Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities in 1998

Scott R. Tilley

February 1998

TECHNICAL REPORT
CMU/SEI-98-TR-001
ESC-TR-98-001

CMU/SEI-98-TR-001
ESC-TR-98-001

Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities in 1998

Scott R. Tilley

February 1998

Reengineering Center
Product Line Systems Program

Carnegie Mellon University
Software Engineering Institute

Pittsburgh, PA
15213-3890

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Jay Alonis, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Asset Source for Software Engineering Technology (ASSET): 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 or toll-free in the U.S. 1-800-547-8306 / FAX: (304) 284-9001 World Wide Web: http://www.asset.com / e-mail: sei@asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218 / Phone: (703) 767-8274 or toll-free in the U.S.: 1-800 225-3842.

Table of Contents

List of Figures

Acknowledgments

Thanks to Steve Woods of the Software Engineering Institute for providing some of the reference material on program understanding-based approaches to the Year 2000 problem discussed in Sections 4.3 and 5. Dennis Smith of the Software Engineering Institute and Steve Woods both made valuable comments on early drafts of this document.

Abstract

This report highlights important developments in program-understanding work in 1997 and outlines some of the opportunities for the field in 1998. A framework of three focus areas is used to categorize research and development activities in program understanding: investigating cognitive aspects, developing support mechanisms, and maturing the practice. Although significant progress was made in these areas, the rapid changes in the software engineering landscape are giving rise to several new challenges. Three of the most important in the coming year are leveraging the Web, black-box understanding, and the Year 2000 problem.

1. Introduction

Program understanding is a relatively immature field of study in which the terminology and focus are still evolving. The goal of program understanding is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner. The process of program understanding is ill defined and poorly understood. Yet it is an important aspect of successfully leveraging legacy systems, especially when parts of the system require internal restructuring at the algorithmic and data-structure level.

Many organizations are faced with maintaining aging software systems that are constructed to run on a variety of hardware types, are programmed in obsolete languages, and suffer from the disorganization that results from prolonged maintenance. As software ages, the task of maintaining it becomes more complex and more expensive. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding. Structuring mechanisms based on programming languages can create compatibility problems between the structure of the program and the structure of the users' mental model. In many cases, the only current, complete, and trustworthy information about a system is its source code; all other information must be derived from this. Consequently, software engineers must spend an inordinate amount of time creating representations of a system's high-level architecture from analyses of its low-level source code.

This report highlights some of the important themes that emerged from program-understanding work in 1997 and outlines some of the opportunities for the field in 1998. This report uses the same framework of three focus areas developed in a previous report to categorize research and development activities in program understanding: investigating cognitive aspects, developing support mechanisms, and maturing the practice [Tilley 96a]. The previous report identified some of the emerging technologies in program understanding, focusing on capabilities under development in 1996 that were believed to have the potential for significant benefit to advanced practitioners within five years. The benefits would accrue either by providing revolutionary technologies to software engineers that would enable them to fully or partially automate previously manual tasks, or by enhancing their capabilities by improving the tools and techniques used to aid program understanding.

Significant progress was made in these areas in 1997, but the rapid changes in the software engineering landscape are giving rise to several new challenges. Hence, this report also provides insight into three of the biggest opportunities to the program-understanding community in 1998: leveraging the Web, black-box understanding, and the Year 2000

problem. Each of these areas represents a unique opportunity, as well as some significant challenges, to the field. There are certainly other areas of opportunity as well, and it is expected that the set may change somewhat next year when this report is updated.

The next section provides an overview of the three main program-understanding focus areas. Section 3 highlights the developments in the field from 1997. Section 4 provides a discussion of some of the opportunities for program understanding in 1998. Section 5 summarizes the report.

2. Program-Understanding Focus Areas

Focus areas in program understanding can be grouped into three broad categories: (1) investigating cognitive aspects, (2) developing support mechanisms, and (3) maturing the practice. This section briefly describes these problem areas and summarizes the topics in each category. Although the categorization of problem areas remains valid, not all of the topics were addressed by program-understanding work in 1997. Figure 1 shows a possible time line of coming attractions in program understanding circa late 1996. This figure, taken from the previous report, summarizes improved capabilities under three promising lines of emerging technologies.

2.1 Investigating Cognitive Aspects

This category identifies how humans apply problem-solving techniques when attempting to understand a program. The *cognitive aspect* of program understanding is the study of the problem-solving behavior of software engineers engaged in understanding tasks. Because the productivity of software engineers varies by more than an order of magnitude, the strategies of successful practitioners are of great interest in producing tools and techniques that better support program understanding. This analysis leads to the development of tools and techniques that better support program-understanding activities.

Most of this research has focused on comprehension strategies of software engineers. A comprehension strategy involves what information software engineers use to understand a software artifact and how they use that information. These studies cross the disciplines of software engineering, education, and cognitive science.

In the previous report, topics discussed were comprehension strategies, computer-supported cooperative understanding, and scenario-driven maintenance handbooks. It was felt that no significant work would take place in the latter two topics, and that remains essentially correct. Some work was performed on comprehension strategies; that is discussed in Section 3.1.

2.2 Developing Support Mechanisms

This category identifies how tools can be used to aid comprehension. There are a variety of *support mechanisms* for aiding program understanding. They can be grouped into three categories: unaided browsing, leveraging corporate knowledge and experience, and computer-aided techniques like reverse engineering.

Emerging Technology	Early Availability				
	1996	1997	1998	1999	2000
1 Investigating cognitive aspects					
Comprehension strategies	x				
Computer-Supported Coop. Understanding			x		
Maintenance handbooks			x		
2 Developing support mechanisms					
Data gathering					
Leveraging mature technology		x			
Alternative sources of data		x			
Data filtering	x				
Knowledge management					
Advanced modeling techniques				x	
Iterative domain modeling					x
Scaleable knowledge bases			x		
Information exploration					
Navigation					
Reduced disorientation		x			
WWW-based interfaces	x				
Advanced pattern matching				x	
Analysis					
End user programmability	x				
Automation levels		x			
Higher-order impact analysis				x	
Presentation					
Advanced visualization techniques			x		
Tailorable user interfaces		x			
Multimedia			x		
3 Maturing the practice					
Technology insertion			x		
Empirical studies		x			
Common terminology			x		

Figure 1: Coming Attractions Time Line for 1996-2000

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. *Leveraging corporate knowledge and experience* can be done through mentoring or by conducting informal interviews with personnel knowledgeable about the subject system. This approach is useful both for gaining a big-picture understanding of the system and for learning about selected subsystems in detail. Unfortunately, this type of corporate knowledge and experience is not always available: 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 this situation, the only recourse is the third category of support mechanisms: *computer-aided tools and techniques*. Such support mechanisms can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level code. These support mechanisms free software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern-matching by inspection. Reverse engineering remains the predominant support mechanism used for program understanding. In the previous report, the following advances in the canonical activities of reverse engineering were discussed: alternative data-gathering techniques, advanced schemes for managing knowledge, and hypertext-based information exploration.

2.3 Maturing the Practice

This category identifies how emerging technology can mature so that it becomes part of the state –of –the practice. As the focus shifts from program understanding to system understanding, from software maintenance to system evolution and migration, and from bottom-up techniques to top-down techniques, the prospects for widespread adoption will improve. One of the impediments to greater adoption of program-understanding technology has been that this technology has been focused on sample programs that do not represent real-world legacy systems. Fortunately, this has begun to change, especially in light of the Year 2000 (Y2K) problem (see Section 4.3 for more discussion of the Y2K problem and its relationship to program-understanding work). System understanding must be integrated with the other migration technologies such as distributed object technology and net-centric computing [Weiderman 97].

As the state of the art *matures* and becomes the state of the *practice*, promising work is focusing on easing the insertion of program-understanding technology, on performing empirical studies, and on creating a common lexicon of related terms. Practitioners will benefit because program-understanding tools and techniques will be easier to use, more applicable to everyday tasks, and more widely perceived as a legitimate technology capable of solving real-world maintenance problems.

In the previous report, topics discussed were making technology sufficiently robust so that it is applicable to real-world problems, performing empirical studies to test the effectiveness of support mechanisms, and defining a universally accepted lexicon of terms.

3. Highlights of 1997

This section summarizes some of the highlights of 1997 in program-understanding research and development. The framework of three focus areas described in Section 2 is used to structure the accomplishments. Where appropriate, areas that would benefit from increased focus are identified.

3.1 Investigating Cognitive Aspects

By far, the most important encouraging development in this area is the increasing focus on empirical studies. There is still a large gap between the actual needs of software engineers and the tools and techniques that are available to them. This is not necessarily related to the tools themselves, but to the lack of understanding of the processes that a real-world software engineer goes through during evolution activities.

Work has been performed by several researchers looking into the effectiveness of different visualization techniques as they apply to aiding program understanding. For example, the work by Storey *et al* describes a framework that identifies the cognitive design elements for software exploration that aid program understanding [Storey 97a]. This framework was developed in part through experiments that investigated the effects of different tool capabilities and user interfaces on how software engineers understand programs [Storey 96, Storey 97b]. Studies of this nature are essential if the community is to provide more effective tools to support the cognitive process.

Earlier work on using protocol analysis for examining experimental results was reported by von Mayrhauser *et al* in [Mayrhauser 96]. In 1997 they continued this work and presented preliminary results on the use of an encoding mechanism for protocol analysis data [Lang 97]. This work is very important to those working in the cognitive aspects focus area because it might allow them to share results of their empirical studies. This would be a step towards repeatability, an important criterion in the maturation of the field. This theme of sharing results is further discussed in Section 3.3.

3.2 Developing Support Mechanisms

Perhaps the most significant development in this area was the clear emergence of powerful commercial offerings. Investigations into one such tool, Software Emancipation's DISCOVER [Software 96], only served to crystallize this phenomenon [Tilley 97a]. The investigation was guided by a preliminary version of a framework for classifying program-understanding tools [Tilley 96b]. The framework is based on a description of the canonical

activities that are characteristic of the reverse engineering process: data gathering, knowledge management, and information exploration. A descriptive model provides a means to classify different approaches to reverse engineering through a common frame of reference. The framework also includes considerations of the quality attributes (nonfunctional issues) of the support mechanism, such as extensibility.

There are now several commercial products that could have been used for this study. Representative tools include the Reasoning Software Developer Kit[1] from Reasoning Inc. [Reasoning 98a], Sniff+ from TakeFive Software [TakeFive 98], and Visual Studio from Microsoft [Microsoft 97]. In the end, DISCOVER[2] was chosen for this independent study for several reasons. It is one of the most advanced commercial environments currently on the market that is representative of a new generation of tools that can be used to aid program understanding. Due to its pricing structure, it is unlikely that many researchers would have the opportunity to explore DISCOVER in an academic setting. As such, it provided an excellent opportunity to look at the state of the practice in program-understanding tools. It also provided an opportunity to exercise and evolve the program-understanding framework itself.

DISCOVER is a "Development Information System" for managing software applications [Software 96]. Central to its use and philosophy is a repository, called the Information Model (IM), which contains complete dependency information about the entire subject system. Creation of the IM is guided by a description of the mappings between the logical and physical structure of the subject system. It is initially populated by parsing the source code (DISCOVER supports both C and C++) and is incrementally updated from then on. There are five major application sets in DISCOVER that make use of the IM. Each application set is composed of a series of modules that address one or more development tasks. Of the five, DEVELOP/set is the one most geared towards program understanding.

Canonical activity support provided by DISCOVER is fairly extensive. Data-gathering capabilities are provided by the underlying Gnu compilation system. DISCOVER also uses information produced by other tools, such as Rational's Quantify [Rational 98], as data to support dynamic analysis and interpretive debugging. Data representing relationships not easily extractable from the source code, such as a semantic dependency between software and its associated documentation (or vice-versa), can be gathered from the user and represented as associations. The IM is relied on for knowledge management. It can be viewed as an associative, consolidated, persistent repository that keeps track of every artifact related to the software project, regardless of its size or apparent complexity, and every relation between artifacts. The IM can reside on either a designated or virtual server. It is essentially a distributed database that is used by all DISCOVER applications. The information exploration capabilities for navigation, analysis, and presentation are also functionally rich.

[1] The Software Developer Kit was formally known as The Software Refinery.
[2] Version 4.0 of DISCOVER was used in the study. This version was available in 1996.

Looking at the capabilities of DISCOVER through the lens of the program-understanding framework raised many issues about the maturity and viability of support mechanisms. The reason DISCOVER was described in such detail in this section is because in the past it was assumed that work coming out of research institutions was more powerful than anything offered by industry. However, this is no longer the case. While there is still important work taking place in primarily academic institutions, the landscape has changed sufficiently that commercial tools have overcome many of their previous limitations and in many cases bypassed research tools. This occurrence has significant implications for advanced practitioners, researchers, and tool developers.

For example, from the three areas of emergent technologies illustrated in Figure 1, DISCOVER currently provides at least partial support for at least five of them, only two of which were expected to appear in 1997. They are

1. Leveraging mature technology: Existing compilers are tailored for use in data gathering.
2. Data filtering: Several techniques for filtering information are provided.
3. Advanced modeling techniques: Groups and the IM provide a logical view of the subject system.
4. Scalable knowledge bases: The IM can handle upwards of one million lines of code.
5. Automation levels: The user can select manual, semi-automatic, or automatic operations in some instances.

It is clear that many of DISCOVER's advances are in the area of knowledge management, which is not surprising given its emphasis on the IM.

The implication for advanced practitioners is that advanced commercial environments like DISCOVER provide a production-quality amalgam of several recent research efforts into reverse-engineering environments. DISCOVER is scalable (able to process very large source files), robust (able to process industrial code), and relatively easy to use. The notable exception to the latter quality attribute (usability) is in the initial creation of the IM. This step requires considerable domain knowledge and intimate familiarity with the system environment in which DISCOVER is deployed. Model creation has proven to be somewhat challenging in a few instances where DISCOVER was introduced into a real-world environment.

The emergence of powerful commercial tools also has profound implications for researchers and tool developers working in program understanding. Research tools should "push the envelope" in some way, advancing the state of the art beyond the current state of the practice. This means that any new tools proposed by the research community must necessarily provide capabilities not found in commercial offerings. Since DISCOVER provides such a rich feature set, it has raised the bar of what should be considered *de facto* tools in a reverse-engineering environment.

However, there are still many gaps in these commercial tools. These gaps are opportunities: new avenues of exploration that would benefit from more attention by researchers. One of the most promising areas is information navigation and presentation. When DISCOVER is used on very large systems, the issue of scale becomes very important. Visualization techniques are needed to sort and filter information. Web-based interfaces would go a long way towards making the capabilities of tools like DISCOVER available to a wider group of users. (The concept of leveraging the Web in this fashion is discussed more fully in Section 4.1.) The application domain in which developers work is changing rapidly, from traditional monolithic programs to client-server to three-tiered net-centric applications that rely on distributed object technology. A tool such as DISCOVER can be used on this type of application, but much more work is needed in this area.

There is another issue that arose from this study that is applicable to all researchers: the yawning chasm of deployment. The issue of successfully deploying a commercial tool in an industrial setting is not new. It remains the final and critical obstacle to overcome for any technology to become successful. Researchers in the computer-aided software engineering (CASE) tool community have been struggling with the adoption issue for a long time. New technology is only successful if it easily integrates with existing tools and processes. Forcing users to adopt radically different ways of doing their jobs rarely succeeds. It is an unfortunate fact that many developers will give a tool only a short window of opportunity to succeed. If they cannot get the tool up and running in 10 minutes and see real results, without looking at the manual, they will often abandon the tool. For tools as complex as DISCOVER, this is a very real concern. It means the potential benefits of using the tool may not be realized. Perhaps one way of addressing this problem is by promoting the capabilities of such support mechanisms as services rather than tools. Another way may be to incorporate more automated support for initial setup and integration with the target environment. Automatic "wizards" such as those found in popular PC office software might be used to provide more hands-on guidance when domain experts are not available. The crux of the matter, from a researcher's point of view, is that no matter how powerful the tool or technique advocated, it will not be successful until it is merged into the normal activities of the users.

3.3 Maturing the Practice

The highlight of 1997 that has the potential for far-reaching positive consequences for maturing the practice of program understanding is the *Reverse Engineering Demonstration Project* [WorldPath 98]. This is an international cooperative study among commercial and non-commercial organizations active in program-understanding research to demonstrate the applicability of their tools and techniques in analyzing or evolving a common software example. The project is run under the auspices of WorldPath Information Services in conjunction with the IEEE Technical Committee on Software Engineering (TCSE) and the Reengineering Forum.

The project had its genesis at the 3rd Working Conference on Reverse Engineering in November 1996 when Elliot Chikofsky, director of the project, challenged the assembled

researchers and product developers "… to demonstrate the value of our tools and methods on a single, common software system. This will allow the industry to understand the relative merits of different software reengineering approaches. It will also let us learn how different methods and tools can complement each other."

This was exactly the type of concerted effort that the fractured program-understanding community needed to coalesce its work. If properly conducted, the project offers the opportunity for researchers to work together and share results. It also offers the opportunity to work closely with industrial partners, thereby focusing effort on real-world problems and providing invaluable feedback to researchers.

The project is analyzing a legacy software system called the WELTAB Election Tabulation System. It was initially developed in an extended version of FORTRAN for IBM mainframes in the late 1970s. In the 1980s it was ported to several computers running variants of IBM's VM/CMS operating system. In the 1980s it was converted to C and ran on PCs under MS-DOS. As with most software maintenance, each port caused significant modifications to the code. The current version is a combination of 190 files representing about 425KLOC (thousand lines of code) of C source code, batch commands, and data.

An initial briefing on the project's status was presented at the 4th Working Conference on Reverse Engineering in October 1997. Although progress has been slower than wished for, it is expected that 1998 will see significant effort spent by participants in meeting the project's goals. A summary presentation of the project and presentations by its participants will be held in March 1998 at the 6th Reengineering Forum conference in Florence, Italy, meeting jointly with the 2nd Euromicro Conference on Software Maintenance and Reengineering [REF 98].

4. Opportunities in 1998

Several themes are emerging in software engineering that present opportunities for the program-understanding community. New middleware is available that reduces the need for creating elaborate and proprietary environment infrastructure. By leveraging the capabilities of the Web, developers of support mechanisms can move beyond the traditional parsing and structure-charting activities to new solutions to current and future challenges. Tomorrow's component-based systems will require new approaches to understanding, approaches that are more focused on system-level interfaces than on program-level structure. The Y2K problem offers perhaps the biggest opportunity of all for program understanding to show its applicability to real-world problems and prove that it is a viable solution to industrial software evolution.

4.1 Leveraging the Web

Creating an infrastructure within which tools can be developed and deployed is one of the many challenges facing developers of program-understanding support mechanisms. The existing infrastructure for program-understanding research and development is inadequate; it is an expensive legacy system in its own right. Currently, most tools are point solutions that use custom-built software that is difficult to integrate with other tools, including those commercially available. This forces tool developers to redo basic infrastructure work, such as writing source-code parsers. It also forces users to adopt tools that use non-standard interfaces. The result is an infrastructure that has produced support mechanisms that have had minimal impact on the real world. As discussed in Section 3.2, several commercial tools are bypassing research efforts.

The Web offers one avenue out of the infrastructure problem. The Web continues to enjoy tremendous attention and growth. Many new and exciting developments are currently taking place in this arena. The Web is now being used for many more applications than just reading text-only documents. For some people, the Web browser interface is becoming the *de facto* standard interface for much of their computing.

There are many ways the community could exploit new Web developments [Tilley 97b]. One of the most important is to use next-generation HTML (hypertext markup language) technology, XML (extensible markup language) [W3C 98], as the basis for representing artifacts and relations of a software system. XML is more properly described as a meta-language because it is used to describe other languages. It is a strict subset of the meta-language SGML (standard generalized markup language), offering many of the same capabilities of SGML but without the complexity. XML has already been used for special

purposes, such as Microsoft's Channel Definition Format (CDF) used for pushing content in Internet Explorer 4. Users define new relations of interest in a portion of the Web document that contains the Document Type Definition (DTD), which consists of new "tags" that become available for use in the same manner as the tags that are predefined in HTML. For example, Figure 2 illustrates a very simple use of XML to represent the relationship between a source file and the functions that are defined in the file. Far more detailed modeling is possible using this technique.

By using XML, many Web tools can be reused for program-understanding purposes. There is general agreement that existing tools could benefit from a common intermediate form to represent artifacts of interest in software systems. There are several intermediate forms that already exist and could be considered candidates for reengineering purposes, (for example, the Rigi Standard Format [RSF] or Tuple-Attribute [TA] languages used in the Software Bookshelf [Finnigan 97]). Using XML as the basis for a common intermediate form is perhaps not an obvious choice. However, XML has the advantage of being an international standard that is already supported by several Web browsers and associated tools. It is expected to enjoy considerable attention and support by major players in the coming year.

```
<elementType id="Name"> <string/> </elementType>
<elementType id="Function"> <string/> </elementType>

<elementType id="File">
  <description>A file contains the definition of
functions.</description>
  <element type="#Name"/>
  <element type="#Function" occurs="ZEROORMORE"/>
</elementType>

<File>
  <Name>myfile</Name>
  <Function>foo</Function>
  <Function>bar</Function>
</File>
```

Figure 2: XML Example (Fragment)

Perhaps the biggest advantage of using XML as the basis for a common intermediate form is that it would immediately make all the existing tools currently available for the Web applicable for reengineering work as well. Finding artifacts of interest, visualizing the structure of information spaces, and extending the environment by integrating other tools would all be made possible essentially for free using tools already in existence. The use of XML would also mean that presentation integration could be achieved using what is fast becoming the ubiquitous user interface: Web browsers.

By using the Web as the primary infrastructure for program-understanding tools, the environment can leverage the integration mechanisms already in place by the primary Web browsers. Currently, most tools are forced to provide point solutions to integration problems. Frameworks for components and distributed objects have proven themselves capable in other areas of software engineering. By using XML representations of software system artifacts, program-understanding support mechanisms could also make use of the same technologies, such as search engines and structure-visualization applications.

4.2 Black-Box Understanding

Several enabling technologies are converging that will have a significant impact on the architecture, acquisition, and integration of software-intensive systems. These are components, distributed objects, and net-centric computing. The adoption of the Web as a ubiquitous net-centric computing platform, supported by distributed object technologies such as Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), and Java Beans, are some of the driving factors behind the emergence of component-based systems. Taken together, these technologies present many challenges to traditional program understanding. Yet it is crucial that the community begin to address these new technologies, for it is with such tools that new systems will be developed and today's legacy systems will be evolved.

Current approaches to aiding program understanding can be characterized as "white-box," relying primarily on static analysis of source code to extract internal structure and control flow information. This type of approach may not be successful when applied to component-based systems, especially when the components are off the shelf and when the system is distributed. Black-box systems typically do not come with source code, which precludes the static analysis that most white-box understanding techniques rely on, except those few that employ binary (object code) reverse engineering. The nature of program understanding should change its emphasis to "black-box" system understanding of the interfaces between software modules.

In many ways, understanding black-box systems is more like hardware debugging than software comprehension. Moreover, understanding is not needed just for maintenance activities, as is the case with traditional systems. Some degree of component-level program understanding is needed to qualify new components, to adapt them to specific project needs, and to integrate them into the whole system. New approaches such as binary reverse engineering, interface probing, and dynamic data analysis may be more useful in understanding the nature of such a system. In many situations, some combination of white-box understanding and black-box understanding will be required to evolve a system successfully.

4.3 The Year 2000 Problem

Of all the software maintenance tasks that have appeared in recent years, none has the potential for such a large and widespread impact as the Y2K problem. Because the task is relatively well defined, it is amenable to an automated solution (at least in part). If program understanding technology, in particular automated plan recognition, cannot be made viable in this circumstance it is doubtful it will ever be widely adopted for other software maintenance tasks.

The crux of the Y2K problem is simple [Smith 97]. Most installed legacy systems have routinely stored and processed dates without a century indicator. Date fields are typically stored in "MMDDYY" format, where "YY" represents the last two digits of a given year. This type of two-digit date encoding will probably fail to recognize "00" as representing the year 2000 and instead will erroneously interpret "00" as representing 1900, since the "19" has been implicitly assumed. There is an additional problem that, contrary to popular belief, 2000 is a leap year based on the rule that it is divisible by 400, and this rule is not commonly programmed into existing systems.

The date problem is crucial because dates are fundamental to the calculation of algorithms for virtually every type of electronic support system, such as command and control, assembly lines, interest and loan payments, invoices, airline reservations, and so on. Since computer systems affect almost every aspect of modern life, it is apparent that the electronic world that we take for granted could be severely crippled if the dates are not corrected.

There are three basic techniques to address the Y2K problem:

1. Expand the database to accommodate a century indicator.
2. Develop date "windows" that are processed differently depending on whether the date is early or late in a century.
3. Compress the date fields in the database.

Although the most complete solution is to expand the database, it is also the most expensive and time consuming. A number of factors may suggest alternative solutions including time, resources, future plans, estimated life span of the system, and relationship to other systems. 1998 may very well be the critical year for Y2K remediation. There is very little slack time available, and industry is reporting a severe shortage of qualified staff. Therefore, automated approaches that rely on program-understanding technology may be considered.

Fortunately, there are signs that this is in fact occurring, albeit in a non-obvious manner. Several tools have emerged in the past year that claim to deal with different aspects of the Y2K problem, such as system inventory, impact analysis, and testing. Most of these tools include some sort of "problem identification" module that uses an experientially compiled database of rules for locating faulty code and, in many cases, an accompanying set of rules for automatically changing the actual source itself. Such tools exploit well-known

technology, such as lexical pattern matching (`grep`-like facilities) and rule-based transformations. These tools are limited primarily by the existence of an adequate library of plans (or rules for recognition), by the size of the source program and subsequent abstract syntax trees, and by the efficiency of the algorithms for matching. Transformation tools are faced with even greater hurdles since false-positive matches can have disastrous consequences if code is changed incorrectly.

Few Y2K tools claim to make explicit use of research results in using plan-based techniques for concept recovery. But closer observation reveals that these recognizer-helper tools are primarily rule based, matching sets of components that describe typical Y2K exposures according to the inter-relationships of components including both data-flow and control-flow relationships. Interestingly, several commercial tools make explicit reference to their exploitation of constraint-based algorithms for controlling the cost of matching in these large source programs. See, for example, [Reasoning 98b] and [Xinotech 98].

Thus many Y2K tools are in fact leveraging the same underlying algorithms and technologies that developed in the program-understanding research community. Rule-based recognizers augmented with constraint-based reasoning systems are at the core of current technology. This can be viewed as a positive aspect of technology transition and should be more widely publicized.

Many commercial products such as Reasoning/2000 claim over 80% automatic remediation. The potential exists for even greater advances in automated problem recognition and remediation. Unfortunately, plans codified in commonly used forms, such as REFINE, are not publicly available. This is primarily for competitive reasons: vendors feel that representations of Y2K plans and code idioms are part of their advantage over competing products. This is unfortunate, especially in light of the discussion of the collaborative demonstration project in Section 3.3.

5. Summary

This report presented some of the highlights from 1997 in the field of program understanding. It also outlined some of the opportunities in 1998. A few of the developments predicted in a previous report on this subject have already come to fruition, while others have not yet received the attention they deserve.

The topics discussed in this report are meant to be representative of ongoing work in program understanding; it is by no means exhaustive. For example, extraction of architectural elements from legacy systems is likely to become an increasingly important activity. This is especially true for organizations seeking to leverage their legacy assets when migrating to a product-line approach. Fortunately, some preliminary work has already taken place in this area [Kazman 97].

One of the most important themes that emerged in 1997 is the importance of collaboration in the research community. Without such collaboration, it is felt that developments in the commercial world will quickly bypass research efforts, with the unfortunate side effect of widening the chasm between research and practice in this emergent field. Program-understanding tools developed in industry have several substantial advantages of existing research demonstrations. Many of these advantages stem from the existence of a viable infrastructure for tool development and deployment that industry enjoys but that research teams usually lack. Essential functionality, such as mature parsing technology, underlies all successful program-understanding tools. But research prototypes often lack these necessary elements because of a lack of resources available to develop them.

Commercial tools can also build upon a wealth of industrial application examples and experience, and consequently focus on the more tractable aspects of the problem that can be addressed quickly. Research projects, by their very nature, tend to go after the harder problems first. But this approach sometimes limits their effectiveness in the field, delaying successful deployment. For example, recent experimental work on the scalability of certain concept recovery algorithms [Woods 97] provided results that suggested that certain poorly written source code had a potential to create pattern-matching complexity problems. This same work observed that real-world code rarely contains such pathological structures. Instead of refining the effectiveness and applicability of those instances that seem relatively straightforward, much effort was spent on improving the effectiveness of the algorithms on

artificially bad generated source instances in order to prove the concept even in extremely unlikely code situations. Consequently, the opportunity to demonstrate the viability of this research work to the practitioner community remains unexplored. It is hoped that this situation will be addressed by all members of the program-understanding community in 1998.

References

[Finnigan 97] Finnigan, P.; Holt, R.; Kalas, I.; Kerr, S.; Kontogiannis, K.; Müller, H.; Mylopoulos, J.; Perelgut, S.; Stanley, M.; and Wong, K. "The Software Bookshelf." *IBM Systems Journal* 36, 4, (November 1997): 564-593.

[Kazman 97] Kazman, Rick and Carrière, S. Jeromy. *Playing Detective: Reconstructing Software Architecture from Available Evidence.* (CMU/SEI-97-TR-010, ADA 330928). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Lang 97] Lang, S. and von Mayrhauser, A. "Towards a Systematic Analysis of Program Comprehension Strategies for Legacy Software." Presented at *Workshop on Migration Strategies for Legacy Systems*, held in conjunction with *The 1997 International Conference on Software Engineering (ICSE-18)*. Boston, MA: May 17, 1997. IEEE Computer Society Press, 1997.

[Mayrhauser 96] von Mayrhauser, A. and Vans, A. "On the Role of Hypothesis During Opportunistic Understanding While Porting Large Scale Code," 68-77. *Proceedings of the 4th Workshop on Program Comprehension.* Berlin, Germany: March 29-31, 1996. IEEE Computer Society Press, 1996.

[Microsoft 97] Microsoft Corp. *Visual Studio* [online]. Available WWW <URL: http://www.microsoft.com/vstudio/> (1997).

[REF 98] The 1998 Reengineering Forum [online]. Available WWW <URL: http://www.rengineer.org/ref98/> (1998).

[Rational 98] Rational Software Corp. *Quantify* [online]. Available WWW <URL: http://www.rational.com/products/quantify/> (1998).

[Reasoning 98a] Reasoning, Inc. *The Reasoning Software Developer Kit* [online]. Available WWW <URL: http://www.reasoning.com/sdk.html> (1998).

[Reasoning 98b] Reasoning, Inc. *Reasoning/2000 for COBOL* [online]. Available WWW <URL: http://www.reasoning.com/r2000.html> (1998).

[Smith 97] Smith, Dennis B.; Müller, Hausi A.; and Tilley, Scott R. *The Year 2000 Problem: Issues and Implications* (CMU/SEI-97-TR-002, ADA 325361). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Software 96] Software Emancipation. *The DISCOVER Development Information System* (Version 4.0) [online]. Available WWW <URL: http://www.setech.com> (1996).

[Storey 96] Storey, Margaret-Anne D.; Wong, Kenny; Fong, P.; Hooper, D.; Hopkins, K.; and Müller, Hausi A. "On Designing an Experiment to Evaluate a Reverse Engineering Tool," 31-40. *Proceedings of the 3rd Working Conference on Reverse Engineering*. Monterey, CA: November 8-10, 1996. IEEE Computer Society Press, 1996.

[Storey 97a] Storey, Margaret-Anne D.; Fracchia, David; and Müller, Hausi A. "Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization," 17-28. *Proceedings of the 5th Workshop on Program Comprehension*. Dearborn, Michigan: May 28-30, 1997. IEEE Computer Society Press, 1997.

[Storey 97b] Storey, Margaret-Anne D.; Wong, Kenny; and Müller, Hausi A. "How Do Program Understanding Tools Affect How Programmers Understand Programs?" 12-21. *Proceedings of the 4th Working Conference on Reverse Engineering*. Amsterdam, The Netherlands: October 6-8, 1997. IEEE Computer Society Press, 1997.

[TakeFive 98] TakeFive Software. *Sniff+* [online]. Available WWW <URL: http://www.takefive.com/products.htm> (1998).

[Tilley 96a] Tilley, Scott R. and Smith, Dennis B. *Coming Attractions in Program Understanding* (CMU/SEI-96-TR-019, ADA 320731). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

[Tilley 96b] Tilley, Scott R.; Santanu, Paul; and Smith, Dennis B. "Toward a Framework for Program Understanding," 19-28. *Proceedings of the 4th Workshop on Program Comprehension.* Berlin, Germany: March 29-31, 1996. IEEE Computer Society Press, 1996.

[Tilley 97a] Tilley, Scott R. *Discovering DISCOVER* (CMU/SEI-97-TR-012, ADA 331014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Tilley 97b] Tilley, Scott R. and Smith, Dennis B. “On Using the Web as Infrastructure for Reengineering,” 170-173. *Proceedings of the 5th Workshop on Program Comprehension*. Dearborn, Michigan: May 28-30, 1997. IEEE Computer Society Press, 1997.

[W3C 98] The World Wide Web Consortium. *Extensible Markup Language (XML)* [online]. Available WWW <URL: http://www.w3.org/XML/> (1998).

[Weiderman 97] Weiderman, Nelson H.; Bergey, John K.; Smith, Dennis B.; and Tilley, Scott R. *Approaches to Legacy System Evolution*. (CMU/SEI-97-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997.

[Woods 97] Woods, Steve; Quilici, Alex; and Yang, Qiang. *Constraint-Based Design Recovery for Software Reengineering: Theory and Experiments*. Volume 3 of The Kluwer International Series in Software Engineering. Boston, Mass.: Kluwer Academic Publishing, December 1997.

[WorldPath 98] Worldpath Information Services. *Reverse Engineering Demonstration Project* [online]. Available WWW <URL: http://www.worldpath.com/reproject/> (1998).

[Xinotech 98] Xintotech Research. *Year 2000 Technology* [online]. Available WWW <URL: http://www.xinotech.com.y2k.html> (1998).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1998	Final

4. TITLE AND SUBTITLE

Coming Attractions in Program Understanding II: Highlights of 1997 and Opportunities in 1998

5. FUNDING NUMBERS

C — F19628-95-C-0003

6. AUTHOR(S)

Scott R. Tilley

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER

CMU/SEI-98-TR-001

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

ESC-TR-98-001

11. SUPPLEMENTARY NOTES

12.A DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited, DTIC, NTIS

12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This report highlights important developments in program-understanding work in 1997 and outlines some of the opportunities for the field in 1998. A framework of three focus areas is used to categorize research and development activities in program understanding: investigating cognitive aspects, developing support mechanisms, and maturing the practice. Although significant progress was made in these areas, the rapid changes in the software engineering landscape are giving rise to several new challenges. Three of the most important in the coming year are leveraging the Web, black-box understanding, and the Year 2000 problem.

14. SUBJECT TERMS cognitive aspects, component-based systems, distributed objects, maturing the practice, net-centric, program understanding, Web, Year 2000

15. NUMBER OF PAGES

25

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL