. Hick, and D. Roland. Database design recovery. *Lecture Notes in Computer Science*, 1080:272ff, 1996.

[34] W. Harrison, H. Ossher, and P. Tarr. Software engineering tools and environments: A roadmap. In *this volume*, June 2000.

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

[36] W. S. Humphrey. *Managing the Software Process*. Addison-Wesley, 1989.

[37] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[38] Imagix 4D. Imagix Corp. http://www.imagix.com.

[39] J. H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, Department of Mathematics and Computer Science, Universität Paderborn, Germany, September 1999.

[40] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *Proceedings of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer, September 1997.

[41] J. H. Jahnke and J. Wadsack. Integration of analysis and redesign activities in information system reengineering. In *Proceedings of the 3rd European Conference on Software Maintenance and Reengineering (CSMR-99)*, Amsterdam, The Netherlands, pages 160–168. IEEE CS, March 1999.

[42] R. Kazman and S. Carrie're. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.

[43] R. Kazman, S. Woods, and S. Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 154–163. IEEE Computer Society Press, October 1998.

[44] U. Kölsch. *Methodische Integration und Migration von Informationssystemen in objektorientierte Umgebungen*. PhD thesis, Forschungszentrum Informatik, Universität Karlsruhe, Germany, December 1999.

[45] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Müller, and J. Mylopoulos. Code migration through transformations: An experience report. In *Proceedings of CASCON-98*, Toronto Ontario, Canada, November 1998.

[46] M. Lehman. Programs, life cycles and laws of software evolution. *Proceedings of IEEE Special Issue on Software Engineering*, 68(9):1060–1076, September 1980.

[47] T. Lethbridge and J. Singer. Understanding software maintenance tools: Some empirical research. In *IEEE Workshop on Empirical Studies of Software Maintenance (WESS-97)*, Bari, Italy, pages 157–162, October 1997.

[48] S. Letovsky. *Cognitive Processes in Program Comprehension*, pages 58–79. Ablex Publishing Corporation, 1986.

[49] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula. Visualizing program dependencies: An experimental study. *Software–Practice and Experience*, 24(4):387–403, April 1994.

[50] J. Martin. Leveraging ibm visualage c++ for reverse engineering tasks. In *Proceedings of CASCON-99*, Toronto, Ontario, Canada, November 1999.

[51] P. Martin, J. R. Cordy, and R. Abu-Hamdeh. Information capacity preserving of relational schemas using structural transformation. Technical Report ISSN 0836-0227-95-392, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, November 1995.

[52] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software Concepts and Tools*, 16:170–182, 1995.

[53] H. Müller and K. Klashinsky. Rigi—A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE)*, Raffles City, Singapore, pages 80–86. IEEE Computer Society Press, April 1988.

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

[55] T. Munakata. Knowledge discovery. *Communications of the ACM*, 42(11):26–29, November 1999.

[56] G. Murphy, D. Notkin, and S. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, Germany, pages 90–100. IEEE Computer Society Press, March 1996.

[57] J. Nielsen. *Usability Engineering*. Academic Press, New York, 1994.

[58] J. Ning. *A Knowledge-based Approach to Automatic Program Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[59] S. Paul and a. Prakash. On formal query languages for source code search. *IEEE Transactions on Software Engineering*, SE-20(6):463–475, June 1994.

[60] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[61] P. Penny. *The Software Landscape: A Visual Formalism for Programming-in-the-Large*. PhD thesis, Department of Computer Science, University of Toronto, 1992.

[62] D. Perry, A. Porter, and J. L. Votta. Empirical studies: A roadmap. In *this volume*, June 2000.

[63] R. C. W. Peter G. Selfridge and E. J. Chikofsky. Challenges to the field of reverse engineering. In *Working Conference on Reverse Engineering (WCRE-93)*, Baltimore, Maryland, USA, pages 144–150, 1993.

[64] W. J. Premerlani and M. R. Blaha. An approach for reverse engineering of relational databases. *Communications of the ACM*, 37(5):42–49, May 1994.

[65] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

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

[67] S. Rugaber and S. Ornburn. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[68] M. Shaw. Software engineering education: A roadmap. In *this volume*, June 2000.

[69] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1998. Third Edition.

[70] O. Signore, M. Loffredo, M. Gregori, and M. Cima. Reconstruction of er schema from database applications: a cognitive approach. In *Proceedings of 13th International Conference of ERA*, Manchester, UK, pages 387–402. Springer, 1994.

[71] S. Sim, C. Clarke, and R. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98)*, Honolulu, Hawaii, USA, pages 180–187, October 1998.

[72] S. Sim and M.-A. D. Storey. A collective demonstration of program comprehension tools, a CASCON-99 workshop, November 1999. http://www.csr.uvic.ca/cascon99/.

[73] J. Singer and T. Lethbridge. Studying work practices to assist tool design in software engineering. In *Proceedings of the 6th International Workshop on Program Comprehension (WPC-98)*, Ischia, Italy, pages 173–179, June 1998.

[74] SNiFF+. User's Guide and Reference, Take-Five Software, version 2.3, December 1996. http://www.takefive.com.

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

[76] P. Stevens and R. Pooley. Systems reengineering patterns. In *ACM SIGSOFT Foundations of Software Engineering (FSE-98)*, Lake Buena Vista, Florida, USA, pages 17–23. ACM Press, 1998.

[77] M.-A. Storey and H. Müller. Manipulating and documenting software structure using shrimp views. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, Opio, France, pages 275–284. IEEE Computer Society Press, October 1998.

[78] M.-A. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. Müller. On designing an experiment to evaluate a reverse engineering tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE-96)*, Monterey, California, USA, pages 31–40, November 1996.

[79] M.-A. Storey, K. Wong, and H. Müller. How do program understanding tools affect how programmers understand programs. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE-97)*, Amsterdam, The Netherlands, pages 12–21, October 1997.

[80] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE-99)*, Atlanta, Georgia, USA, pages 304–313. IEEE Computer Society Press, October 1999.

[81] S. Tilley, K. Wong, M.-A. Storey, and H. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, December 1994.

[82] S. R. Tilley. Coming attractions in program understanding II: Highlights of 1997 and opportunities for 1998. Technical Report CMU/SEI-98-TR-001, Carnegie Mellon Software Engineering Institute, February 1998.

[83] S. R. Tilley. *The Canonical Activities of Reverse Engineering*. Baltzer Science Publishers, The Netherlands, February 2000.

[84] S. R. Tilley and S. Huang. Just enough understanding and not enough time. Technical report, Department of Computer Sciene, University of California Riverside, December 1999.

[85] J. Troster, J. Henshaw, and E. Buss. Filtering for quality. In the Proceedings of *CASCON-93*, Toronto, Ontario, Canada, pages 429–449, October 1993.

[86] A. Umar. *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. Prentice Hall, 1997.

[87] A. von Mayrhauser and A. Vans. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of CASE-93*, Singapore, pages 230–239, July 1993.

[88] R. C. Waters and E. J. Chikofsky. Reverse engineering—Introduction to the special section. *Communications of the ACM*, 37(5):22–25, May 1994.

[89] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[90] K. Wong. *Reverse Engineering Notebook*. PhD thesis, Department of Computer Science, University of Victoria, October 1999.

[91] K. Wong, S. Tilley, H. Müller, and M.-A. Storey. Structural redocumentation. *IEEE Software*, 12(1):46–54, January 1995.