Panel On: Program Understanding—
Does It Offer Hope for Aging Software?

Moderator: Lawrence Miller
Computer Systems Research Department
The Aerospace Corporation
El Segundo, California 90245
lmiller@aero.org

Introduction

This panel looks at two questions: just what does it mean to understand a program; and what solutions to the "aging software crisis" does program understanding offer. When we look at the work in this area as exemplified by the panel members, we see a broad cut at program understanding.

There are several reasons why we might want to automate the process of program understanding: we could offer help and guidance to the programmer; we could assist in program testing, verification and debugging; we could provide long term assistance during maintenance, revisions and updates. We could also assist the programmer in understanding for himself or herself what a program is doing, and how it does it.

Program understanding means building a knowledge base that represents the program, and recognizing program fragments as components in the knowledge base. The question is, what knowledge do we want to represent, and having done so, what are we going to do with the matched (recognized) program fragments? The panelists take differing approaches to these questions. Two of the panelists view the support that program understanding brings in terms of a cooperative interaction between programmer and system; however, they differ substantially in their approach. One attempts to build a library of program cliches, with emphasis on domain-specific representations. The other begins with a large a-priori knowledge base of the program's domain, and uses this knowledge base in an interaction with the programmer as he or she queries the system about properties and components.

A third panelist looks at program understanding as a means of revitalizing ancient legacy code. This view uses the knowledge representations to assist in reverse engineering code, then re-engineering to produce better, more maintainable systems. The fourth panelist views old code as maintenance nightmare, one that grows worse with time. One effect of systems that have grown over time is that changes are more difficult to localize. This panelist describes an environment that assists in understanding the nonlocal properties of changes, with program understanding aimed at discovering interactions between fragments (views).

We briefly summarize the panelist's views on the support that program understanding methods provide the aging software problem. Then we present each panelist's position statement.

LEWIS JOHNSON views the aging software problem as one of support for program maintenance. One characteristic of programs as they age is that locality of effect seems to broaden: it becomes more and more difficult to contain the effects of changes to code. Johnson's view of program understanding is that it should extract "views," then note how changes to one view can cause changes to another. Program understanding must assist in limiting interactions among changes.

JIM NING views the problem of applying program understanding techniques to aging software as one of both extending the life of existing relics, and mining them for valuable components. His view is that he can apply "concept level understanding" methods to recover reusable components. He is working on specification recovery from old, poorly documented code, and using the recovered specifications for re-engineering.

ALEX QUILICI's view is that domain specific cliches are necessary in program understanding. He is concerned with techniques that aid the programmer in understanding the program. He believes that environments that guide and assist the programmer are required if we are to be effective in dealing with the
realities of large existing programs. His work is based on recognizing program cliches within the code. But unlike others using these methods, he emphasizes the need for domain-dependent cliches. Recognition requires a cooperative interaction between user and system, with the system providing support for recording program fragments.

PREM DEVANBU's view on program understanding support for software is how programmers can come to understand a program through cooperative interaction with a knowledge base. He is looking at the limits of knowledge bases, in particular terminological knowledge bases, in assisting programmers in discovery in very large systems under development. The knowledge base contains a domain model for the system being built. Through a natural language front end, users query the knowledge base about code fragments within the system, using classification to find the appropriate answers. The work focuses on the flexibility provided by natural language interaction, while emphasizing the semantic cleanliness of the knowledge base.

Lewis Johnson—Position Statement

Program understanding work, as exhibited in such systems as PROUST, CPU, Programmer's Apprentice, and the work of the other panelists, is increasingly capable of recognizing high-level programming plans and domain concepts in programs. We would like to hope that such techniques can help address the aging software problem.

For program understanding tools to have an impact on this problem, they must be designed to support the process of maintenance, i.e., program modification. Most existing program understanders operate in a catch-as-catch-can fashion, recognizing instances of programming plans. Such systems do not focus on understanding those aspects of the system that may be impacted by a proposed modification.

Experience with the ARIES requirements specification environment, designed to support evolution of specification, yields lessons applicable to the maintenance of aging software. Most modifications involve making changes to some aspect of the system, while limiting the impact of the change on other aspects. Therefore, program understanding should focus on the follow areas: first, program understanding should be used to extract specific views of the system. Second, analysis should identify places where the view can potentially be affected by changes to other views. Aging systems are harder to modify because the important views are harder to extract, and because more interactions between views can be expected. However, the fundamental process is the same.

Suppose, for example, we wish to modify an aging air traffic control system, to add a feature for automatically initiating handoff of control of radar tracks when the tracks reach designated crossover points between control sectors. It is useful when planning such a modification to focus on the possible states of tracks, e.g., normal, handoff-in-progress, and so on, and the operations that can lead to state transitions. The proposed change should not be inconsistent nor interfere with the existing states and transitions. For example, manual operations for initiating handoffs and cancelling handoffs could interfere with automatic handoff, and vice versa. Program understanding can be employed to recognize the components of such abstracted views, e.g., data representations for state variables and operations for performing state changes.

Programming plans must be represented so that program understanding can detect possible interactions between views. For example, suppose there is a feature for restoring the track database to an earlier state when an error occurs, or for updating the track database with information transmitted from other air traffic control centers. These operations can have indirect effects on the status of tracks. Such effects are difficult to detect automatically, but it may be possible to detect potential interactions automatically.

Program understanding offers hope for aging software only if it proves capable of extracting useful views of systems, and detecting potential interactions between those views and other parts of the system. It is unlikely that in typical aging systems, views and interactions can be identified entirely by automatic means. It will be necessary for the maintainer to provide some of the information the program understanding tool cannot extract. If the right views are chosen, this information is worth extracting in its own right.

Jim Q. Ning—Position Statement

Typically, legacy systems are very difficult to understand because of the poor structures and lack of documentation caused by many years of maintenance. Yet, these systems must be maintained, renovated, migrated, or reused because they represent valuable assets to their companies. This creates a need to automate the understanding of system programs.

We distinguish four levels of program understanding: text level, syntactic level, semantic level, and con-
cept level. Text-level understanding views a program as a sequence of text characters. Any text-based editors "understand" programs at this level. Syntactic-level understanding can be automated by parsing. Syntax-directed editors and language-to-language translation tools can be developed based on the representations generated by parsing. Semantic-level understanding derives semantic information about programs such as control flow, data flow, control dependence, and calling relations, based on programming language semantics. Most existing program understanding tools are built on this level of information. Typical examples are design graph browsing tools, verification tools, restructuring and optimization tools, slicing and ripple-effect analysis tools, and complexity analysis tools. Finally, concept-level understanding is intended to automatically recover the "meaning" of the programs.

Concept-level program understanding can be achieved by recognizing concepts in the code. There are concepts that are explicit in the program such as program statements, variables, declarations, modules, and so on. We call these language concepts because they can be "recognized" using syntactic knowledge about the programming language. There are also abstract concepts that represent language-independent but domain-specific ideas of computation. Technical aspects of abstract concept recognition can be found in the paper "Program Concept Recognition" included in these proceedings.

Concept-level program understanding is not a pure exploration of intellectual curiosity. There are many practical reasons behind it:

- High-level support for program understanding
- Recovery of reusable code and design components
- Reverse synthesis of specifications
- Support for other re-engineering tasks (code validation, transformation, and the like)

Alex Quilici—Position Statement

Automated program understanding can contribute to solving the problem of aging software, but only as part of a complete environment designed to assist programmers in understanding software.

Current systems view program understanding as the problem of recognizing instances of programming cliches. Roughly speaking, there are three classes of cliches:

- Domain-independent programming structures (such as a read-process loop)
- Domain-independent programming plans (such as a bubble sort)
- Domain-dependent programming plans (such as computing the distance between a pair of satellites)

Most program understanders focus on the first two categories. The assumption is that a significant portion of many programs is composed of some reasonably sized set of domain-independent cliches.

But is this assumption valid with aging software? We recently studied a set of C textbook programs to find out. Since these programs should be highly filled with domain-independent cliches, our guess is that they can provide a rough upper bound on the percentage of real-life code that is composed from these sorts of cliches. But over half of these programs consisted of either domain-dependent cliches or code that seemed too specialized to classify as a cliche. Our guess is that aging software systems are much worse. And in examining textbook and student explanations of program behavior, we found that it is the domain-dependent cliches that are often the key to understanding a program's behavior. Just consider the difference between describing a code fragment solely in terms of domain-independent cliches (generates pairs of array subscripts, performs a computation involving the paired elements of the array, and saves the smallest result) rather than in terms of domain-dependent cliches (finds the distance between the closest points in a set of points).

So let's be realistic: automated program understanders aren't going to be able to tackle realistic aging software systems any time soon. To do so, they would require vast numbers of domain-dependent cliches. They would have to scale up to orders of magnitude more cliches than current program understanders can deal with. And they would have to deal with plenty of code that doesn't seem to be cliched at all. So what do we do in the meantime?

Our suggestion is to view program understanding as a cooperative process between system and programmer. This suggests several duties for the system, all of which we have been exploring in the context of programs that make heavy use of geometric objects.

The first is to record as much of the current combined understanding of the program as possible. This means it is necessary to provide a mechanism by which programmers can record their understanding of the program. The programmer, for example, should be
able to record that a particular code fragment is computing the distance between two points. Similarly, it's necessary to allow the programmer to interactively indicate new cliches, as well as new variants of existing cliches. This suggests one way to gradually build a library of domain-dependent cliches. Both tasks require an explicit representation of the domain model, as well as an explicit, programmer understandable representation of cliches and the constraints used to recognize them.

The second is to suggest to the programmer what code might be doing when the system can't confirm it. The system, for example, might point out that a fragment appears to be computing a distance but the values being subtracted and squared don't appear to be coordinates. This requires extending current program understanders to use indexing techniques to limit the number of cliches considered for any given fragment of code, and to have an explicit representation of the importance of different components of cliches. Along these lines, the system must be able to explain its cliches and reasoning. Ideally, the programmer may query the system about the purpose of a particular piece of code, or why a particular cliche wasn't recognized in a particular place where the programmer believes it is occurring.

Prem Devanbu—Position Statement

In the overall arena of re-engineering and reverse engineering research our work is concerned with using knowledge based technologies to help programmers with the every day task of understanding a system under active development. One can distinguish our approach from re-engineering and reverse engineering, in general, which are usually posteriori methods to recover design from existing code, so that it can be re-implemented or restructured.

Our work, on the other hand, is concerned with helping programmers deal with "living" code. This approach is particularly viable for large and complex systems such as telephone central office switches, that are simply too expensive to re-engineer or re-implement. We envision a knowledge based system that comprises a powerful reasoning system, friendly user interface, and a customizable code analysis and querying facility. This system can be used by programmers to analyze code, generate new concepts, and incorporate these concepts into a knowledge base using classification. This knowledge base is subsequently available for querying and exploration. Customizable analysis facilities provide support for knowledge acquisition. The inferential services of the knowledge base (through classification), together with the user interface, assist programmers in finding the information they seek.

This software knowledge base is then a "living, growing" document that can help programmers do discovery.

Our overall research interests can be summarized as follows:

- What are the useful aspects of knowledge representation frameworks for representing software knowledge bases (knowledge bases used to document large software systems)? What can be done with terminological systems such as CLASSIC (our KL-ONE-like knowledge base system)?

- What inferences in the knowledge bases are useful in supporting discovery? In particular, what are the uses of terminological reasoning?

- What are the uses of generative and formal methods to give programmers a high degree of flexibility in creating tools that can assist in analysis, discovery, static code checking, design constraint and coding style enforcement, and the like?

- How can we exploit the combination of inferential ability of a system such as CLASSIC with the analysis tool-generation methods described above to help programmers do discovery, and store their "discoveries" in the knowledge base (i.e., assistance for the knowledge acquisition task)?

Biographies

Prem Devanbu

Prem Devanbu is a member of the technical staff in the Software and Systems Research Center at AT&T Bell Labs in Murray Hill, New Jersey. His main research interest is the use of database and knowledge base technologies to help programmers in large software projects manage complexity.

Lewis Johnson

Lewis Johnson is project leader of the ARIES Project and the KBSA (Knowledge-Based Software Assistant) concept demonstration project, at USC/Information Sciences Institute, and Research Assistant Professor in the Computer Science Department, University of Southern California. Johnson's interests have been
in program understanding, and transformation based software environments that support the acquisition and validation of software requirements specifications.

Lawrence Miller

Lawrence Miller (Moderator) is a member of the research staff in the Computer Systems Research Department of the Aerospace Corporation, Los Angeles, where he runs the Aging Software project. His research interests are in knowledge based techniques for program representations, and in extending the life span of old programs through enhanced user interfaces.

Jim Ning

Jim Ning is an Associate Scientist at Andersen Consulting's Center for Strategic Technology Research (CSTaR). He has been responsible for the research and development of knowledge-based program understanding and analysis systems as part of the software re-engineering environments produced by CSTaR.

Alex Quilici

Alex Quilici is Assistant Professor of Electrical Engineering at the University of Hawaii. His primary research interests lie in the intersection of AI and software engineering (constructing systems that can automatically form explanations of a program's behavior), and in natural language processing (constructing systems capable of participating in argumentative dialogs).