Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation Call for Participa

Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation Call for Participation

PROGRAM COMPREHENSION WORKSHOP

In cooperation with the Conference on Software Maintenance Orlando, Florida at the Hyatt Orlando (407.396.5090) on November 9, 1992

Position papers are due September 5, 1992. Papers are limited to 3 pages. Send to Vaclav Rajlich (address provided in Organizers box at right). Acceptance/ rejection notices will be mailed on October 1, 1992.

Workshop attendance is limited. The workshop will be organized around a very small set of presentations and subsequent discussions. All accepted position papers will be distributed to the workshop participants. The conclusions of the workshop will be presented to the Conference on Software Maintenance during a special session.

To participate in the Program Comprehension Workshop, you are invited to submit a position paper.

The issues related to programmer productivity are becoming increasingly critical. A significant part of programmer effort is devoted to software maintenance. Here, the critical issue is software comprehension.

The Program Comprehension Workshop will gather researchers from both academia and industry. They will explore the issues of software comprehension, its influence on programmer productivity, state-of-the-art, and research directions.

Position papers related, but not limited, to the following topics are invited: • *Impact of software comprehension on programmer productivity: What are the available data?* • *Theories of software comprehension* • *Experiments related to software comprehension* • *Tools facilitating software comprehension* • *(Re)Structuring software for better comprehension* • *Comprehension of programming in-the-small versus in-the-large* • *Software comprehension and AI* • *Software comprehension and general reading comprehension* • *State-of-the-art and research agendas* • *Domain knowledge and software comprehension.*

Organizers

Vaclav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202
Telephone: 313.577.5423
FAX: 313.577.6868
e-mail: vtr@cs.wayne.edu
express mail: 431 State Hall
5143 Cass Ave

Norman Wilde
Department of Computer Science
University of West Florida
Pensacola, FL 32514
e-mail: wilde@cs.uwf.ed

Malcolm Munro
Centre for Software Maintenance
University of Durham
Durham DH1 3LE, England
e-mail:
Malcolm.Munro@durham.ac.uk

Aniello Cimitile
DIS — Dipartmento di Informatica e Sistemistica
University of Naples
via Claudio 21, 80125 Naples, Italy
e-mail: cimitile@inacised.unina.it

Peter Selfridge
AT&T Bell Labs, room 2B-425
600 Mountain Ave
PO Box 636
Murray Hill, NJ 07974
e-mail: pgs@research.att.com

Workshop fee

$35 for IEEE Computer Society members. $45 for nonmembers (paid during registration). Workshop participants are encouraged to stay for the duration of the entire Conference on Software Maintenance on November 10 — 12.

Call for Participation Call for Participation Call IEEE Computer Society Call for Participation Call for Participation Call for

CSM-92

Program Comprehension

Workshop Notes

(Distribution limited to workshop attendees)

Workshop Committee:

Vaclav Rajlich, Wayne State University
Aniello Cimitile, University of Naples
Malcolm Munro, University of Durham
Peter Selfridge, AT&T Bell Labs
Norman Wilde, University of West Florida

November 9, 1992
Hyatt Orlando
Orlando, Florida

Sponsored by IEEE Computer Society

CSM-92

Program Comprehension

Workshop Notes

(Distribution limited to workshop attendees)

November 9, 1992
Hyatt Orlando
Orlando, Florida

Sponsored by IEEE Computer Society

Participants

Altizer Charles
Naval Postgraduate School
Monterey, CA 93943

Ballantyne Michael
EDS Research, Austin Lab
1601 Rio Grande, Suite 500
Austin, TX 78701

Berzins Valdis
Naval Postgraduate School
Monterey, CA 93943

Boldyreff Cornelia
School of Eng. & Comp. Sci.
Univ. of Durham, Sci. Lab.
South Road
Durham DH1 3LE,
United Kingdom

Calliss Frank W.
Dept. of Comp. Sci. & Eng.
College of Eng. & Applied Sci.
Arizona State University
Tempe, AZ 85287-5406

Canfora G..
Dis-Dep. of "Informatica e Sistemistica
University of Naples "Federico II"
via Claudio 21, 80125
Naples, Italy

Cimitile A.
Dis-Dep. of "Informatica e Sistemistica
University of Naples "Federico II"
via Claudio 21, 80125
Naples, Italy

Clayton R.
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Curtis Bill
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Cutillo Filippo
Formatica s.r.l.
Via Amendola 162/1
70126 Bari, Italy

Dampier David A.
Naval Postgraduate School
Monterey, CA 93943

De Carlini U.
Dis-Dep. of "informatica e Sistemistica
University of Naples "Federico II"
via Claudio 21, 80125
Naples, Italy

Dietrich Suzanne W.
Dept. of Comp. Sci. & Eng.
College of Eng. & Applied Sci.
Arizona State University
Tempe, AZ 85287-5406

Harrold Mary Jean
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906

Kozaczynski Wojtek
Center for Strategic Technology Research
(CSTaR), Andersen Consulting
100 South Wacker Dr.
Chicago, IL 60606

Lanubile Filippo
Dipartimento di Informatica
Universityof Bari
Via Amendola 173, 70126 Bari
Italy

LeBlanc Richard J.
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Luqi
Naval Postgraduate School
Monterey, CA 93943

Malloy Brian
Dept. of Computer Science
Clemson University
Clemson, SC 29634-1906

Munro Malcolm
Centre for Software Maintenance
University of Durham
Durham DH1 3LE
England

Ning Jim
Center for Strategic Technology Research
(CSTaR), Andersen Consulting
100 South Wacker Dr.
Chicago, IL 60606

Olshefski Davis P.
P.O. Box 704
Yorktown Heights, NY 10598

Ornburn Stephen B.
College of Computing
Georgia Institute of Tehnology
Atlanta, GA 30332-0280

Quilici Alex
University of Hawaii at Manoa
2540 Dole St, Holmes 483
Honolulu, HI 96822

Rajlich Vaclav
Department of Computer Science
Wayne State University
Detroit, MI 48202

Rugaber Spencer
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Sametinger Johannes
Institut fur Wirtschaftsinformatik
CD Laboratory for Software Engineering
Johannes Kepler University of Linz
a-4040 Linz, Austria

Selfridge Peter
AT&T Bell Labs
room 2B-425
600 Mountain Ave.
P.O. Box 636
Murray Hill, NJ 07974

Van Sickle Larry
EDS Research, Austin Lab.
1601 Rio Grande, Suite 500
Austin, TX 78701

Vans A. M.
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80524

Visaggio Giuseppe
Dipartimento di Informatica
University of Bari
Via Amendola 173, 70126 Bari
Italy

von Mayrhauser A.
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80524

Wilde Norman
Dept. of Computer Science
University of West Florida
Pensacola, FL 32514

Zuse Horst
Technische Universitat
Berlin (FR 5-3)
Franklinstrabe 28/29
1 Berlin 10
Germany

Zvegintzov Nicholas
141 Saint Marks Place, Suite 5F
Staten Island, NY 10301

CONTENTS

ROLE OF TRANSLATION MECHANISMS IN SOFTWARE COMPREHENSION

Charles Altizer and Valdis Berzins
Naval Postgraduate School
Monterey, California 93943

Abstract

Software production costs have steadily spiraled upward as technology leads to cheaper but ever more powerful hardware. Higher demands and expectations for software systems have resulted in large software systems with correspondingly large price tags. In response to growing software costs software reengineering is emerging as a viable methodology. It is often more cost-effective to keep existing software systems and maintain them as user requirements or operating environments change over time. Significant to software reengineering, are the processes of forward and reverse engineering. Due to the lack of available documentation for many legacy software systems it is necessary to analyze source code to extract design information so that modifications can be applied to the software in an effort to update it. A common point of reference for software reengineering is the original system design. The design must be understood before maintenance can be performed on software, that is, the comprehension of software systems is significant to reengineering. Some aspects of reverse engineering can be expressed as a translation process. Translation processes already exist in the realm of forward engineering which can translate system specifications into implementation source code. Sophisticated translator generator tools allow a maintenance programmer to develop forward and reverse engineering translators which automate much of the reengineering process and allow legacy systems to be effectively updated and maintained quickly at lower costs.

I. Introduction

Software reengineering refers to the act of modifying a software product to correct faults, improve performance, or adapt it to a changed environment. Software reengineering is a growing concern today because the cost of software system development has reached a point where it is often more cost-effective to keep and maintain existing software than it is to build new software systems.

At the heart of software reengineering is software comprehension, the ability to capture the purpose and design of existing software so that modifications can be intelligently implemented. Formal specification languages exist today which allow a developer to specify system requirements at a level of abstraction higher than conventional programming languages. Formal specifications can be translated into source code and subsequently executed. Such systems are common and demonstrate that formal specification languages are a feasible method of expressing the semantics of a software system. Given the ability to allow semantic expression of software systems, specification languages will play a key role in program comprehension.

Mapping formal specification languages to source code is accomplished using a translation mechanism. Translator generator tools allow a programmer to construct a translation system quickly. Given a formal source language and a target language a translator generator provides a method to describe the general semantics of the source language in terms of an attribute grammar. The attribute grammar is used to produce a translation into an equivalent expression in terms of the target language. Ordinarily, when a software system is being constructed using a specification language the system design is already known and the translator maps the design specifications into some target language like Ada, C, or pascal. When our concern is software comprehension, there is often source code but little or no accompanying design documentation. Translator generators can be used to reverse engineer the source code back to specification. Since the existing source code is written in a formal programming language, all that is necessary is to choose a formal specification language capable of representing the semantics of the source code and then derive the attribute equations necessary to map source code constructs into specification language constructs. Given these conditions, a translator generator tool can produce the translation mechanism required to capture the semantics of any program written in the source language.

This research was supported in part by the National Science Foundation under grant number CCR-9058453 and in part by the Army Research Office under grant number ARO-145-91.

II. Choosing A Suitable Specification Mechanism

The Computer Aided Prototyping System (CAPS) currently being developed at the Naval Postgraduate School is primarily a forward engineering tool. CAPS is used to rapidly prototype real-time systems[1]. Central to CAPS is a specification language called Prototyping System Description Language (PSDL). A system designer details the specifications of a software system in terms of dataflow, control flow, timing constraints, abstract data types, and execution modules. PSDL has sufficient constructs to represent all of these specifications. Such specifications are input to CAPS via various tools including a graphical editor and a syntax-directed editor. CAPS models the designer's system as a program written in PSDL. A translator built-in to CAPS translates the PSDL specification into compilable Ada modules which, when executed, will effectively implement the desired prototype system.

The ability to model the semantics of a system in terms of its dataflow, control flow, data structures, and execution modules makes PSDL a suitable standard for the semantic expression of a software system. Given an existing programming language, a translator can be constructed that will scan a program written in that language, extract information about the semantic components of the program, and finally map the semantic information into PSDL constructs. Program comprehension of an existing software system is enhanced after the system is translated to PSDL. A maintenance programmer gains insight into the system's purpose and design by studying the PSDL specifications. Modifications can then be applied to the PSDL version of the system and CAPS can subsequently reengineer the new modified system into a working prototype.

III. Domain And Problems

Software reengineering actually consists of three processes, forward engineering, restructuring, and reverse engineering.

Forward engineering is the process of translating from system requirements specifications into a physical implementation. The analysis and design phases of a software system life-cycle ideally result in a complete set of specifications for a new system. The forward engineering process takes the implementation-independent specifications and produces a physical implementation of the software system. It must be understood that the new system being created will probably be expected to have an extended lifetime, that is, maintenance will be a significant part of the system after it is delivered. Forward engineering plays a role in the software maintenance effort by embedding information in the software system that assists with program comprehension in the future. In systems like CAPS which automate forward engineering it is a relatively simple matter to automatically embed design information into the translated source code implementation. Information about dataflow, control flow, module connections, etc., can be gleaned from the PSDL code and embedded into the implementation code as comments.

It should be pointed out that the implementation source code produced by CAPS is not intended for human eyes. CAPS maintains a design database for each prototype created. If the need for maintenance arises the maintenance designer deals with the system at the design level, the Ada source code is never touched. Any design changes are maintained in the design database and the updated system specifications are automatically converted into a PSDL program which is subsequently translated into the new version of Ada source code.

Reverse engineering is the process of analyzing a subject system to identify its components and their interrelationships and then creating representations of the system in another form or at a higher level of abstraction. Translation mechanisms play a key role in reverse engineering. They provide the means to syntactically analyze a program and translate it into a higher level of abstraction where the program can be more easily comprehended.

Through the use of Knuth's attribute grammars it is possible to semantically redefine a program written in any formal language [2]. An attribute grammar associates zero or more attributes with each syntactic construct of a formal language. As a program is parsed according to its grammar rules, information about the program is collected, analyzed, created, and stored in attributes. The semantics of the original program can be expressed in terms of attribute equations on the set of attributes. A semantically equivalent translation can be produced from the set of attributes. For example, it is possible to extract information about dataflow in an Ada procedure or function by parsing the module specification. In doing so, the names of any input or output parameters can be scanned and traced throughout the module. Variable references within a module can be analyzed to determine if they are local or global. The manipulation of global variables indicates the presence of implicit dataflow in the module. The point is that such items as procedure parameters and variables are syntactically detectable through parsing but they can also be semantically analyzed through the use of an attribute grammar.

Once a translator has been constructed to recognize and analyze the syntactic constructs of a given programming language the translator is valid for any program written in that language. Existing software systems written in a com-

mon language like FORTRAN or COBOL can be submitted to a single translator designed for FORTRAN or COBOL, respectively, and a consistent translation will be produced regardless of the original application.

CAPS currently is not capable of reverse engineering. The existence of translator generator tools, however, makes this a feasible capability in future versions. As stated earlier, a translator can be produced that will translate from some original programming language into PSDL. By examining the PSDL version of the program insight can be gleaned about the purpose and design of the program so that changes can be implemented. Along these lines, it has been demonstrated that it is possible to build such a translation system for Ada modules [3]. A translation system was developed which translates an Ada module into a PSDL representation. The PSDL and Ada modules are then stored in a reusable software base to be used later. This project does not translate an entire system in Ada but it demonstrates that the automated reverse engineering technique does indeed have potential.

Software comprehension will greatly benefit from translation mechanisms which automate reverse engineering. In particular, when it becomes possible to translate a system into a PSDL representation, the maintenance programmer can be relieved of the burden of having to learn a variety of programming languages. Maintenance can instead be performed on the PSDL version at a higher level of abstraction and the forward engineering problem can be handled by CAPS.

IV. Conclusion

The task of reengineering existing software systems will continue to grow in importance as long as software production costs continue to escalate. However, reengineering is difficult, at best, when dealing with legacy software that was developed at a time when modern software engineering and maintenance techniques did not exist. Technology is quickly moving to the point where many aspects of software reengineering can be automated. The CAPS project at NPS has demonstrated that through the use of sophisticated translator generation tools it is possible to deal with software systems at a high level of abstraction, particularly the design level. PSDL allows a designer to specify the requirements for a real-time software system. It is suggested that PSDL can be used to express the semantics of a software system by formally defining the system in terms of its dataflow, control flow, data structures, and execution modules. For purposes of reengineering, technology exists which suggests that translators can be produced which are capable of scanning existing software systems and mapping them into a semantic expression in terms of PSDL. Once a program can be converted to a high level abstraction, it is relatively straightforward to move from the abstraction forward into a new implementation of the original system, thus completing the whole software reengineering process.

List Of References

1. Luqi, "Software Evolution Through Rapid Prototyping," IEEE Computer, Volume 22, Number 5, pp. 13-25, May 1989.

2. Knuth, D.E., "Semantics of Context Free Languages," Mathematical Systems Theory, November 1967, pp. 127-145.

3. Sealander, J.M.,"Building Reusable Software Components for Automated Retrieval," M.S. Thesis, Naval Postgraduate School, September 1992.

Design Frameworks: a Basis for Recording Program Understanding

Cornelia Boldyreff

School of Engineering and Computer Science,
University of Durham,
Science Laboratories,
South Road,
Durham DH1 3LE, U. K.
Telephone: +44 (91) 374 2638 (direct line).
Fax: +44 (91) 374 3741 (via the Computer Unit).
E-mail: Cornelia.Boldyreff@durham.ac.uk

EXTENDED ABSTRACT

A broader perspective of software engineering research now views the whole of the software life cycle as inherently encompassing both software reuse and maintenance, see, for example, the research agenda proposed by the Computer Science and Technology Board reported in the March 1990 issue of the CACM (1990a). A long-term action identified as part of this agenda is the building of a unifying model for software system development. This research action is motivated by a fear that many of the large computer-based systems on which our society depends are becoming unmaintainable; and a pressing need to improve our understanding of how to create and maintain large and complex software systems.

As systems grow over time in size and complexity; various problems arise. Old systems may become insupportable because of reliance on out-dated platforms, software as well as hardware. Large systems may be in part the result of accummulation of dead code as new components are added to an existing system. Uncontrolled evolution of software results in unnecessary complexity.

How can software engineers build bridges from old software systems to the improved systems that are needed in a changing world? Ideally new development must be able to exploit advances in technology and design techniques supporting better engineering practice without throwing away the hard earned experience and know-how embodied in what might be termed "existing software assets".

The development of software concept reuse provides one such bridge, particularly where existing software concepts are taken as the starting point for populating a reuse support system (Boldyreff1990a, Boldyreff1992a). Through the reuse of software concepts, the designer is able to build new systems on established conceptual foundations while still taking advantage of improved technology and techniques in the implementation of the new system. An understanding of the software concepts employed in the construction of a system also provides a firm basis for its maintenance, particularly, where in the form of perfective maintenance, an evolutionary approach is taken.

Futhermore, large software systems have evolved to the point where many organisations both in industry and commerce could not operate effectively without them. It is important to ensure that the knowledge and experience of those commissioning, developing and maintaining such systems takes a recorded form open to scrutiny for managerial as well as technical evaluation and appraisal. Such understanding of software must be recorded at appropriate levels to ensure its accessibility to all those within an organisation who need to rely on it throughout its lifetime.

The method of design frameworks developed by the author (Boldyreff1992b, Boldyreff1990b) has been developed to support both the recording of program understanding and its subsequent reuse in both maintenance and development. Although the initial focus in development has been primarily on describing software concepts at the higher levels of abstraction required to facilitate their reuse during

the conceptual phase of design, this method is equally applicable to accommodating other levels of understanding more appropriate for maintenance of existing software.

Lack of such understanding is acknowledged as a potential inhibitor to productivity as the following quote from Corbi's paper on Program Understanding illustrates:

> *In terms of skills that are needed as our software assets grow and age, lack of academic training in how to go about understanding programs will be a major inhibitor to programmer productivity in the 1990s.* (Corbi1989a)

Corbi also points out that programmers having gained understanding of particular systems may become key "gurus" and find their careers are inhibited because their expertise is so valued that they cannot be spared to work on anything else, thus they are trapped. Software maintenance based on explicitly recorded understanding frees such guru programmers from being trapped in this way. Codifying experts' experience results in its availability for both further development and new developments where it can be reused. Attempting to maintain software without publically recorded conceptual understanding is simply storing up problems for the future, and design frameworks provide a means of accomplishing this.

The development and use of a design framework in the domain of Steel Production has given this research its practical application (Boldyreff1992c). The control systems found in a typical steel mill are subject to continuous development and provided the basis of a case study in which the design framework method was successfully applied to support both maintenance and reuse.

References

1990a.CSTB Computer Science and Technology Board, "Scaling Up: A Research Agenda for Software Engineering," *Communications of the ACM*, vol. 33, no. 3, pp. 281-293, March 1990. Excerpts from the report by the Computer Science and Technology Board.

Boldyreff1990b.
Cornelia Boldyreff, "Supporting System Design From Reusable Design Frameworks," in *Proceedings of the Second International Conference on INFORMATION SYSTEM DEVELOPERS WORKBENCH Methodologies, Techniques, Tools and Procedures, Gdansk, 25-28 September 1990*, University of Gdansk, 1990.

Boldyreff1992a.
Cornelia Boldyreff and Uwe Krohn, "The Practitioner Reuse Support System (PRESS): A Consideration from the Standpoint of Tool Interconnection," in *Proceedings of the Fourth IFAC/IFIP Workshop on Experience with the Management of Software Projects, Austria, May 18-19, 1992*, 1992.

Boldyreff1992c.
Cornelia Boldyreff, "A Design framework for Software Concepts in the Domain of Steel Production," in *Proceedings of the Third International Conference on INFORMATION SYSTEM DEVELOPERS WORKBENCH Methodologies, Techniques, Tools and Procedures, Gdansk, 22-24 September 1992*, University of Gdansk, 1992. In press.

Boldyreff1990a.
C. Boldyreff, P. Elzer, P. Hall, U. Kaaber, J. Keilmann, and J. Witt, "PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software," in *Proceedings of Software Engineering 1990, Brighton, UK*, Cambridge University Press, 1990.

Boldyreff1992b.
C. Boldyreff, "Design methods for integrating system components," in *Software Reuse and Reverse Engineering in Practice*, ed. P. A. V. Hall, pp. 81-97, Chapman & Hall, 1992.

Corbi1989a.
T. A. Corbi, "Program Understanding: Challenge for the 1900's," *IBM Systems Journal*, vol. 28, no. 2, February 1989.

Specification Recovery using Transformations

Frank W. Calliss

Department of Computer Science and Engineering
College of Engineering and Applied Sciences
Arizona State University
Tempe, AZ 85287-5406, U.S.A.

Motivation

Existing reverse engineering techniques are not well suited to large programs, as they often require that an entire program be analyzed before any meaningful information is returned to a programmer. A form of incremental reverse engineering is presented, whereby a programmer can derive the specification for sections of the program's code. These partial specifications can be stored in such a way that they can be reused in later reverse engineering activities, thereby overcoming one of the problems that exist with current reverse engineering techniques, namely that the result of a previous reverse engineering activity is lost or recorded in a form that does not facilitates reuse.

Another important aspect of the technique is that a programmer should not always have to reverse engineer an entire program whenever a modification has been made to the code. The proposed technique intends to help a programmer reverse engineer only that section of code affected by a modification. This allows the reverse engineering to be performed much quicker. All of these points will be outlined in more detail in the following sections.

Specification Recovery

Most reverse engineering techniques consist of a combination of programmer expertise and informal strategies. This is an unacceptable approach when we are deriving a formal specification of a program, where a formal specification is a mathematical description of *what* the code is doing. The proposed technique uses a combination of program transformations and program slicing to help a programmer derive the specification of a program. The use of program transformations provide a rigorous technique whereby a specification can be derived using a formal and well defined process. The use of program slicing helps with the use of transformations as they limit the code that a programmer has to deal with. The use of program slicing also helps in limiting the amount of rederivation of specifications that needs to be performed as a result of a program modification. Each of these points will be explained below.

Program Slices

The concep of a program slice was introduced by Weiser [2]. Weiser defines a program slice to be a sequence of statements taken from a section of code, where the order of the statements is the same as in the original section of code, and where all the statement affect the value of a variable in some way.

With procedural programming languages, programs function by manipulating or modifying data values in specific ways. Therefore, in order to understand what a routine is doing, we need to understand what is happening to the data values in the routine. A slice can provide a description on *how* a portion of code achieves some function. Consider a routine R. The function of R can be characterized as being the combination of the slices that R consists of. These are the slices with respect to the local variables, formal parameters and global variables. In practice not all these slices need to be considered as not all slices contribute to the perceived function. For example, a user only recognizes the function of a routine by the output that it generates. The output of a routine

is through the output parameters, the global variables that are modified, and if the routine is a function, the result. As a result, the function of R can be characterized as the combined function of the slices of R with respect to the modified global variables, output parameters and local variables that are involved with the expression returned as the result of a function.

Deriving a Specification

The process of deriving the specification of a program is done in two stages: firstly the algorithms being employed are unravelled, and secondly the algorithms are then transformed into specifications.

Program slicing can help in deriving the algebraic description by helping a programmer to concentrate on only those statements that contribute to some function.

Associated with a program is a logical formula which encapsulates the initial conditions under which the programs is guaranteed to terminate in a state that satisfies a given final condition. This formula is Dijkstra's weakest precondition wp(P, R), the weakest precondition that ensures that program P terminates in a state R.

As an example of of using weakest preconditions for deriving the specification of a section of code consider the following section of code.

```
procedure MYSTERY(x, y:  in out INTEGER) is
begin
   y:= y + x; -- S1
   x:= y - x; -- S2
   y:= y - x; -- S3
end MYSTERY;
```

The weakest precondition for the sequence S1;S2;S3 is,

$$\text{wp(S1;S2;S3, } y = y_\omega \wedge x = x_\omega)$$

where y_ω and x_ω are the final values of y and x respectively. This weakest precondition is equivalent to,

$$\text{wp(S1, wp(S2, wp(S3, } y = y_\omega \wedge x = x_\omega)))$$

As S1, S2 and S3 are assignments we only need to consider the weakest precondition for assignments. This is

$$\text{wp(target:= expr, R)} = R\left[\frac{\text{expr}}{\text{target}}\right]$$

where $R\left[\frac{\text{expr}}{\text{target}}\right]$ means that all instances of target are replaced by expr in R. Applying this rule to each of the weakest precondition for S1, S2 and S3 we get,

$$\begin{aligned}
\text{wp}(S3, y = y_\omega \wedge x = x_\omega) &= (y = y_\omega) \wedge (x = x_\omega)\left[\frac{y - x}{y}\right] \\
&= y - x = y_\omega \wedge x = x_\omega \\
\text{wp}(S2, y - x = y_\omega \wedge x = x_\omega) &= (y - x = y_\omega) \wedge (x = x_\omega)\left[\frac{y - x}{x}\right] \\
&= (y - y + x = y_\omega) \wedge (y - x = x_\omega) \\
&= (x = y_\omega) \wedge (y - x = x_\omega) \\
\text{wp}(S1, x = y_\omega \wedge y - x = x_\omega) &= (x = y_\omega) \wedge (y - x = x_\omega)\left[\frac{y - x}{y}\right] \\
&= (x = y_\omega) \wedge (y - x - x = x_\omega) \\
&= (x = y_\omega) \wedge (y = x_\omega)
\end{aligned}$$

This shows that the procedure MYSTERY swaps the value of is parameters, so a specification for this procedure could be of the form

```
procedure MYSTERY(x, y:  in out INTEGER)
pre  x=x0 and y=y0
post x=y0 and y=x0
```

This was just a simple example of the use of weakest preconditions for reverse engineering. Other transformations also exist that can be use in reverse engineering [1]. The selection and automation of these transformation can be semi-automated, thereby helping to ensure that programmers do not reach the wrong conclusion because of an error in the reverse engineering process.

Combining Specifications

An important part of the proposed research is the ability to combine specifications. Consider a routine P that is composed of the slices S_1, S_2,..., S_n. If $\mathcal{M}$ is a meaning function that derives a specification then,

$$\mathcal{M}[\![P]\!] \equiv \mathcal{M}[\![S_1]\!] \uplus \mathcal{M}[\![S_2]\!] \uplus \ldots \uplus \mathcal{M}[\![S_n]\!]$$

That is, the specification of P is the compound specification of the specification for the slices S_1, S_2,..., S_n.

An important contribution of this ability to combine slices. The amount of reverse engineering needed after a modification to a section of code already reversed engineered is minimized. When a program is modified it is only be necessary to derive the specifications for the affected slices. The specification for the routine can be obtained by combining the specifications for the unaffected slices with the specifications for the new slices.

Storing Slices and Specifications

In order to be able to reuse the results of past reverse engineering work, it is important that the slices and the associated specifications be stored in a database that records the links between a slice and its specification. A hypertext database will be needed to store the slices and associated plans, as it is not possible to dictate a strict description of the data that relational databases require. The reverse engineering tool will interact with this database by storing slices and specifications as well as retrieving slices and plans. In order to keep the database current, a program slicer will generate all the slices for a section of code. If that section of code has been previously analyzed, then a comparison will be made between the current set of slices and the old set. Any slices that existed in the old section of code but not the new one will be removed the database.

References

[1] M. Ward, F.W. Calliss and M. Munro, "The Maintainer's Assistant," in *Proceedings of the Conference on Software Maintenance - 1989*, IEEE Computer Society Press, pp. 307-315, October 1989.

[2] M. Weiser, "Programmers use Slices when Debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, July 1982.

Understanding Object-Oriented Programs Through Deduction

Frank W. Calliss and Suzanne W. Dietrich

Department of Computer Science and Engineering
College of Engineering and Applied Sciences
Arizona State University
Tempe, AZ 85287–5406, U.S.A.

Introduction

Our system for program comprehension targets object-oriented languages by focusing on understanding the dependencies between the program's components. Specifically, our system uses inter-module code analysis [1], which is a process by which a programmer can analyse a program consisting of interconnected modules.

Several approaches for program comprehension exist, including plan-based systems, e.g., PROUST [6] and the Programmer's Apprentice [7] and relational database systems, e.g., CIA [2]. Our system is rule-based, investigating the use of deductive databases as an enabling technology for understanding large object-oriented programs. The goal of this work is to take advantage of the flexibility and extensibility that the rule-based (deductive) databases have to offer.

Deductive Databases

A deductive database is an extension of a relational database that utilizes a declarative, logic-based language for database operations. This declarative language offers the flexibility to define a relation explicitly with facts, implicitly with rules or a combination thereof. This flexibility offers a referential transparency to the maintenance programmer such that references to any relation are the same. This abstraction also provides a level of data independence, which gives the knowledge-base system the ability to provide optimization techniques for evaluation. For example, a common global optimization technique, known as multiple query optimization, is to materialize a relation defined by rules for subsequent use in a user session rather than having to re-derive the relation on each reference.

The application of a deductive database is appropriate for the complex domain of inter-module code analysis. Since the components of a program are highly structured through the concepts of classes (superclasses and subclasses), the use of a deductive database allows the modelling of the superclass/subclass relationship by its rule capability. Rules allow for the derivation of information and provide a natural mechanism for inheritance in class hierarchies. These superclass/subclass relationships lead to the specification of queries that are inherently recursive. Relational database query languages, such as SQL, do not have the capability to express recursive queries. Relational databases use a query language embedded in a host language to answer such queries. A deductive database allows for the specification of recursive queries using a single, declarative language. Since a deductive database extends relational technology, a deductive database retains the conceptual simplicity of relational databases, yet offers additional flexibility by providing a single, computationally complete language with a unified rule capability that allows for the declarative specification of recursive views and complex ad-hoc queries.

Understanding Through Deduction

Our system uses deductive database technology to store, derive and reason about the program. The conceptual design of the code analysis knowledge-base [4] uses both facts and rules. A language-specific preprocessor processes a successfully compiled program to create the factual database instance, while uniquely numbering the regions in the program with a non-negative integer. The declarative rule capability

is used to derive and reason about the information in the database. We currently identify three uses of rules in our system: *schema rules*, *code query rules*, and *problem domain rules*.

Schema rules are rules that form an integral component of the database design. Our conceptual design emphasizes class hierarchies of program components. Rules provide a natural mechanism for the derivation of information using the class hierarchies.

As an example, consider languages that allow for local and global modules (classes). A global module is a module that is declared in the outermost region of the program, whereas a local module is a module that is declared within another module. Schema rules for the global and local modules are:

```
global_module(Pgm, Entity, 0) :-
   module(Pgm, Entity, 0, OwnReg).
local_module(Pgm, Entity, DecReg) :-
   module(Pgm, Entity, DecReg, OwnReg), DecReg > 0.
```

The rule for `global_module` indicates that a global module is any module that is declared in region 0, the number given to the outermost region in a program. The rule for `local_module` indicates that a local module is any module declared in a region whose number is greater than 0.

Code query rules derive information about the source code. These rules can be arbitrarily complex, including the power of recursion. Typical recursive queries in inter-module code analysis include finding the origin of a program component (e.g. in a program that contains nested regions and inheritance [3]) and finding the closure of variable aliasing.

As an example, consider aliasing where a variable is known by more than one name. The following rules define the `alias` relation as the symmetric and transitive closure of the `redefines` factual relation, which is assumed to be populated by the preprocessor based upon the aliasing capabilities of the language.

```
alias(VAR, ALIAS) :- redefines(ALIAS, VAR).
alias(VAR, ALIAS) :- alias(ALIAS, VAR).
alias(VAR, ALIAS) :- alias(VAR, ALIASX), alias(ALIAS_X, ALIAS).
```

The first rule for `alias` indicates that a redefinition of a variable is an alias. The second rule for `alias` defines the symmetry of the arguments since there is no ordering of the arguments of an alias. The third rule for `alias` defines the transitive closure of the `redefines` relation: if `ALIAS_X` is an alias of `VAR` and `ALIAS` is an alias of `ALIAS_X`, then by transitivity, `ALIAS` is an alias of `VAR`.

The following query finds all aliases for variables that have been redefined :

```
alias(VAR, ALIAS).
```

If we want to disallow the case where a variable is its own alias, we can add another rule to compute the answer:

```
answer(VAR, ALIAS) :- alias(ALIAS, VAR), VAR \== ALIAS.
```

which checks to see that `VAR` and `ALIAS` are not identical before returning the answer.

The inherent rule capability of deductive databases also allows a programmer to formulate queries at the problem level, and to have the answer represented at the program level. This is a form of *problem domain rule*. This is done by having semantic information recorded as facts and rules, which can then be used by the deductive database to answer queries. This capability to draw on semantic information to assist in deducing information is an improvement over plan based system.

Consider for example a program that records zip codes. A `what_is` relation records a known mapping between a real world entity and program entity. With our example, the real world entity is the zip code, and the program entities are the variable(s) that represent a zip code. Let us assume that the variable `Z` represents a zip code. This is represented by the following fact.

```
what_is('zip-code', 'Z').
```

With this fact recorded, then the programmer need only ask

```
what_is('zip-code', PE_ZIP).
```

and `PE_ZIP` would take on the value of all the variables that are explicitly linked to the entity `zip-code`.

The `what_is` relation is bi-directional, so a programmer can also ask,

```
what_is(RW_Ent, 'Z').
```

and they will discover that `Z` represents the zip code.

These facts do not take advantage of any known dependencies between program entities. For example, if another variable is an alias for `Z` then that variable is also a zip code, but why should this information have to be explicitly recorded? This weakness can be overcome by adding a rule called `what_is` that *derives* the mapping between real world entities and program entities:

```
what_is(RW_Entity, P_Entity) :- what_is(RW_Entity, TMP_Entity),
                                alias(TMP_Entity, P_Entity).
```

This last example outlines several important features that deductive databases offer. The ability to build rules from other rules shows the extendability of the database. The referential transparency between facts and rules means that programmers do not have to worry about whether the information is explicitly stored or derived. The same query will work regardless.

Summary

In summary, deductive databases allow for the declarative specification and *efficient* evaluation of (recursive) rules expressed in logic. By extending the conceptual simplicity of the relational data model with a rule capability, deductive databases offer the flexibility to both retrieve and *reason* about knowledge in the database, which is in the form of both facts and rules.

The reasoning capability of the rule-based system allows for the extension of the system in several important directions. One extension is the inclusion of a change tracking [5] expert system, which would be fully integrated with the program comprehension component. Another extension would be to widen the scope of understanding to a *collection* of programs that share persistent data.

References

[1] Calliss, F.W., *Inter-Module Code Analysis Techniques for Software Maintenance.* PhD thesis, University of Durham, Computer Science, 1989.

[2] Chen, Y.F., Nishimoto, M., and Ramamoorthy, C.V., "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 325–334, March 1990.

[3] Dietrich, S.W. and Calliss, F.W., "The Application of Deductive Databases to Inter-Module Code Analysis," in *Proceedings of the Conference on Software Maintenance — 1991*, (Sorrento, Italy), pp. 120–129, IEEE Computer Society Press, October 1991.

[4] Dietrich, S.W. and Calliss, F.W., "A Conceptual Design for a Code Analysis Knowledge Base," *Journal of Software Maintenance — Research and Practice*, vol. 4, no. 1, pp. 19–36, March 1992.

[5] Garland, J.K. and Calliss, F.W., "Improved Change Tracking for Software Maintenance," in *Conference on Software Maintenance — 1991*, (Sorrento, Italy), pp. 32–41, IEEE Computer Society Press, November 1991.

[6] Johnson, W.L. and Soloway, E., "PROUST: Knowledge-Based Program Understanding," in *Proceedings of the 7th International Conference on Software Engineering*, (Washington, D.C.), pp. 369–380, IEEE Computer Society Press, November 1985. Also in 'Readings in Artificial Intelligence and Software Engineering', eds. Rich, C. and Waters, R.C.

[7] Rich, C., "A Formal Representation for Plans in the Programmer's Apprentice," in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, (Vancouver), pp. 1044–1053, IJCAI, August 1981. Also in 'Readings in Artificial Intelligence and Software Engineering', eds. Rich, C. and Waters, R.C.

VAPS: Visual Aids for Pascal Software Comprehension

(Position Paper)

G. Canfora, A. Cimitile and U. De Carlini
DIS — Dep. of "Informatica e Sistemistica"
University of Naples "Federico II"
via Claudio 21, 80125, Naples Italy

This position statement is intended to present Visual Aids for Pascal Software comprehension (VAPS) a prototype environment to support the program understanding and design recovery activities in the maintenance and reuse re-engineering processes.

Understanding programs is one of the most complex, time consuming and expensive activities in the processes of the engineering of existing software systems.

The activity of understanding a software system benefits from the availability of documents that describe the software system at higher levels of abstraction than code. Unfortunately experience shows that these documents, that should typically be produced in the system specification and design phases, are rarely available. Even when these documents are available, their degree of consistency with code is generally very low because of previous maintenance operations that are not reflected in the system's documentation. In any case the degree of consistency between documentation and code is very difficult to verify.

The consequence is that quite often the source code is the only item on which the program understanding activities can be based. But source code is too detailed for human readers and understanding the overall architecture of a large software system by manual code inspection is an almost impossible task. More abstract and readable representations must be produced from code by reverse engineering.

The number of reverse engineering tools for the production of design level documents proposed by either research or industrial organisations is constantly growing. Nevertheless, many open problems prevent these tools from being widely used in the software production environments. These problems are mainly related to the nature of the documents produced by reverse engineering, which poses more than just a few problems in designing the user interface for the reverse engineering tools.

Despite the fact that the documents produced by reverse engineering tools are the ones defined by well-known software development methodologies introduced in the 1970s, these documents tend to be much more specific and rich in details than the documents produced in the forward engineering activities. Such a richness provides the user with information that is relevant to understand the program, although it is not useful in the software development and therefore is not shown in the corresponding documents.

On the other hand if such information is directly given to the user, he may get lost in the large amount of details available. In this case the richness of the information available slows down program understanding by compelling the user to look for the piece of information he needs within a large amount of unnecessary details.

For a reverse engineering tool to be effective in program understanding the interaction with the user must be as natural and simple as possible and the information recovered must be presented in a highly readable format.

The front end interface should allow the user to select the piece of information he needs for developing his activity and should provide him with more details as his confidence with the system grows. This may be obtained, for example, by collecting the documents produced in an hyper-textual fashion and presenting them to the user in a graphical form.

Hyper-texts are a good base to develop a tool aimed to support the design recovery and program understanding activities. A hyper-text system associates windows with objects in the data base of the documents produced by reverse engineering, thus allowing the user to navigate trough the various levels of documentation of a software system — from the high level design to the code. In such a way the needed information can be easily retrieved

and each document can be related to other documents or to pieces of code. Furthermore the use of graphs to represent the information obtained by reverse engineering can improve the readability of the documents produced, since graphs are much more intuitive and effective than textual representations.

The readability of a graph greatly depends on the way in which it is displayed, i.e. on the layout adopted. A graph can be represented by several different layouts, but only few of these are the optimum to express the information content of the graph in an incisive and intuitive way. As the graphs to be displayed grow in size the manual production of the optimum layout become more and more difficult and, therefore, automatic layout algorithms must be used. Automatic layout algorithms reduce the cost to produce and manage graphs by generating the optimum layout on the basis of the type of the graph to be displayed (i.e. trees, planar graphs, directed graphs) and of some additional user's requirements (i.e. minimising the edge crossings and/or the area occupied or preserving the possible symmetries of the graph).

The above considerations acted as a requirement and a guideline in the design of VAPS.

The VAPS environment is being developed in an on-going project jointly by the DIS (Dep. of "Informatica e Sistemistica") of the University of Naples and the DIS of the University of Rome. VAPS consists of a reverse engineering subsystem and a visual environment. The reverse engineering subsystem produces information and documents by static analysis of Pascal code, while the visual environment interacts with the user. Both graphical and textual documents are used to provide the user with the information abstracted by reverse engineering. All the documents produced are linked together in an hyper-text fashion and a multi-window environment is used to show them. These documents describe a software system according to three different levels of abstractions: high level design, low level design and code.

The first level describes the relationships existing among the procedures and/or functions of a Pascal software system, while the second one describes the relationships existing among the components of the control flow of a procedure or a function.

In the current implementation the documents produced for the first level are the declaration tree (a tree showing the declaration nesting of procedures and/or functions) and the structure chart. The documents produced for the second level are the nesting tree (showing the nesting relationships among the 1-in/1-out control structures), the exemplar-path tree (describing the equivalence classes defined in the set of the potential execution of a procedure and/or function) and the control flow graph. All these documents are presented to the user in a graphical form.

VAPS is provided with a library of automatic layout algorithms: all the graphs are automatically displayed by one of the algorithms in the library. The most suitable layout algorithm for a given graph is automatically selected on the basis of the characteristics of the graph. Editing operations are also provided in order to allow the user to tailor the layout.

Passing from a level of abstraction to the one above and displaying one or more documents requires simple mouse pointing and clicking operations. For example a window showing the nesting tree of a procedure can be opened by clicking the related node of the structure chart; in a similar way the code of a control structure can be displayed by clicking the related node in the nesting tree. The declarative section of a procedure or function and comments, if any, describing its general meaning can also be displayed by icon selection and clicking operations.

The current version of VAPS runs in the MS-DOS/Microsoft-Windows environment and analyses both standard Pascal and TurboPascal programs. A new version is being designed for the Unix/X environment.

Next Steps for Literate Programming

R. Clayton
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
clayton@cc.gatech.edu

Introduction

Code comments have long been recognized as a treacherous form of program documentation, either because they don't exist or they bear at best a passing resemblance to the code they're supposed to describe. *Literate programming* is Donald Knuth's attempt to restore the value of code comments as program documentation by tightly intertwining the code and comments, and by making comment writing the leading component of program development [Knu84].

The layout of a system supporting literate programming system is shown in figure 1; the term "WEB" refers to a general literate programming system. The literate programmer writes a WEB document consisting of both the program code and the documentation for the program. The program code syntax is more or less that of the programming language being used; WEB defines some syntax extensions of its own and allows the programmer to extend the syntax with macros. The documentation text is written in the style required by the text formatter used in WEB; in almost all cases this is TEX. The literate programmer writes simultaneously in two worlds: that of the code and that of the documentation. The two worlds can be mixed to allow both snippets of formatted code in the documentation and well-formatted comments in the code.

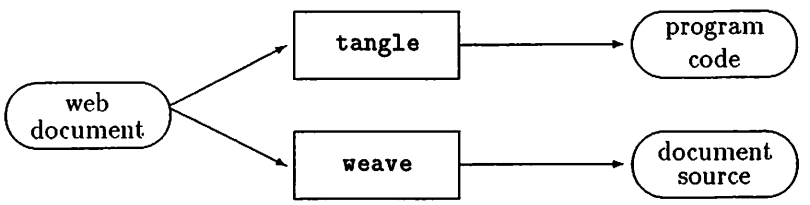

Figure 1: The WEB system

The two major programs in a WEB system are **tangle** and **weave**. **tangle** takes a WEB document and strips out all the documentation, leaving the program text, which is rearranged in a form correct for the programming language's syntax. The result is a file containing program code. **weave** takes a WEB document and produces the documentation source in a form suitable for the text processor. Figure 2 shows a fragment of a WEB document (from **weave** itself) after it has been run through **weave** and formatted with TEX.

Since literate programming is an unusual method of system development, it would be comforting to have some indication that it has any value at all. Developing rigorous indications of value is a research topic, but two literate programs have been published in book form ([Knu86a] and [Knu86b]), so at least one person has used literate programming to develop systems of significant size. Other reports ([RM91], [RS89]) indicate that people other than Knuth can use literate programming to develop systems.

1. To insert token-list p into the output, the *push_level* subroutine is called; it saves the old level of output and gets a new one going. The value of *cur_mode* is not changed.

```
procedure push_level(p : text_pointer);  { suspends the current level }
    begin
    if stack_ptr = stack_size then overflow('stack')
    else
      begin if stack_ptr > 0 then stack[stack_ptr] ← cur_state;  { save cur_end ... cur_mode }
      incr(stack_ptr); stat
      if stack_ptr > max_stack_ptr then max_stack_ptr ← stack_ptr; tats
      cur_tok ← tok_start[p]; cur_end ← tok_start[p + 1];
      end ;
      end ;
```

Figure 2: A literate programming fragment

Next Steps

Soon after Knuth published [Knu84], the ongoing process of revising WEB began [Lec85]. I consider most revisions to be in one of two categories: general system enhancements or change of medium.

General system enhancements include work to extend the programming languages covered by a WEB system (e.g., C-WEB) or creating a meta-level WEB (SPIDER) as well as all of the usual bell and whistle development (e.g., the ability to write several program code files or controlling the format of program text in the documentation); see [Sew89] for details.

Changing the medium involves moving WEB from paper to a computer-based medium, which is almost always some form of hypermedia. The move to hypermedia involves both the process of creating a WEB document and the process of reading it; see [SP92] for an example.

What follows are my suggestions for further enhancements to WEB. They are a result by my experiences with literate programming and are ones that I find particularly interesting or important:

Multi-lingual programming The ability to describe a system written in several languages. Even if a system is written in just one language, it usually ends up being multi-lingual due to things like make files or other forms of job control languages. In any case, WEB should be able to deal cleanly and consistently with whatever set of languages are used to implement a system.

Generic literate programming A current trend in text processing is to cleanly and completely separate a documents logical structure from its physical representation in print. The prime exemplar of this trend is the Standard Generic Markup Language (SGML, [Gol86]). Although literate programming is not quite the same as regular document writing, the difficulties in defining an SGML-derived markup language for literate programming are probably far smaller then the benefits of using the language in literate programs. In addition to text-formatter independence, SGML would impose (though its document type description) a database-like organization on literate programs, raising the possibility of all sorts of interesting program retrieval operations.

History A program's history comes from two sources: planed changes due to versions and unplanned changes due to error repair. The value of being able to track a program's development though its versions is well known. Knuth has argued that, keeping track of a program's

history through its error maintenance is also valuable [Knu89]. WEB should be able to capture all forms of a program's variant representation and present in which ever way the reader finds convenient.

Tailored documents A WEB document currently generates one type of system documentation. As WEB documents cover more of a program's design and development history, they will interest a wider range of readers, who will begin to express a preference for one type of information over another. WEB should be able to both extract and present information according to reader specifications.

Program typography Despite other possible audiences, programs are always designed to be read by compilers, and as such are full of syntactical and lexical decorations to help the compiler in its reading. At best, however, decorations like line ending semi-colons and begin/end bracketing are of marginal help to the human reader and are most likely just noise. WEB provides the opportunity to redesign program typography for humans. Note the motivation here is "perfection is the state in which there's nothing left to remove" and not its opposite ("... when there is nothing left to add").

The combination of multi-lingual and historical literate programming combine to provide a important feature in program comprehension: the ability to capture a program's design history and intertwine the history with the program's development. This feature is particularly useful when formal methods are a part of a program's development.

References

[Gol86] C. F. Goldfarb. *The Standard Generalized Markup Language (ISO 8879).* International Organization for Standardization, Geneva, 1986.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[Knu86a] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting.* Addison-Wesley, Reading, Mass., 1986.

[Knu86b] Donald E. Knuth. *METAFONT: The Program*, volume D of *Computers & Typesetting.* Addison-Wesley, Reading, Mass., 1986.

[Knu89] Donald E. Knuth. The errors of TEX. *Software—Practice and Experience*, 19(7):607–685, July 1989.

[Lec85] O. Lecarme. Literate programming. *ACM Computing Reviews*, 26(1):75, January 1985.

[RM91] Norman Ramsey and C. Marceau. Literate programming on a team project. *Software—Practice and Experience*, 24(6):52–61, June 1991.

[RS89] T. Reenskaug and A. L. Skaar. An environment for literate Smalltalk programming. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 337–45, New Orleans, Louisiana, October 1989. SIGPLAN, ACM Press.

[Sew89] Wayne Sewell. *Weaving a Program.* Van Nostrand Reinhold, New York, 1989.

[SP92] J. Sametinger and G. Pomberger. A hypertext system for literate C++ programming. *Journal of Object-Oriented Programming*, 4(8):24–29, Jan 1992.

Position Statement — Program Comprehension Workshop

Bill Curtis
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-7619
curtis@sei.cmu.edu

I would like to participate in the Program Comprehension Workshop. As a position paper I offer the text of a cognitive model of program comprehension I published in the Journal of Systems and Software in the February 1989 issue. This model was developed in conjunction with an experiment on how documentation formats affected software comprehension. This model led to some explicit hypotheses about the effects of documentation formats, some of which were consistent with the data my coauthors and I collected.

3. EXPERIMENT 1—COMPREHENSION

3.1 A Model of Program Comprehension

The comprehension of a program or its design is a central component of many larger software tasks such as coding, debugging, and modification. Based on observations from our previous research [50], and reports such as that by Basili and Mills [5], we propose a simple model of program comprehension from which to derive our dependent variables and operational hypotheses for this experiment. The model will be limited to the performance of tasks involving a single module rather than to tasks involving several modules. The structure of

a model for multimodule tasks might be similar to the one proposed here, but would be tailored for processes at a higher level of program abstraction.

For experienced, competent programmers, the process of comprehending a modular program can be decomposed into at least four processes; segmentation, hypothesis formation, verification, and synthesis. Each of these processes employs at least some information that is unique to the process. Further, these processes involve both top-down and bottom-up reasoning and are not strictly sequential.

Segmentation. Segmentation is a process of breaking the program into manageable pieces for comprehension. Basili and Mills [5] suggest that this segmentation process should be organized around prime programs. The work of Soloway and Ehrlich [59] suggested that these segments would be chunks of the program that a programmer can recognize as implementing a particular plan, a whole or partial algorithm, or section of code that executes a recognizable function.

Curtis et al. [23] argue that this segmentation process frequently begins with the syntactic structure of the program, especially the high-level program syntax representing the control structure. Segmentation based on the control structure can result in a prime decomposition of the program [36]. However, the information that describes what the program actually does, the plan structure (Soloway and Ehrlich [59]), typically lies embedded in or threaded through the control structure and would not be immediately apparent from a segmentation of the program. Segmentation merely divides the problem so that other processes can begin to reconstruct the semantics of the program. The separation of segmentation from other comprehension processes is similar to Shneiderman and Mayer's distinction between syntactic and semantic aspects of programming behavior [55]. An excellent example of segmentation is Weiser's program slicing [64].

This segmentation process will be aided by those forms of documentation that highlight the control structure of the program. The use of branching and hierarchical spatial arrangements highlight the control structure. In sequential forms, indenting is used to highlight the control structure.

Having segmented the program, the programmer begins to semantically reconstruct it. This part of the model is similar to a model proposed by Brooks [15]. Semantic reconstruction includes at least the processes of hypothesis generation, verification, and synthesis. These processes are not necessarily performed in strict sequence, but often as an iterative cycle. This is especially true as higher-level abstractions of the program are synthesized from comprehending lower-level functions.

Hypothesis generation. In hypothesis generation, the programmer develops a notion of what a particular segment of code does. This hypothesis can be drawn from comments embedded in the program or can be based on recognizing a particular pattern ("plan" according to Soloway and Ehrlich [59]) in the code. This process is based on recognizing plan cues in the program segment ("beacons" according to Brooks [15]) and using them to deduce the segment's function. For instance, the mnemonic variable names SUM, COUNT, and AVG are cues suggesting the hypothesis that a mean is being computed.

Hypothesis generation will be aided by documentation formats that display plan cues such as mnemonic variable names and statements that alter a variable's value. These types of cues can be provided in all types of spatial arrangements. However, we believe that constrained language is the best symbology for highlighting these cues, because they are typically learned in the context of an implemented program, and constrained language most closely resembles this form.

Verification. Having developed a hypothesis about what a particular segment of code does, the next step is to verify that the hypothesis is correct. Basili and Mills [5] formally verified program segments by proving their correctness. The typical approach of many programmers is to hand simulate the function of the segment. Other programmers are satisfied that they understand the function of a segment if, on closer inspection, it continues to match a pattern they recognize as performing the particular function. If the hypothesis cannot be verified, additional hypotheses must be formed which may alter hypotheses made about segments other than the one under consideration. Upon verification, the hypothesis is bound to the segment in the programmer's comprehension process [15, 65].

Hypothesis verification requires working through the subprogram using either formal logic or some form of simulation. A constrained language is probably the most effective symbology for conducting this exercise, because of its closer resemblance to statements as they would be represented in either formal logic or the final program. A branching or hierarchical arrangement may provide the visual cues that help the programmer bo the scope of that portion of the program which must be verified.

Synthesis. After the hypotheses have been verified for the lowest-level segments constituting a higher-level

segment of the program, the programmer synthesizes the higher-level function from the functions now bound to the segments. This process is also probably cue-driven, since programmers attempt to use the lower-level functions they have recognized as cues to identify higher-level functions.

These higher level functions will be represented as a schema in long-term memory. A schema is a template against which lower-level pieces of the function are matched, much as in putting together a jigsaw puzzle when one can look at the picture on the cover of the box. When there is a sufficient match between the lower-level functions and the slots in the schema, the programmer can represent the combination of the lower-level functions by the label of the higher-level function and think about the program in more powerful ways [24].

This synthesis process can be aided by embedded comments at appropriate levels in the program. Unfortunately, many programmers systematically disregard such comments [42], because they believe from experience that comments are typically not updated with subsequent changes to the code. The function of one portion of the program may be synthesized before an attempt is made to generate hypotheses for another section of the code. This process is repeated iteratively until the highest-level function of the program has been synthesized.

Synthesis may be enhanced by documentation presented in natural language. A novice perusing the computer science literature will quickly conclude that information about computer programs is largely verbal rather than pictorial. Even material referred to as graphic information about computer programs consists of verbal material encased in boxes that are connected by lines. Paivio [45] has argued for a dual encoding scheme for mental representation that consists of both verbal and imaginal components. The verbal representation of programs would appear to result in the cognitive processing of primarily verbal information. The low imagery words characteristic of the verbal descriptions of most programs would not appear to stimulate processing in Paivio's proposed imaginal system that would be as strong or complete as the verbal processing already underway [7]. Thus, without better evidence to the contrary, it seems reasonable to conclude that the primary processing of programming information is verbal/linguistic. When comprehending a specification, a programmer is trying to compile the function of the program into a form more like the verbal/linguistic form most typical of its cognitive representation and processing. Thus, there will be less translation to this form from information presented in natural language than from a constrained language or ideograms.

From the above discussion, we can deduce the following hypotheses:

Hypothesis 1. Comprehending computer programs is aided most by documentation presented in a constrained language during those tasks that involve decomposing the program and performing microlevel tasks on the module.

Hypothesis 2. Natural language is more efficient for synthesis tasks.

Hypothesis 3. On tasks where control flow information is important, such as segmentation and hypothesis verification, branching and hierarchical arrangements will be the most beneficial.

Using Program Slicing for Software Comprehension

Filippo Cutillo (*) Filippo Lanubile (**) Giuseppe Visaggio (**)

(*) Formatica s.r.l.
Via Amendola 162/1, 70126 Bari, Italy

(**) Dipartimento di Informatica
University of Bari
Via Amendola 173, 70126 Bari, Italy
fax: +39-80-263196 email:giuvis@vm.csata.it

1. Introduction[1]

Comprehending a program is ahead of a number of activities in the software life cycle. In fact, you must necessarily achieve a clear understanding to test, modify or reuse a program. This is a crucial problem when you must deal with a large program which lacks for the quality characteristic of understandability.

Program comprehension requires the capacity of recognizing the domain functions into a program which could be foreign to the occasional maintainer. This is a complex task because the domain functions are often spread over the source code, as shown in [9]. In [7], the recovery of function abstraction was based on a structured approach, called stepwise abstraction, which iteratively rewrites a program into a design language, starting from elementary one-in/one-out structures until to arrive to the higher abstractions which explain the program behavior.

From a cognitive point of view, the stepwise abstraction is coherent with the bottom-up theory of program understanding [5], where programmers build chunks of information which expand the limited capacity of the short-term memory. On the other hand the top-down theory [4] says that program understanding is expectation-driven, i.e. it is based on the expectation of domain concepts in the program which can be confirmed, refined or rejected. This approach is mostly applied by expert programmers when they learn an unknown program, because their experience suggests a number of hypotheses to verify. On the contrary, novice programmers tend to concentrate first on details because their expectations are too many few [1].

We propose program slicing as a technique of software comprehension which is coherent with the top-down theory. Program slicing has been previously applied to various activities such as debugging [12], parallel processing [13], module integration [8], evaluation of module cohesion [10], testing [6] and modification [6]. Our objective is to use program slicing to extract conceptual functions from an unfamiliar program. In order to achieve this goal we use a partial knowledge of the application derived from both the maintainer expertise and the reverse engineering of data.

This paper moves from a real experience, described in [3], where design recovery was applied to a banking application system. Design recovery [2] is a reverse engineering process which takes information not only from source code but also from other external sources, like the domain knowledge or the programmer expertise. Our approach divided the process in two phases. The former, data analysis, focused on the recovery of data abstraction until to reconstruct a full data model. The second phase, procedure analysis, focused on the recovery of function abstraction. The information sources to the procedure analysis phase come from the source code, the maintenance programmer expertise, and the knowledge acquired from the data analysis phase which suggests the probable presence of a number of functions in the application. For example, if a file contains the account balances and the program accesses the file to put information, then the program must be examined by looking for the function which computes the account balance.

The procedure analysis phase was based on the stepwise abstraction. Although the technique was successfully applied to a limited portion of source code, the process was greatly manual. Since too much time was spent, we looked for an alternative way of recovering the function abstraction which could much more exploit the expectations derived from experiencing the program and from the analysis of data. In fact, we believe that the information which results from the data analysis phase allows a novice programmer to acquire enough knowledge to become expert for that application because the reconstructed data model contain all the information

[1]This work has been supported by "Progetto Finalizzato Sistemi Informatici and Calcolo Parallelo" of CNR under grant no.90.00785.PF69.

for describing the input and output of the business functions.

The following section briefly summarizes the program slicing technique, the third section describes the extraction criteria and the last section suggests some conclusions and describes the future work.

2. Program slicing

Program slicing is a decomposition method introduced by Weiser [11], [13]. It is based on the observation that we are often interested to only a portion of the program behavior, as in the debugging and modification tasks. So, program slicing isolates that portion, by analyzing the data flow and the control flow of the program. In order to make automatic this capacity of projection, the behavior of interest is formally specified. The specification, called *slicing criterion*, takes the form $<i, V>$, where i is a statement and V is a subset of the program variables. A *slice* S of a program P, defined on a slicing criterion $C=<i,V>$, is an executable subset of P containing all the statements which contribute to the values of V just before statement i is executed. In [12] is reported an experiment which shows that programmers implicitly use slices when debugging unknown programs. The problem is that slices are often scattered through the entire program, making difficult the task.

In order to capture all the computation which is relevant for a given variable, the concept of program slicing has been extended in [6]. A *decomposition slice*, *S(v)*, is defined as the union of all the program slices on the variable v starting from the output statements and the last program instruction. So, while a program slice depends on a variable and a statement number, a decomposition slice depends only on a variable. Decomposition slices form a lattice based on the definition of the binary relation *strong dependence*. Impact analysis can be done by exploiting the algebraic properties of the lattice of decomposition slices.

3. Application to the recovery of function abstraction

We make the following assumptions:

a. There is a working program which is part of the application portfolio of a given organization.
b. There is a knowledge about problem and application domain which makes it possible to make hypotheses on the existence of functions inside the program.
c. The data analysis phase has produced a data model which describes the entity types, the relationships among them, and the entity attributes. There is also a traceability matrix between data model and source code.

The extraction criteria which we propose aim to intercept the conceptual functions, hidden inside the source code.

3.1 Function extraction

For each expected function f_i:

1) Give it a meaningful name.
2) Define the input data $IN_i = (id_{i1}, id_{i2}, \ldots, id_{in})$ which the function needs, and the output data $OUT_i = (od_{i1}, od_{i2}, \ldots, od_{im})$, which it yields. It is reasonable expecting that the data would be in the data model obtained from the data analysis phase. In the opposite case, you must locate the data as internal variables of the program or complete the data model with the missing data.
3) Extract from the source code the decomposition slice $S(OUT_i)$. It contains all the program statements which influence, both directly and indirectly, the output production.
4) Extract from the source code the decomposition slice $S(IN_i)$. It contains all the program statements which influence, both directly and indirectly, the input production.
5) Prune all the statements from $S(OUT_i)$, which contribute only indirectly to yielding the output of the function, because they are dedicated only to obtain the necessary input. A first hypothesis for cutting away the unnecessary statements assumes that the function can be obtained by deleting from the output slice the statements which belong also to the input slice: $f_i = S(OUT_i) - S(IN_i)$.

3.2 Subfunction extraction

A function could contain other functions, at a lower abstraction degree, which are bound by a composition relation: $f_i = f_{i1} \circ f_{i2} \circ \ldots \circ f_{in}$. If the function is complex then the nested subfunctions must be recovered.

The process of subfunction extraction must be preceded by a preliminary data analysis so that the following hypotheses can be made: (1) the codomain of the subfunction is strictly contained into the codomain of the compound function, (2) the codomain of the subfunction is equal to the codomain of the compound function, or (3) both the subfunction domain and codomain are equal to those of a subfunction which was previously recovered.

Case 1: $OUT_{ij} \subset OUT_i$

1) Give a meaningful name to the subfunction.
2) Extract from the source code the decomposition slice $S(OUT_{ij})$. It contains all the program

statements which influence, both directly and indirectly, the subfunction output.

3) Make an hypothesis on the subfunction input. It could be a proper subset or be equal to that of f_i: $IN_{ij} \subseteq IN_i$.
4) Extract from the source code the decomposition slice $S(IN_{ij})$. It contains all the program statements which influence, both directly and indirectly, the subfunction input.
5) Prune all the statements from $S(OUT_{ij})$, which are not essential for the subfunction f_{ij}. Also in this case there is a slice subtraction: $f_{ij} = S(OUT_{ij}) - S(IN_{ij})$.

Case 2: $OUT_{ij} \equiv OUT_i$

1) Give a meaningful name to the subfunction.
2) Make an hypothesis on the subfunction input. Since the codomains are equal to one another, it is reasonable to expect that the domain of the subfunction is strictly contained in that of f_i: $IN_{ij} \subset IN_i$.
3) Extract from the source code the decomposition slice $S(IN_{ij})$. It contains all the program statements which influence, both directly and indirectly, the subfunction input.
4) Prune all the statements from $S(OUT_{ij})$, which are not essential for the subfunction f_{ij}: $f_{ij} = S(OUT_i) - S(IN_{ij})$.

Case 3: $IN_{ij} \equiv IN_{mn}$ and $OUT_{ij} \equiv OUT_{mn}$

This case must be preceded by the application of either case 1 or case 2 to isolate the subfunction f_{ij} :

1) Verify if the subfunction f_{ij} is equal to f_{mn}, a subfunction which was previously recovered. In accordance with the mathematical definition, we regard functions as static relations between arguments and results. So, two functions can be equal if they contain the same ordered pairs, even though the algorithms for computing the result from argument are different. Since the question is undecidable, the verification must be manual.
2) If the subfunctions are equal there is a duplication of function. The problem must be recorded so that other programmers can recognize the redundancy in the code.

4. Conclusions and future research

At the current date, the use of program slicing for recovering the function abstraction has been successfully applied only on programs written by students.

The observed times for extracting functions have encouraged us to conduct an empirical evaluation, in the form of a pilot experiment in the banking domain. The samples will be two COBOL programs: the former is a TP program of 22000 LOCs which manages information about clients; the latter is a batch program of 15000 LOCs which builds a report containing the history of the transactions between the clients and the bank.

In order to support slicing, we intend to use a commercial tool, VIA/Renaissance of VIASOFT, which makes it possible to analyze COBOL systems on MVS/XA operating systems.

References

[1] B.Adelson, "Problem Solving and the Development of Abstract Categories in Programming Languages", *Memory & Cognition*, vol.9, 1981, pp.422-433.

[2] T.J.Biggerstaff, "Design Recovery for Maintenance and Reuse", *IEEE Computer*, July 1989.

[3] G.Como, F.Lanubile, G.Visaggio, "Design Recovery of a Data-Strong Application", in *Proceedings of the Third International Conference on Software Engineering and Knowledge Engineering*, 1991, pp.205-212.

[4] T.A.Corbi, "Program understanding: Challenge for the 1990s", IBM Systems Journal, vol.28, no.2, 1989, pp.294-306.

[5] B.Curtis, "Cognitive Issues in Reusing Software Artifacts", in *Software Reusability*, vol.II: *Applications and Experience*, T.J.Biggerstaff, and A.J.Perlis (Eds), Addison-Wesley, Reading, MA, 1989.

[6] K.B.Gallagher and J.R.Lyle, "Using Program Slicing in Software Maintenance", *IEEE Transactions on Software Engineering*, vol.17, no.8, August 1991.

[7] P.A.Hausler, M.G.Pleszkoch, R.C.Linger, and A.R.Hevner, "Using Function Abstraction to Understand Program Behavior", *IEEE Software*, January 1990, pp.55-63.

[8] S.Horwitz, T.Reps, and D.Binkley, "Interprocedural Slicing Using Dependence Graphs", in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988, pp.35-46.

[9] S.Letovski, and E.Soloway, "Delocalized Plans and Program Comprehension", *IEEE Software*, May 1986, pp.41-49.

[10] L.Ott, and J.Thuss, "The Relationship Between Slices and Module Cohesion", in *Proceedings of the 11th International Conference on Software Engineering*, 1989, pp.198-204.

[11] M.Weiser, "Program Slicing", in *Proceedings of the Fifth International Conference on Software Engineering*, 1981, pp.439-449.

[12] M.Weiser, "Programmers Uses Slices When Debugging", *Communications of ACM*, vol.25, no.27, July 1982, pp.446-452.

[13] M.Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, vol.SE-10, no.4, July 1984, pp.352-357.

AUTOMATED SOFTWARE MAINTENANCE USING COMPREHENSION AND SPECIFICATION

David A. Dampier
Luqi
Naval Postgraduate School
Monterey, California 93943

ABSTRACT

As software becomes more complex, more sophisticated software development and maintenance methods are necessary. These methods can also be used when dealing with old software systems. Program comprehension methods are used to extract a specification from an old software system, and our methods are then be used in forward engineering of the new software system. In the Computer-Aided Prototyping System (CAPS), quickly built and iteratively updated prototypes of the intended system are demonstrated to the user. As these updates occur, a formal mechanism must be developed to automatically integrate these changes into the existing prototype. This paper formalizes the update/change integration process and extends the idea to multiple changes to the same base prototype. Applications of this technology include: automatic updating of different versions of existing software with changes made to the base system; integrating changes made by different design teams during development; and performing consistency checking after integration of seemingly disjoint changes to the same software system.

I. INTRODUCTION

Maintaining sophisticated software is difficult. Maintaining old software is more difficult. In many cases, there is software in the field performing valuable tasks for which there is no written documentation. These pieces of software are prime candidates for *reengineering*. Reengineering consists of three activities; reverse engineering, restructuring and forward engineering. This research deals with the last of these activities, forward engineering. Reverse engineering uses program comprehension methods to produce a domain model, requirements and a specification. Given a domain model, the change-merge model outlined in this paper can be used in developing the new system.

Rapid Prototyping with automated tools makes the requirements conform more closely to the real needs of the users. An appreciable part of the maintenance activity is carried out in terms of changing and updating the prototype rather than the production code for the intended system. This is useful because the prototype description could be significantly simpler than the production code if the prototype is expressed in a notation tailored to support modifications, and the software tools in the computer-aided prototyping environment can help carry out the required modifications rapidly[3]. Prototyping software using tools decreases forward engineering time and increases future maintainability, because it reduces customer dissatisfaction with the delivered system. The designers construct/change prototypes of the intended systems quickly to meet the customer's desires during the requirements phase. The designers need automated tools which will allow these changes to be made to a base version of a software prototype as well as automatically propagated through multiple versions of the prototypes and automatically combine changes made to different versions of the system possibly by different people. Formal models are the keys and foundations for building such automated tools. Current technology provides formal models for tools which allow this only on a limited scale[1].

The goal of our research is to develop a tool for the CAPS[3] system which will support automatic merging of different versions of a prototype. We have already developed a model which shows that it is possible to correctly perform a merge operation in most cases[2]. This paper formalizes the change process for the Prototyping System Design Language (PSDL), a design based language written specifically for CAPS, and uses this formalization to strengthen our merging model.

II. CHANGING PROTOTYPES IN PSDL

The Prototype System Description Language (PSDL) is the prototyping language associated with CAPS. It was created to provide the designer with a simple way to abstractly specify software systems and produce an executable prototype of the system. A PSDL program is a set of PSDL operators and data types, containing zero or

This research was supported in part by the National Science Foundation under grant number CCR-9058453 and in part by the Army Research Office under grant number ARO-145-91.

more of each. PSDL operators and types consist of a specification and an implementation. The specification defines the external interfaces of each operator through a series of interface declarations, provides timing constraints, and describes the functionality of the operator through the use of formal and informal descriptions. The implementation can either be in PSDL or Ada. PSDL implementations are data flow diagrams augmented with a set of data stream definitions and a set of control constraints.

A current focus of our research is formalization of the change process in PSDL. In order to discuss the merging of changes made to a prototype, we must first provide a model for changes to PSDL prototypes.

PSDL prototypes are iterative approximations to a software system. If S is the intended final version of the software system, then each successive iteration of the prototype can be viewed as an element of a sequence S_i where $\lim_{i \to \infty} S_i = S$. Each prototype S_i is modelled as a graph $G_i = (V_i, E_i, C_i)$, where:

A. V_i is a set of vertices. Each vertex can be an atomic operator or a composite operator modelled as another graph.
B. E_i is a set of data streams. Each edge is labelled with the associated variable name. There can be more than one edge between two vertices. There can also be edges from an operator to itself, representing state variable data streams.
C. C_i is a set of timing and control constraints imposed on the operators in version i of the prototype.

The prototype designer repeatedly demonstrates versions of the prototype to users, and designs the next version based on user comments. The change from the graph representing the ith version of the prototype to the graph representing the (i+1)st version can be described in terms of graph operations by the following equations:

(1) $S_{i+1} = S_i + \Delta S_i$

The change $\Delta S_i = (VA_i, VR_i, EA_i, ER_i, CA_i, CR_i)$ is a set of subsets where:

$\Gamma_{i+1} - \Gamma_i = \Gamma A_i$: The set of components to be added to S_i.

$\Gamma_i - \Gamma_{i+1} = \Gamma R_i$: The set of components to be removed from S_i.

Γ can represent V, E or C.

The + operation in (1) above is defined as follows:

$$\Gamma_{i+1} = \Gamma_i \cup \Gamma A_i - \Gamma R_i$$

The following example illustrates a change to the vertex set of a composite operator:

$V_i = \{A, B, C, D\}$ $\quad VA_i = \{E\}$ $\quad VR_i = \{C\}$

$$V_{i+1} = V_i \cup VA_i - VR_i = \{A, B, C, D\} \cup \{E\} - \{C\} = \{A, B, D, E\}$$

IV. MERGING PSDL PROTOTYPES

Merging different variations of a program is useful in performing automatic maintenance of software systems. In very large software systems, it is common for different variations to evolve from the base system. If the system designer discovers a fault in the base version of the system, it would be desirable to have the capability to automatically apply that change to all of the variations currently in use. The merging process must be able to apply the change to the common parts of each variation without affecting the functionality peculiar to each one.

In [10], a definition of merging two compatible extensions of a software system was given as follows:

> If the software systems are represented using sets, then the result of merging two extensions, A & C of a base version B is defined as:

$$M = A[B]C = (A - B) \cup (A \cap C) \cup (C - B)$$

Consider a version, S_i, which has been changed in two different ways, via Δ_A and Δ_B. The result of these two changes is S_A and S_B respectively. Now let us define the (i + 1)st iteration as

(2) $$S_{i+1} = S_A[S_i]S_B = (S_A - S_i) \cup (S_A \cap S_B) \cup (S_B - S_i)$$

The components of S_{i+1}; V_{i+1}, E_{i+1} and C_{i+1} can be defined similarly.

The following example illustrates merging the vertex sets of two different variations:

$V_i = \{A, B, C, D\}$ $V_A = \{A, B, D, E\}$ $V_B = \{A, C, D, F\}$

$$V_{i+1} = (V_A - V_i) \cup (V_A \cap V_B) \cup (V_B - V_i) =$$
$$(\{A, B, D, E\} - \{A, B, C, D\}) \cup (\{A, B, D, E\} \cap \{A, C, D, F\}) \cup$$
$$(\{A, C, D, F\} - \{A, B, C, D\}) = \{A, D, E, F\}$$

This simple example illustrates the merging of two changes which do not conflict with one another. The result of merging two conflicting changes could be an inconsistent program, causing a conflict which has to be resolved either automatically or manually by the designer.

V. Conflict Resolution

Conflicts arise when two changes affect the same portion of the prototype in different ways. Some examples of conflicts are as follows:

1. One change adds an output edge to a vertex A, while another change removes vertex A from the prototype. In this case, automatic resolution of the conflict is not yet possible, so the system would have to notify the designer that a conflict has occurred and give him/her the opportunity to resolve it.

2. Two changes assign different timing constraint values to the same operator, i.e., (max_exec_time, F, 50ms) and (max_exec_time, F, 40ms). In this case, the conflict can be handled automatically, as any operator which executes in under 40ms would certainly execute in under 50ms. In situations where different maximum execution times have been assigned, the minimum value can be chosen. This is also true for the latency, maximum response time and finish within timing constraints. The minimum calling period timing constraint can be merged using the maximum of the two values. Different periods for the same operator result in a conflict which has to be resolved by the designer. Different control constraints for the same part of the prototype in different changes can also result in a conflict. Some of these conflicts can be resolved automatically. Current work is addressing methods for automatic resolution of conflicts.

VI. Conclusions

Tool support for manipulating and combining specifications is especially important for rapid prototyping. We are currently implementing the method presented here to evaluate its effectiveness in practical contexts. We are also conducting theoretical studies to evaluate its limitations and to discover improvements. The method described here works correctly whenever the functions computed by the operators are one to one. In the general case, a global analysis of the system may be necessary to ensure that the functions computed by the operators do not interfere. For a more detailed discussion of the reasons for this, see [4]. Some issues to be considered in future work are treatment of data types and component specifications, and the detection/diagnosis of semantic interference between modifications.

LIST OF REFERENCES

1. Horwitz, S., Prins, J., and Reps, T., "Integrating Non- Interfering Versions of Programs", Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, New York, 13 - 15 January 1988.

2. Dampier,D., A Model for Merging Different Versions of a PSDL Program, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1990.

3. Luqi, "Software Evolution Through Rapid Prototyping", IEEE Computer, May 1989.

4. Berzins, V. "Software Merge: Semantics of Combining Changes to Programs", Submitted for publication in ACM Transactions on Programming Languages and Systems, 1990.

An Integrated Representation for Program Comprehension†
(Position paper on program comprehension)

Mary Jean Harrold and Brian Malloy
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
(803) 656-0809
harrold@cs.clemson.edu

Program maintenance is an expensive process where an existing program is modified for a variety of reasons, including correcting errors, adapting to different data or processing environments, enhancing to add functionality and altering to improve efficiency. The problem is especially difficult since the maintainer is rarely the author of the code. Thus, an important goal of maintenance tools is to assist the maintainer in understanding different aspects of a program. To provide efficient tools for program maintenance, an integrated program representation that contains the required information is needed. We support other researchers'[6] claims that a language-independent program representation, the *program dependence graph* (PDG)[1,5,7] is extremely useful for program understanding. The PDG unifies both control- and data-dependence uniformly since it contains both a control dependence subgraph and a data dependence subgraph, and techniques such as slicing can be performed on it. The main problem with the PDG is that is contains no control- or data-flow information and thus, another representation, the control flow graph is required. The control flow graph is used to construct the PDG, and it is continually accessed whenever control- or data-flow information is required. Thus, for large programs, this technique is expensive in both space and time since both a PDG and a control flow graph are required for each procedure.

Our approach is to develop an integrated program representation and use it to provide many different maintenance tools to assist in program understanding. Previously, we presented our *unified interprocedural graph*[2,4] that integrates control- and data-flow along with control- and data-dependency information into a single representation. However, to get this interprocedural information, we require both the control flow graphs and the PDG's of each procedure involved. We have extended this research to provide a single integrated representation for each procedure [3] that contains the data- and control- flow information provided by the individual graphs. Our integrated representation is an extension of the PDG that eliminates the need for the control flow graph of a procedure in most cases. Thus, we use our extended PDG as our only procedure representation from which we abstract information to construct the unified interprocedural graph and for tools assist in program understanding.

We have extended the PDG so that it contains control-flow, data-flow, control-dependence and data-dependence in a single representation. For program without GOTO statements, we efficiently build our extended PDG from the abstract syntax tree produced by the compiler[3], without requiring the program's control flow graph. For programs without GOTO statements, the structure is recognizable by the parser. Thus, we compute the exact control dependencies for the program. Additionally, we order the nodes in

† This work was partially supported by NSF under grant CCR-9109531 to Clemson University.

the PDG so that we can always identify the control flow predecessors of any node in the PDG. Thus, we perform data flow analysis directly on the PDG. In preliminary studies of over 2000 functions found in the X Consortium source distribution, only 23 functions actually contain GOTO statements. Thus, this technique for constructing the extended PDG will be widely applicable. For programs with GOTO statements, we construct a partial PDG and use the control flow graph to complete construction. However, we still order the nodes according to source code statements and perform the data flow analysis directly on the extended PDG. Building the extended PDG at this stage of compilation provides significant savings over previous methods that require the intermediate code, the control flow graph, and the post-dominator tree to finally construct the PDG.

With our extended PDG, we provide powerful tools for program understanding. Along with previously developed tools, such as slicing, we can efficiently display multiple views of the program. For example, with our single representation, we can illustrate the program's control flow, data flow, data flow sets and control dependencies to the user. Other previous techniques that provide multiple views including, control- and data-flow[8], use several program representations. We also provide new tools for program understanding such as automatic recognition of a program's control structures that are not obvious by code inspection. Additionally, once we recognize these control structures, we transform them to recognizable constructs such as *while* loops and *if-then* statements. We developed a set of templates that identify subgraphs of the extended PDG representing *while* loops, *if-then* statements and *if-then-else* statements.

References

1. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.
2. M. J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," *IEEE Transactions on Software Engineering, to appear.*
3. M. J. Harrold and B. A. Malloy, "Performing data flow analysis on the PDG," Technical Report #92-108 Clemson University, March 1992 .
4. M. J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," *Proceedings of Conference on Software Maintenance 1991*, pp. 138-147, October 1991.
5. S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345-387, July 1989.
6. S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," *Proceedings of 14th International Conference on Software Engineering*, pp. 392-411, May 1992.
7. D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe, "Dependence graphs and compiler optimizations," *Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 208-218, January 1981.
8. M. Platoff, M. Wagner, and J. Camaratta, "An integrated representation and toolkit for the maintenance of C programs," *Proceedings of the Conference on Software Maintenance-1991*, pp. 129-137, October 1991.

Automating Program Comprehension by Concept Recognition

Wojtek Kozaczynski and Jim Ning
Center for Strategic Technology Research (CSTaR), Andersen Consulting
100 South Wacker Drive, Chicago, Illinois 60606, U.S.A.

1. Program Comprehension

Program comprehension, the understanding of program source code, is essential to many software engineering and re-engineering activities such as maintenance, debugging, verification, renovation, migration, and design recovery.

Many tools have been developed to facilitate program understanding. With few exceptions, however, these tools are little more than browsers that present different text and graphical views of the code. For example, VIFOR [Raj90] graphically displays the *calling* and *cross reference* relations in FORTRAN programs. ESW, a COBOL re-engineering toolset developed by VIASOFT, extracts *structure charts* from PERFORM statements and produces *condensed source listings* to highlight "structurally significant" parts of programs. The program understanding tool prototyped at IBM [Cle88] supports cross referencing, calling relation, and control flow and data flow presentations. More recently, LaSSIE [DBSB91] demonstrated the need for capturing architectural and conceptual information (in addition to code-level information) to support program understanding. LaSSIE provides a classification-based representation and retrieval framework for the user to encode, browse, and query the information about high-level concepts; but these concepts must be obtained and encoded manually.

Browsing-based tools may assist the user in exploring hidden properties of programs. It remains still the user's responsibility, however, to reason about the meaning of the program under analysis. Moreover, as the user builds understanding of individual concepts, existing tools don't help generalize the knowledge to automate the understanding of similar concepts in the future. In order to provide this type of assistance to program maintainers, it is necessary to automate the program comprehension process.

2. Concept Recognition

What does it mean to comprehend a program? Syntactically, a program is a sequence of text strings. Semantically, however, it contains many *language* and *abstract* concepts. Language concepts are variables, declarations, modules, statements, etc., and are defined by the coding language. Abstract concepts represent language-independent ideas of computation and problem solving methods. Abstract concepts can be further classified into:

- *programming concepts* that include general coding strategies, data structures, and algorithms;
- *architectural concepts* that are associated with interfaces to execution architecture components such as operating systems, transaction monitors, networks, databases, etc.; and
- *domain concepts* that are application or business logic functions implemented in the code.

Language concepts can be "recognized" automatically by parsing. A parser generator takes *a language model* and *a language grammar* and generate a parser for the language. This parser can then be used to process program text and produce program abstract syntax trees (ASTs). Nodes in an AST can be seen as language concepts recognized by the parser.

Abstract concepts are not equally easy to recognize. Two questions must be addressed in order to recognize them, namely, *what* types of concepts to recognize and *how* they should be recognized. A concept classification hierarchy, called *a concept model*, answers the *what* question. Like a language model, it establishes the ISA hierarchy between concept classes for which instances are expected in the target programs.

The *how* question cannot be answered with a purely syntactic approach because the abstract concepts are not tightly related by syntactic structures. Instead, they are connected by semantic relationships such as: control-flow relationships, data-flow relationships, calling relationships, etc. For example, a READ-AND-PROCESS concept that represents a function that sequentially reads and processes records in a sequential file would require two of its component concepts (sub-concepts), the READ-RECORD concept and the PROCESS-RECORD, to follow each other on a control flow path. We call these semantic relationships and other requirements on the sub-concepts of abstract concepts *constraints*.

A specification of how to recognize an abstract concept must, therefore, contain information about: 1) components or sub-concepts of the concept, and 2) constraints on and among the sub-concepts. We encode this information in the form of *concept recognition rules*. For historical reasons [Wil87, JS85, Let88, Nin89, Har91], we also use the term *plans* to refer to recognition rules. The following is the structure of a recognition rule/plan:

```
plan C
    consists of   S1, S2, ..., Sm
    such that     E1, E2, ..., En
```

where C describes the abstract concept to be recognized by this plan; S_1, S_2, ..., S_m describe the sub-concepts; and E_1, E_2, ..., E_n are logical expressions (constraints). Both C and S_1, S_2, ..., S_m are called *pattern descriptions* of concepts.

Assume that C_1, C_2, ..., C_m are language and abstract concepts already recognized in a program P. Let us also define σ, called a set of *bindings*. Each binding in σ is a mapping from a pattern variable v to a value l, represented by a tuple $<v, l>$; $\sigma(v) = l$. More generally, we use $\sigma(S)$ to describe the instantiation of all the pattern variables in a pattern description S to values defined by σ.

With these definitions, we say that a plan is *applicable* to P at C_1, C_2, ..., C_m iff there exists a σ, such that:

1) $\forall$ i: $1 \le i \le m$ $C_i \equiv \sigma(S_i)$; and
2) $\sigma(E_1)$ & $\sigma(E_2)$ & ... & $\sigma(E_n)$ = TRUE i.e. logical expressions instantiated by σ can be evaluated to truth values.

When these two conditions are met, the plan will be applied to P on top of C_1, C_2, ..., C_m to create a new concept instance $C' = \sigma(C)$.

To illustrate the above in some detail, let us consider how we could specify a concept of adding new copies of a book in a hypothetical book inventory system. The top-level concept, ADD-NEW-COPIES, can be written as follows:

```
plan ADD-NEW-COPIES(ADD-TO-BOOK : @rec, QUANTITY : @new-copies)
  consists of  book-rec : BOOK-RECORD( RECORD-NAME : @rec,
                                       IN-STOCK : @amount)
               test : EQUAL-OP(OP1 : @trans-type, OP2 : 1)
               add-copies : ACCUMULATE(TO : @amount, FROM : @new-copies)
               update : WRITE-FILE(FROM : @rec)
  such that    CONTROL-DEP(add-copies, test)
               DATA-DEP(update, add-copies, @amount)
```

In this plan, @rec, @new-copies, and etc., are pattern variables, and 1 (in the second sub-concept following OP2) is a constant. The plan states that the ADD-NEW-COPIES concept has four sub-concepts:

book-rec, test, add-copies, and update. book-rec is an abstract, problem domain concept of type BOOK-RECORD. This record contains information about the book such as the amount of copies in stock (amount). The other three sub-concepts are programming concepts: test checks whether the current transaction is an add new copy transaction; add-copies adds the number of new copies to the existing amount; and update writes the record into the book inventory file. In addition, the constraints require add-copies to be control dependent on test and update to be data dependent on add-copies with respect to data item amount. If instances of sub-concepts can be found in the program and they satisfy the constraints, an instance of ADD-NEW-COPIES concept is created:

```
σ(ADD-NEW-COPIES(ADD-TO-BOOK : @rec, QUANTITY : @new-copies)
= ADD-NEW-COPIES(ADD-TO-BOOK : σ(@rec), QUANTITY : σ(@new-copies))
```

Notice that the recognition of an ADD-NEW-COPIES concept requires other plans to recognize its sub-concepts.

3. Status

For the last few years we have developed two prototype program concept recognition systems. The works has been completed as a part of the software re-engineering program at Andersen Consulting's Center for Strategic Technology Research (CSTaR). Currently, we are developing a next layer of re-engineering functionality such as: *program transformation, program segmentation,* and *concept browsing.* All of these tools use the concept recognition engine and aim at increasing the practical utility of the software understanding systems.

References

[All90] Allemang, D., *Understanding Programs as Devices*, Ph.D. thesis, Ohio State University, 1990.

[BHW89] Biggerstaff, B., Hoskins, J., and Webster, D., "DESIRE, A System for Design Recovery," *MCC Technical Report* STP-081-89, April 1989.

[Cle88] Cleveland, L., "A User Interface for an Environment to Support Program Understanding," *Conference on Software Maintenance*, Phoenix, Arizona, October 1988.

[DBSB91] Devanbu, P., Brachman, R., Selfridge, P., and Ballard, B., "LaSSIE: A Knowledge-Based Software Information System," *Communications of the ACM*, May 1991.

[EKN91] Engberts, A., Kozaczynski, W., and Ning, J. Q., "Concept Recognition-Based Program Transformation," *Conference on Software Maintenance*, Sorrento, Italy, October 1991.

[Har91] Hartman, J., Automatic Control Understanding for Natural Programs, Ph.D. thesis, University of Texas at Austin, May 1991.

[HN90] Harandi, M. T., and Ning, J. Q., "Knowledge-Based Program Analysis," *IEEE Software*, January 1990.

[JS85] Johnson, W. L., and Soloway, E., "PROUST: Knowledge-Based Program Understanding," *IEEE Trans. on Software Engineering*, 11(3), 1985.

[KLN91b] Kozaczynski, W., Letovsky, S., and Ning, J., "A Knowledge-Based Approach to Software System Understanding," *Sixth KBSE Conference*, September 1991.

[KNS92] Kozaczynski, W., Ning, J., and Sarver, T., "Program Concept Recognition," *Seventh KBSE Conference*, September 1992.

[Let88] Letovsky, S., *Plan Analysis of Programs*, Ph.D. thesis, Yale University, December 1988.

[Nin89] Ning, J. Q., *A Knowledge-Based Approach to Automatic Program Analysis*, Ph.D. thesis, University of Illinois at Urbana-Campaign, October 1989.

[Raj90] Rajlich, V., "VIFOR: A Tool for Software Maintenance," *Software - Practical and Experience*, January 1990.

[Ric81] Rich, C., "A Formal Representation of Plans in the Programmer's Apprentice," *Seventh IJCAI Conference*, 1981.

[Wil87] Wills, L., *Automated Program Recognition*, Master's thesis, MIT AI Lab, February 1987.

Conference on Software Maintenance 1992
Position Paper: Tools Facilitating Software Comprehension

David P. Olshefski

P.O. Box 704
Yorktown Heights, NY 10598

Claims

Our recent work has involved the development of a prototype program understanding system. Its function is to serve as a vehicle for exploring and testing ideas in the area of program understanding. The system, named PUNDIT, combines the use of statically collected semantic information with debugging capabilities in an attempt to help the programmer understand both the static nature and dynamic behavior of a program. Our experience has led us to the following conclusions concerning program comprehension systems:

1. Static and dynamic information should be integrated and presented together.
2. Presenting large amounts of data to the user in a useful manner is a more difficult problem than source code analysis.
3. Integration into current build processes and library systems is important.
4. Program comprehension systems should take advantage of existing compiler technology.
5. Performance is a key to user acceptance and this means, for now at least, that the use of a database to contain analysis information is unacceptable.

Architecture

The goal for PUNDIT was to design a responsive, lightweight, highly interactive tool for analyzing source code. After trial and error we came to the approach depicted in the accompanying figure.

The PUNDIT system is composed of two executables, the C semantic analyzer and the user interface. The C semantic analyzer, PCC, is simply a compiler front end designed to generate *extended* debug information. This extended debug information takes the form of a binary file and is thought of as a *supplement* to the debug information contained in the object module file. The binary file contains two types of information. The first is semantic information about control flow and data use and the second is information used by the user interface for display purposes (such as column position of a variable on a particular line). Although the current analyzer only supports C analysis, the format of the binary file is language independent. The binary-file produced by PCC is about the same size as the object module. PCC itself was written using a parser generator.

Although the analysis is not incremental, it's fast enough for the user to run as often as he would a compiler and, as expected, it is faster than the compiler since PCC only generates semantic information, not executable code. A slight modification to the make file allows a source file to be analyzed by PCC whenever it is recompiled, a process which should fit well with most existing library systems and environments that currently manage compilation and object files.

The user interface is a multi-threaded, multi-windowed OS/2 Presentation Manager application which presents both textual and graphical views. It gathers all existing information from the binary files, object modules, and executable files and attempts to present that information to the user in a useful manner. *Inter-module* information is resolved by the user interface in a manner similiar to the way a linker resolves external references between object modules. Therefore, one C source file can be modified and re-analyzed by PCC independently of any other source file.

The major advantage to the approach of storing information in flat files is that it eliminates the need for installing and using a database. Our experience has shown that semantic analysis of source code can generate a great deal of highly interrelated, light-weight information. Available database managers are not

designed well to handle those characteristics. Our experiments using a commercially available relational database manager, for example, showed unacceptable performance.

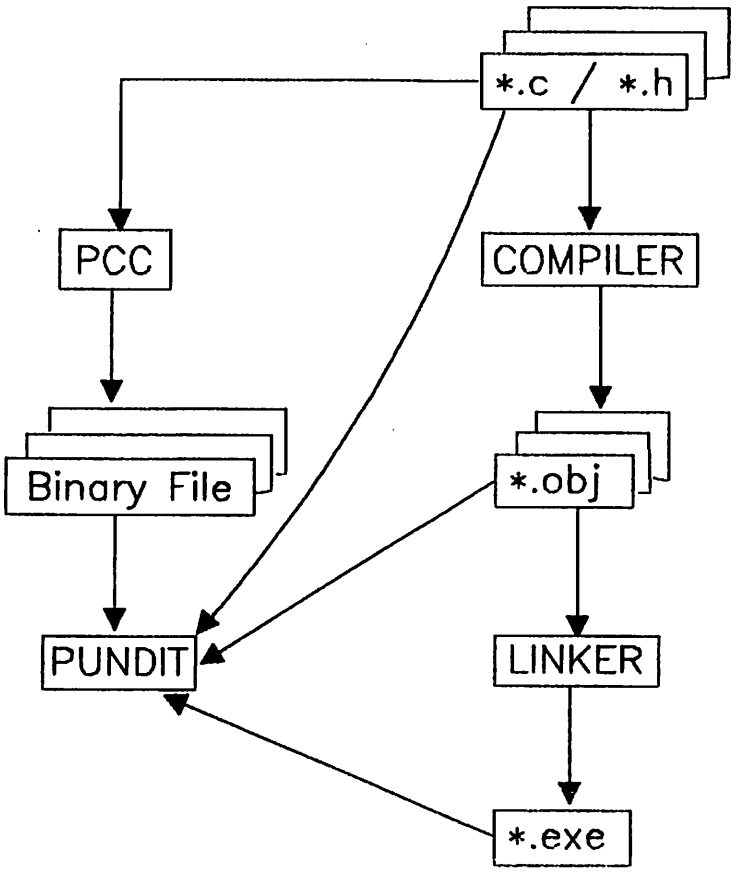

Figure 1. PUNDIT runs under OS/2 2.0 on the IBM PS/2.

Another advantage of this approach is that it makes use of the information already collected by the compiler. The separate analysis step is seen only as a necessity since current compilers do not provide *all* the necessary semantic information. If compilers eventually make this information available, the PCC component of PUNDIT can be discarded.

Major Functions Provided By PUNDIT

The following is a list of some of the major functions supported by the user interface:

- Display the high level structure chart
- Create a dynamic call graph during debugging
- Show the control flow graph for a function
- Animate the control flow graph and source code during program execution

- Textually display the type definitions for all types used in a source file
- Graphically display the static relationships between type structures
- Allow the user to graphically layout data structures during debugging
- Display a textual list of the functions contained in each source file
- Textually display the header file include nesting
- Display a static program slice
- Show symbol cross-reference information
- Set break points, view registers and storage

The above functions are realized in PUNDIT via a set of windows, or views. Each view has its own set of functions that operate on the data in its window. Each view is also tied to other views, usually by functions which link the data in one window to corresponding data in another. A user action in one view results in appropriate changes to other views. Typically, the user may click on anything that is visible and perform some function. It seemed important not to impose a particular mode of navigation on the user but rather leave the user interface flexible enough to allow the user to navigate in any direction he or she so chooses.

Not surprisingly, not all of the above views appear to be of equal benefit to the user. As is usually the case, those functions which perform something the user cannot do in a reasonable time are valuable. The high level structure chart is one example. It provides the user with a high level picture of the calling structure and global use within the program. Also of value are those functions performed by hand which can be automated. An example would be the graphical layout of data structures during debugging.

Although important for use in compiler construction and data flow analysis, the control flow graph, in and of itself, is not viewed by our users as very important. If a programmer wants to know the internals of a particular function, he or she will simply read the source code. In most cases, the user can determine the function's control flow on his or her own, thus the control flow graph view provides little in addition to the users own mental capabilities. It does provide benefit though, during debugging. The control flow graph can be set to animate program execution which allows the user to sit and watch how the program behaves with live data. It can also track which statements have been executed during a particular run thereby providing an automated test coverage capability. Both are functions which would require much work if done manually.

In general, it seems that the static views which are most valuable are those providing information at a higher level and the dynamic views which are most valuable are those providing more detailed information concerning the internals of a data structure or function.

Presentation Problems

Unlike a compiler writer who is attempting to perform deep analysis for purposes of optimization, we found that source analysis was not the major hurdle. Our difficulties arose when attempting to display large amounts of information to the user in a meaningful way. An example is displaying a large call graph, with hundreds of nodes. If some way is not found to simplify and summarize the graph, it can end up baffling and confusing the user. Our approach is to provide the user with extensive filtering and graph editing capabilities. The user can edit, trim and build the view to display only those nodes of interest.

In order to do this, graph displays had to be very fast and highly interactive. The graph display toolkit we used was developed by our colleague Vance Waddle and performs both graph layout and presentation. The quality of the PUNDIT user interface is due largely in part to using this full function graph display toolkit.

Whether PUNDIT is considered a debugger with static analysis capabilities or a program understanding system with debugging capabilities is irrelevant. We found that either one without the other is still only half the solution. Integrating static and dynamic information into one tool provides a much richer set of functions than having both in separate tools.

Recovering Application Knowledge from Imperative Code

Stephen B. Ornburn and Richard J. LeBlanc
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

A model relating rule-based representations of application knowledge and imperative programs encoding is introduced. This model, developed over the course of several experiments in reverse engineering, provides insight into the technical problems which must be solved when making explicit the conditions under which operations are performed. This model is particularly useful in its ability to distinguish between code supporting the program's functional behavior from code responsible for ensuring that various safety and liveness constraints are satisfied.

Introduction: Software generation can be viewed as the process of producing an imperative program by combining application knowledge, including business rules, with program control. The goals of a reverse engineering project often include reconstructing the application knowledge used in developing the program. This goal is difficult to achieve because much of this application knowledge is not explicitly represented in the implementation code. Furthermore, that knowledge which is explicitly in the code is organized so as to facilitate execution rather than comprehension. In other words, the knowledge is organized according to where and how it is to be used in the computation.

Any program's control mechanisms apply application knowledge to effect a computation. For example, in rule-based programs, including logic programs, an inference engine effects the computation by selecting and applying rules from a rule base. While imperative programs integrate application knowledge and program control into a single structure, a rule-based model can still provide useful insight into the program and its operation. More specifically, the imperative program can be related to a hypothetical "smart" rule-based system which does not waste time backtracking, recomputing values, or searching. Instead, the "smart" system picks the right rules, remembers the relevant portions of its own history, and represents information in convenient data structures.

The "smart" system tailors the control strategy to a particular computing problem. Some of the control decisions can be made at design time, while others are deferred until run time. Generally, those control decisions requiring the greatest intelligence are made at design time by the software engineer, who then integrates the results of those decisions into the implementation. At the same time, the software engineer also integrates into the implementation both the decision logic required for the residual control decisions and suitable knowledge representations. The set of residual program control mechanisms defines a software architecture and includes mechanisms for sequence control, data control, and memory management.

In many rule-based systems every rule potentially applies in every context, and complex guards can be required to ensure that rules are applied appropriately. An alternate approach, based on finite state program control, can be used to limit *a priori* the contexts in which a rule applies. Restricting the context in this way allows the guards to be simplified and, in turn, reduces the amount of data which must be maintained as part of program state.

Structuring application knowledge: A rule-based representation of application knowledge is more easily understood if the rules are organized in accordance with a state transition model. Even if the initially hypothesized state transition model only names the states and events without specifying them in detail, it can still guide subsequent reverse engineering. Reverse engineering can confirm or revise the initial hypotheses and then can go on to discover the additional detail need to complete the state and event definitions.

A typical specification provides a software engineer with a large volume of information; some of this information is explicitly included in the specification, but most of it is background information. This background information includes the definitions, assumptions, and domain theory needed to understand the specification, but may not be explicitly referenced by the specification. Often, much of this background information is assumed to already be in the possession of the software engineer.

When a specification is to be formalized, several types of formulae are required. For example, the specification will generally include a formula describing the function of the system by relating its inputs to its outputs. In addition, the specification can include

formulae relating the abstract values of the functional specification to lower-level interpretations, including program state. Also, the specification can include various temporal constraints limiting the run-time behavior of the computation. Temporal constraints are used to define various safety and liveness properties. For example, requirement that a user must supply the correct password before modifying a file is an example of a temporal constraint. The stipulation that a particular resource is not to be shared among processes is another example of a behavior which can be formalized in terms of temporal constraints. In principle, a set of production rules, formalized as a set of Horn clauses, can be derived from this specification and its supporting definitions. To be comprehensible, however, this rule base must be structured in some way.

One approach to structuring this rule base employs a state transition model. The state transition diagram is used to control the sequence of operations on a set of auxiliary variables and objects. A state in the model denotes an equivalence classes of states and is defined by an invariant. Events are also defined by invariants and can denote a combination of external and internal software events. Some application rules become transition rules describing how the system moves from one state to another in response to events. Other rules are classified as repair rules and are regarded as derived from a state invariant. Repair rules operate at a lower level than do transition rules and are responsible for modifying the program's actual state to ensure that it actually satisfies the invariant of the newly entered state. Further structure can be imposed on the rule base by dividing state among objects and limiting the application of a repair rule to a particular object. Yet even more structure can be imposed by introducing hierarchical, communicating state machines similar to those used in state charts.

A state transition model provides sequence control and, as a consequence, satisfies a number of temporal constrains but certain types of temporal constraints are hard to accommodate in this way. An alternative approach to the difficult constraints introduces communicating state machines: one machine proposes an initial schedule of events, a second modifies that schedule, and a third implements the modified schedule, possibly by dispatching commands to yet other machines. A simple example of such a model controls a pipeline of functional components. The synchronization of these functional components is controlled by an additional component which schedules operations and dispatching commands to the components of the pipeline, ensuring that they fire in the appropriate sequence. This type of control can be used for more complex constraints such as ensuring that a component releases all acquired resources even in the event of a fault which prematurely terminates the computation. While state machine models of this sort can depend on future knowledge and cannot always be directly implemented, they can provide a comprehensible model of the program.

Recovering application rules: Considerable code level analysis is required to recover application rules and organize them in terms of a state transition model. This type of reverse engineering effort must be guided by a strong set of expectations as to the structure of the model to be recovered. This expected model should identify the set of hierarchical, communicating state machines which the engineer expects to use use in structuring the rule base, and, as discussed above, these machines can negotiate the sequence in which operations are executed. The ability to model the parties to this "negotiation" as parallel, communicating state machines allows rules governing resource sharing, deadlock avoidance, or fault tolerance to be separated from the underlying functional behavior. These expectations provide important guidance to the reverse engineering process, and the software engineer should make conjectures as to the states, events, and transitions out of which these machines will be built even if they will be revised in light of code-level analysis.

Once the state-machine model has been hypothesized, the software engineer can begin to reconstruct the application knowledge used in the imperative program. Because of the design time evaluation of selected control components, information determining which operations are to performed and in which sequence is spread throughout the program text.

There are several important examples of "information spreading" which must be considered in future research. For example, software designs frequently fuse conceptually separate computational functions in order to eliminate large intermediate data structures or eliminate redundant computation. Similarly, a design will ignore the natural parallelism among a set of computational activities, replacing them with a single sequential program in which a schedule interleaving their execution has been hard wired. Most difficult of all to recognize, functional components will be textually interleaved with other code responsible for ensuring various safety and liveness constraints. Information spreading results from the integration of application knowledge and program control, but regardless of the types of information which have been interleaved, the recurring and difficult problem is that of recognizing and distinguishing among transition and repair rules; a task which we believe cannot be completed based on textual cues alone but also requires a prior understanding of the application domain.

As discussed above, limiting the context in which rules can be applied allows their guards to be simplified and also allows corresponding simplification of the program's state. This interleaving and simplification must be "undone" as part of recovering appli-

cation rules. These optimizations are troublesome to the reverse engineering effort because they obscure the global conditions under which a particular operation is executed. As part of reconstructing the rules, the software engineer must recognize the state transition models representing the various " parallel" computational activities and must associate operations with these various models. As has been discussed above, these parallel activities can relate not only to the software's functional behavior, but to nonfunctional aspects, including operation scheduling, data control and memory management, of it's behavior as well. This is an important modeling capability because in some situations, e.g., fault avoidance, fault handling, and synchronization, the decision logic can be come quite complex.

Because of interleaving, textually adjacent operations can be assigned to different models in a set of parallel models. When decomposing a block of code in this way, the software engineer must reconstructing the guards associated with the individual operations by first recognizing the local conditions under which the operation is executed and then proceeding to recognize progressively more global conditions.

The hardest part of reconstructing application rules is that of determining the implicit conditions represented by the program counter. The conditions under which an operation is performed include both relationships satisfied by the program's variables at the time the operation is executed and conditions which held at earlier times in the program execution thus allowing the program counter to reach the operation. While designing the program so that the program counter can implicitly represent complex conditions greatly simplifies the residual problems in program control and reduces the amount of information which must be explicitly represented in the program's data structures, the identity of the application rules is correspondingly obscured.

One approach to reconstructing application model transforms the imperative program into a set of iterative guarded commands, and the iterative guarded command can be used for finite state machines of the sort we have been discussing.

As we use them a transition rule has the form $\mathbf{S}_i \wedge \mathbf{E}_j \rightarrow \mathbf{transition}(\mathbf{S}_k)$. The guard in this rule has two components, $\mathbf{S}_i$ and $\mathbf{E}_i$, the first denoting an invariant defining an equivalence classes of states and the second an equivalence class of events. The transition transition(S_k) "repairs" the program state so that it satisfies the invariant S_k instead of $\mathbf{S}_i$, which had characterized the state preceding the transition.[1]

In principle just one set of repair rules is needed for a state, and this set fires whenever the state is enter, thereby ensuring the state invariant is satisfied. In practice, however, a set of repair rules is specialized to a particular transition. This specialization is possible because the invariants for the source states and events serve as preconditions for the associated set of repair rules.

The basic analytical problem is that of replacing a sequence of statements with an iterative guarded command. This requires that each operation in the sequence be associated with a guard and that the program state be correspondingly enriched to ensure that the operations will still be performed in the correct sequence. In imperative code, explicit operation sequencing may be used either to implement a transition rule or to control chaining of repair rules. The software engineer has considerable latitude as to how a sequence of operations is to be viewed, and his decision depends on the expected model guiding the work. A similar problem arises in decomposing conditionals, but at least with them some of the conditions under which an operation is performed are already explicitly represented.

In developing the state machine based model, it is important to distinguish between those conditions related to state from those which describe an event triggering a transition. Since there can be considerable latitude in the definition of an event (e.g., recognizing input to the program or an event external to the system, a "parallel activity" enters or leaves a state, or an operation returned a result within a designated range), this decision must also be driven by the structure of the expected model, typically consisting of several communicating, hierarchical state machines.

This abstract has introduced a model for structuring application rules, and has described a related process for recovering these rules from application code. We have developed over the course of our experiments in reverse engineering, and appears to be especially useful when studying code which must coordinate the operation of several hardware devices. Accounts of our experiments are available in [1], [2] and [3]. In the near future, the model described here is to be applied to a real-time component having extensive fault diagnosis and fault handling responsibilities.

[1] A set of repair rules responsible for ensuring a state invariant can be structured in terms of a lower-level state transition model.

References:

[1] B. Johnson, S.Ornburn and S. Rugaber, "A Quick Tools Strategy for Program Analysis and Software Maintenance," *Proceedings of the Software Maintenance Conference*, to appear, 1992.

[2] S.Ornburn and S. Rugaber, "Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs," *Proceedings of the Software Maintenance Conference*, to appear, 1992.

[3] S. Rugaber, S. Ornburn, and R. LeBlanc, "Recognizing Design Decisions in Programs," *IEEE Software*, 7(1), January, 1990.

Program Understanding Through Ad Hoc, Interactive Query Facilities On A Reverse Engineering Repository

Gary Ostrolenk
Lloyd's Register, 29 Wellesley Rd, Croydon, CRO 2AJ, UK

Abstract

In any reverse engineering toolkit, facilities for the maintenance engineer to query its repository directly are essential for program comprehension. Along with structured source code browsers and static analysis tools, they enable the engineer to gain an initial understanding of the code and they support the engineer in abstracting to higher level descriptions of the application. This paper outlines some of the requirements such query facilities should satisfy.

1 Introduction

The European *Esprit* project REDO established the utility of building a reverse engineering toolkit on top of a central repository shared by all its tools [6]. The repository, the System Description DataBase (SDDB), contains a fine grain model of the semantics of the code of the application under maintenance [4]. This is populated initially by a collection of parsers for the programming language, job control language, DBMS DDL and DML etc. in which the application is written. Other tools subsequently browse, analyse, manipulate, restructure, annotate and abstract from this representation of the application.

The SDDB is built using a PCTE-compliant software engineering database, the IPSYS Tool Builder's Kit (TBK), which provides direct support for an Entity-Relationship-Attribute (ERA) data model with entity type inheritance, 1:1, 1:M and M:M relationships, Unix-like path specifications treated as functional compositions and some of the features of the SQL SELECT statement [1].

One serious limitation of software engineering databases identified during the REDO project is their lack of a high level interface through which users (in our case, maintenance engineers) can make interactive and *ad hoc* queries and updates. Existing interfaces are implemented either as library functions embedded within a compiled programming language or as interpreted procedural scripts.

[5] identifies three different approaches to reverse engineering: *user-directed search and transformation* techniques, *transformational* approaches, and *knowledge-based* approaches. The REDO project concentrated on transformational approaches. Provision of a Query Tool embodying a high level database interface significantly enhances the REDO toolkit, enabling it to support user-directed search and transformation techniques.

In addition to providing a valuable interactive, *ad hoc* data interrogation facility in its own right, the Query Tool gives added value to the existing REDO tools and exploits their synergy as a toolkit. The Query Tool enables users to specify selection conditions for the entities to be manipulated by other tools. It can be used for simple static analysis, and to retrieve information required as parameters for the invocation of other tools, such as the names of application procedures. The REDO structure and diagram editors provide a user-friendly interface for the Query Tool.

2 Detailed Requirements

The detailed requirements for the Query Tool can be grouped into successively higher-level categories of facilities for the user: hiding physical details of the TBK DataBase Interface (DBI), logical enhancements to the TBK DBI, ergonomic enhancements to the TBK user interface, and hiding logical details of the SDDB schema. For brevity, this paper discusses only logical enhancements to the TBK DBI, focussing on features of SQL not available in the TBK DBI.

Queries on a TBK database are evaluations of attributes applied to entities. Attributes are either entity-valued (i.e. links) or printable (i.e. scalar).

It is possible to express some complex queries by *composing* attributes. Consider a database recording information about a college's departments, tutors and students. To find out the location of the department to which the tutor of Jones belongs, a user might first evaluate the attribute, `tutored_by`, applied to Jones, and then in a separate query evaluate the attribute, `belongs_to`, applied to the tutor returned by the first query. Finally, the user would evaluate the attribute, `located_at`, applied to the department returned by the second query. The TBK DBI allows the three attributes to be composed. The user can evaluate the single complex attribute, `tutored_by/belongs_to/located_at`, applied to Jones.

This is equivalent to the following SQL query:

```
SELECT DEPARTMENT.LOCATED_AT
FROM DEPARTMENT, TUTOR, STUDENT
WHERE STUDENT.NAME = 'JONES'
AND STUDENT.TUTORED_BY = TUTOR.NAME AND
TUTOR.BELONGS_TO = DEPARTMENT.NAME
```

Composition of attributes is valuable not only because it can make queries more concise, but also because it enables the user to specify the sequence in

which the *joins* of the query are evaluated. The SQL expression of the query above does not indicate whether the database should: (1) retrieve all the tutors and the locations of their departments, and then select the one who tutors Jones, or (2) retrieve the student called Jones, then retrieve the tutor who tutors her, and finally retrieve the location of the department to which that tutor belongs. The sequence of evaluation of the *joins* of a query can significantly affect its performance. In some ways, it is an advantage of SQL that it abstracts from such procedural considerations. Query optimisation is the concern of the DBMS, not of the user. Often, however, the optimum expression of the query is intuitively clear to the user, reflecting the logic of the query itself. This is particularly the case in domains such as software engineering which lend themselves to representation in network data models (c.f. Section 19.3 of [6]).

However, there are a number of features of SQL which enable complex queries not expressible in the TBK DBI as single declarative attributes. Currently, to make such queries involves embedding more than one query within procedural code. Provision of such features in a Query Tool would make *ad hoc* querying of the SDDB significantly easier.

Application Of An Attribute To The Members Of A Sequence

One common form of database query is to retrieve the value of an attribute applied to all the entities of a certain type. For instance, and continuing the college example, a user might want to know the names of all the tutors in the college. In SQL, this query is simply expressed as `SELECT TUTOR.NAME FROM TUTOR`.

The TBK DBI does not support such a single query because it distinguishes single and multiple-valued attributes. The attribute, `!TUTOR`, retrieves a sequence of tutors. Each composition of attributes, say `att1/att2`, applies `att2` to the result of evaluating `att1`. The attribute, `name`, characterises individual tutors, not sequences of tutors. Therefore, `!TUTOR/name` is an invalid attribute composition.

To achieve the same result using the TBK DBI, the user must first evaluate the keyed bag attribute, `!TUTOR ^ name`, which returns a sequence of tutors keyed by name, and then iterate through the members of the sequence evaluating each of their names.

Joining Attributes Of More Than One Entity Type

Another common form of database query is to join attributes of more than one entity type. For instance, a user might want to retrieve Jones' home address and the address of her hall of residence. Suppose that each student's home address is stored as an attribute of the student, and that information about college halls of residence is stored centrally. In SQL, the query could be expressed as:

```
SELECT STUDENT.HOME_ADDRESS, HALL.ADDRESS
FROM STUDENT, HALL
WHERE STUDENT.NAME = 'JONES'
AND STUDENT.LODGES_IN = HALL.NAME
```

The TBK DBI does not support such single queries because functional composition is linear. Although the tuple construct can be used to retrieve two printable attributes of the same entity, the attributes of a tuple cannot themselves be composed. Hence, `[home_address;lodges_in/address]` is not a valid tuple attribute and cannot be applied in a single query to Jones.

To achieve the same result using the TBK DBI, the user must first evaluate the attribute, `home_address`, applied to Jones, and then in a separate query evaluate the attribute, `lodges_in/address`, again applied to Jones.

Boolean Logic

Support for boolean operators within attributes is minimal in the TBK DBI. Alternate values (or value ranges) can be specified for a printable key attribute. If a key is a tuple of several attributes, values or value ranges can be specified for each of them. Hence the TBK DBI provides *disjunction* for key values of a single attribute, and *conjunction* of key values for different attributes of tuples.

Continuing with the college database illustration, a user might want to know the name, funding authority and age of all mature students from Yorkshire or Kent. The following SQL query could be used:

```
SELECT NAME, AGE, FINANCED_BY FROM STUDENT
WHERE AGE > 26 AND (FINANCED_BY = 'Kent'
      OR FINANCED_BY = 'Yorkshire')
```

Using the TBK DBI, the user could evaluate the attribute,

```
![*;>26;Yorkshire,Kent].STUDENT
       ^ [NAME;AGE;FINANCED_BY]
```

and then iterate through the sequence of students returned evaluating their key values.

However, a user might also want to know the names and funding authorities of all students who do not come from Yorkshire or Kent. This query can be expressed using the following SQL command:

```
SELECT NAME, FINANCED_BY FROM STUDENT
WHERE NOT (FINANCED_BY = 'Kent'
           OR FINANCED_BY = 'Yorkshire')
```

It cannot be expressed as a single query using the TBK DBI because *negation* is not supported.

A user might also want to know the name, age and funding authority of all students who are mature or are self-financing. This can be expressed using the following SQL query:

```
SELECT NAME, AGE, FINANCED_BY FROM STUDENT
WHERE AGE > 26 OR FINANCED_BY = 'self'
```

It cannot be expressed as a single query using the TBK DBI because disjunction between values of different key attributes is not supported.

Embedding Queries Within Queries

SQL enables queries to be nested. Often this facility is a convenience, and equivalent "flat" queries can be formulated using joins. However, there are some queries which can only be expressed using nested constructs. The TBK DBI does not support any nested queries in this sense. Several examples of nested queries are presented below:

1. *Get the names of all students funded by Scottish local authorities.*

```
SELECT STUDENT.NAME FROM STUDENT
WHERE FINANCED_BY IN
 (SELECT AUTHORITY.NAME FROM AUTHORITY
  WHERE COUNTRY = 'Scotland')
```

2. *Get the names of all students who have research assistants as tutors.*

```
SELECT STUDENT.NAME FROM STUDENT
WHERE EXISTS
 (SELECT * FROM TUTOR
  WHERE STUDENT.TUTORED_BY = TUTOR.NAME
  AND TUTOR.STATUS = 'research_assistant')
```

3. *Get the names and addresses of all students who have gained an "A" grade in all subjects.*

```
SELECT STUDENT.NAME, STUDENT.ADDRESS
FROM STUDENT WHERE NOT EXISTS
 (SELECT * FROM SUBJECT WHERE NOT EXISTS
   (SELECT * FROM EXAM
    WHERE EXAM.SUBJECT = SUBJECT.NAME
    AND EXAM.EXAMINEE = STUDENT.NAME
    AND EXAM.GRADE = 'A' ) )
```

Queries 1 and 2 can be expressed as flat joins. For instance, Query 1 can be reformulated as:

```
SELECT STUDENT.NAME FROM STUDENT, AUTHORITY
WHERE STUDENT.FINANCED_BY = AUTHORITY.NAME
AND COUNTRY = 'Scotland'
```

Similarly, queries involving one existential quantification (i.e. one `SELECT ... WHERE [NOT] EXISTS ...`) can be reformulated as flat queries. However, Query 3 cannot be flattened. Its double nesting of negated existential quantification, equivalent to universal quantification, captures an important feature of queries.

Definition Of Views

To minimise data redundancy, and hence to minimise the need for application source code to maintain integrity in the database, SQL enables *views* to be defined onto the base relations of a schema. These views appear to the user as relations, and are tailored to the needs of particular users. However, they are not stored directly on the database but are derived at the time of query from the base relations. Each view is defined as a `SELECT ...` statement.

As an example, a view might be defined for Query 3 above, called PERFECT_STUDENT:

```
CREATE VIEW PERFECT_STUDENT ( NAME, ADDRESS )
 AS SELECT STUDENT.NAME, STUDENT.ADDRESS
  FROM STUDENT WHERE NOT EXISTS
   ( SELECT * FROM SUBJECT WHERE NOT EXISTS
      ( SELECT * FROM EXAM
        WHERE EXAM.SUBJECT = SUBJECT.NAME
        AND EXAM.EXAMINEE = STUDENT.NAME
        AND EXAM.GRADE = 'A' ) )
```

It is then possible for an inexperienced user to make use of this complex query simply by using the PERFECT_STUDENT view. For instance, to find out which mature students have "A" grades in all subjects, the following three line query can be used:

```
SELECT STUDENT.NAME FROM PERFECT_STUDENT
WHERE STUDENT.AGE > 26
```

The definition of views is not supported by the TBK DBI.

Entity Identity

The Relational Model [2] treats all data as relations (tables), i.e. as sets of tuples (records) of attribute values. Entity types are modelled as relations. Each individual entity is represented by a tuple of attribute values in the relation modelling its entity type. There are two important consequences of this representation: it is not possible to have two entities of the same type with identical attribute values, and each entity type must have a unique key identified for it in the schema.

The Extended Relational Model [3] and most OMSs support the concept of entity identity independently of the attributes which are defined for the entity type in the schema. This has enabled OMSs to provide a means of referring to particular entities through the use of *variables* whose domain is the class of entities. These variables are, in effect, pointers to entities.

The TBK DBI supports such entity identity through the use of *entity tokens*, which are variables within the environment within which the query is made. They can be the subject of queries (i.e. that to which an attribute is applied) but they cannot be used within attributes. Different entity tokens can be compared for identity using the function, `same_entity`. Entity tokens are relative to the process in which they are created, and cannot be used for entity identification over several processes.

The TBK DBI also provides a system-defined attribute, $ID, for all entities. In effect, this is a unique key for all entities whose value is assigned and maintained by the system, but which can be read by users. Hence it can be evaluated and compared within queries to establish identity between entities.

It is essential that any software engineering query language satisfying the requirements outlined in this paper retains the ability to refer to entities using entity tokens within the query, rather than performing explicit joins on the $ID attribute.

Acknowledgement

The author would like to thank Kevin Lano and Mary Tobin for their helpful comments on this paper, and the Committee at Lloyd's Register for permission to publish it. The views expressed in the paper are the opinions of the author and are not necessarily the views of Lloyd's Register.

References

[1] John Cartmell and Albert Alderson. The Eclipse Two-Tier Database. In Frank Bott, editor, *ECLIPSE: An Integrated Project Support Environment.* Peter Peregrinus Limited, 1989.

[2] E F Codd. A relational model of data for large shared data banks. *CACM 13 N0.6*, June 1970.

[3] E F Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397 – 434, December 1979.

[4] Gary Ostrolenk and Mary Tobin. The REDO SDDB: A repository for reverse engineering. Technical report, Lloyd's Register, Lloyd's Register House, 29, Wellesley Road, Croydon CRO 2AJ, England, 1992.

[5] Mary Tobin (editor). Reverse engineering: A review. Technical Report SREDM\TP01\01.010, Lloyd's Register, Lloyd's Register House, 29, Wellesley Road, Croydon CRO 2AJ, England, October 1992.

[6] Henk van Zuylen, editor. *The REDO Compendium of Reverse Engineering for Software Maintenance.* Wiley, 1992.

Program Comprehension As A Cooperative Process

Position Statement - 1992 Workshop on Program Comprehension

Alex Quilici
University of Hawaii at Manoa
2540 Dole St, Holmes 483
Honolulu, HI 96822
alex@wiliki.eng.hawaii.edu

We've been exploring approaches to automated program understanding of realistic C and Fortran programs. Our experience in this endeavor have led to us to the belief that automated program understanding is doomed unless it's designed as part of a complete environment 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:

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

Most program understanders focus on the first two categories. The underlying 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? We recently studied two large classes of programs to find out. The first class involved a set of small C textbook programs. 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's composed from these sorts of cliches. Unfortunately, over half of these programs consisted of either domain-dependent cliches or code that seemed too unique to classify as a cliche. Our guess was that real-world software systems would be much worse. Furthermore, in examining textbook and student explanations of program behavior, we found that it's 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, does some computation involving the paired elements of the array, and saves the smallest result) rather than in terms of domain-dependent cliches (find the distance between the closest points in a set of points).

The second class we studied involved a set of real-world Fortran simulation programs. These tend to solve domain-specific modeling problems, so we would expect that a significant fraction of their code would be domain-dependent. Not surprisingly, we would wound up finding that these programs were so domain-dependent

that making any progress on understanding these programs required the help of a domain expert. While close to half of these programs consisted of domain-independent cliches, those cliches contributed little to understanding what the program was actually doing. In fact, for these simulation programs the vast amount of understanding involved recognizing how a piece of code contributed to modeling the objects and actions in the domain.

So it seems that automated program understanders aren't going to be able to tackle realistic software systems any time soon. To do so, they would require vast numbers of domain-dependent cliches. That means they would have to scale up to orders of magnitude more cliches than current program understanders can deal with, and that they would need a mechanism for forming these large libraries of domain-dependent cliches. And on top of that they would have to deal with plenty of code that doesn't seem to be cliched at all.

Our approach has been to view program understanding as a cooperative process between system and programmer. This view 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 duty is to record as much of the current combined system and programmer understanding of the program as possible. This means it's necessary to provide a mechanism by which programmers can record their understanding of a program. The programmer, for example, should be able to record that a particular pair of variables represent the X and Y coordinates of a point, and 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 construct a library of domain-dependent cliches. Both tasks require an explicit representation of the domain model, as well as an explicit, programmer understandable and providable representation of cliches and the constraints used to recognize them.

Our approach has been to let the programmer build a detailed type hierarchy of the objects and actions in the domain, and to provide links between the code in a particular program and the domain objects and actions. For example, in the simplest case a variable may correspond directly to a particular attribute of a given domain object, and a section of code may correspond to the computation of another attribute of that object or a relationship between a pair of objects. The idea is that as various programmers record their understanding of the code, the system gradually builds up a domain model and an explicit mapping of that model to program code. In addition, when the programmer link a code fragment to a domain action, the programmer specifies the constraints the programmer used to recognize that relationship. This provide information that the system can use to automatically recognize that cliche in the future.

The second duty is to suggest to the programmer what a particular piece of code might be doing when the system can't confirm it. The system, for example, might point out that one code fragment appeared to be computing a distance, but that the values being subtracted and squared didn'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 the same lines, the system must be able to explain its cliches and reasoning to the programmer. Ideally, the programmer could be able to 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 that it's occurring.

Our approach in this area has been to explicitly store indexing information with each cliche and to explicitly represent certain cliches as specializations of other cliches. For example, the distance cliche is indexed by the explicit presence of a square root function or a sum of squares. In addition, each cliche also has a computationally inexpensive filter associated with it for determining whether it was falsely indexed. For example, with the distance cliche the filter is whether the values being summed are both the result of subtractions. The system only scans the code for indexes to cliches in its library, and tries to fully match only those cliches that pass through the filter. When a particular piece of code doesn't correspond to any cliche, the system presents the considered cliches and failed constraints to the programmer, so the programmer can have an idea of what the system thought were relevant possibilities. The programmer can then create new cliches that are a generalization or specialization of these existing cliches.

Our work on both these tasks is its early stages. However, we feel our approach of treating program understanding as a cooperative effort between the programmer and the system is a promising one, and that the problems of representing, accessing, and acquiring domain-dependent cliches are crucial to eventually creating useful program understanding systems.

Position Paper
Workshop on Program Comprehension

REVERSE ENGINEERING BY SIMULTANEOUS PROGRAM ANALYSIS AND DOMAIN SYNTHESIS

Spencer Rugaber

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

1. INTRODUCTION

There is several issues concerning appropriate strategies for analyzing a program. One such issue concerns whether the program analysis should proceed top-down or bottom-up. For example, Soloway and his colleagues propose a bottom-up approach involving the detection of patterns or *plans* in the program text[9]. Brooks, on the other hand, proposes a scheme whereby analysts are guided by their expections of a program's content and search for confirming *beacons* in a top-down fashion[2].

Another issue arises when considering the relative importance of a program's formal content, the executable statements and declarations in the program's text, and its informal aspects, such as comments and mnemonic variable names. Biggerstaff *et al.*[1] is among the few that concentrate on the informal information to construct a conceptual hierarchy describing the application domain with which a program in concerned.

Synchronized Refinement is a reverse engineering methodology that addresses these issues. It coordinates the bottom-up analysis of the program text with the top-down construction of a description of the specific application that the program accomplishes.

2. SYNCHRONIZED REFINEMENT

Chikofsky and Cross[3] define reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction". Typically, this includes both a description of *what* the system does, but also *how* it goes about doing it.

Synchronized Refinement consists of the parallel analysis of the source code and synthesis of a functional description. The process is driven by the detection of design decisions in the source code. Each decision is annotated in the functional description. The annotation states how the decision contributes to the function accomplished by the corresponding code. After each decision is detected and annotated, the corresponding part of the source code is replaced by an abbreviated description. In this way, the source code continually grows shorter while the functional description grows more complete.

Design decisions are structural decisions made by the original designer or programmer. Typical design decisions include the decomposition of a function into its subfunctions, the handling of special cases, and the use of one data structure to represent another that is not directly provided by the programming language. Other decisions include the encapsulation of a set of procedures into a module

and the introduction of a data item to save the result of a computation for later use. A more detailed description of design decisions is given in[8].

The synthesis process begins with a high-level description of the overall program as obtained from a review of the documentation, possibly augmented with comments from the source code. This description leads to certain expectations in the reverse engineer's mind. For example, if a sorted report is to be produced, it is expected that part of the code will be responsible for the sort (or preparing data for an external sort), and there is likely to be a section controlling the pagination of the report. Note that the expectations need not be complete nor even entirely accurate at this stage.

A dynamic list of expectations is constructed. As the analysis process proceeds, decisions are detected that relate to an expectation. For example, in the case of the pagination expectation, a section of code is found that keeps a counter that resets after reaching the length of a page. The annotation for the detected decision is placed together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program. During the process, certain expectations will be confirmed and others may be refuted. A confirmed expectation engenders others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the functional description expands. An alternative procedure, based on control flow analysis and program slicing, is described in a paper by Hausler *et al*[4].

3. EXPERIENCE

Synchronized Refinement has now been used successfully on several projects. Besides the exercize described in[8] that involved its use on a short numerical program written in FORTRAN, it has been used on two other substantial systems. Papers in this year's Software Maintenance Conference[6, 7] describe its use in understanding a large (> 100KLOC) real-time, embedded system written in a systems programming language. The reverse engineering resulted in both the detection of several bugs and in enabling a redesign that significantly improved the maintainability of the system. Synchronized Refinement has also been used to reverse engineer a moderate-sized (~ 10KLOC) COBOL information system. The reverse engineering enabled an object-oriented re-engineering into Ada[5].

4. ISSUES RAISED

Synchronized Refinement is a labor-intensive process. In principle, it is capable of guiding a reverse engineering activity to any degree of resolution. However, in doing this, the point may be reached where the cost of the reverse engineering approaches the cost of complete redevelopment. Thus the challenge becomes to guide the reverse engineering process to those aspects of the program that will yield the highest payback, for example, the capture of reusable components or the codification of business rules.

An issue also arises of how to represent the information derived from the reverse engineering process. The use of an integrated representation makes possible internal consistency checks, supports the production of documentation in a variety of formats, and may be useful in a later redesign effort.

References

1. Ted J. Biggerstaff, Josiah Hoskins, and Dallas Webster, "DESIRE: A System for Design Recovery," MCC STP-081-89, April 1989.
2. Ruven Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
3. Elliot J. Chikofsky and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13-17, January 1990.
4. Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, no. 1, pp. 55-63, January 1990.

5. Reginald L. Hobbs, John R. Mitchell, Glenn E. Racine, and Richard Wassmath, "Re-engineering Old Production Systems: A Case Study of Systems Re-development and Evaluation of Success," *Emerging Information Technologies for Competitive Advantage and Economic Development: Proceedings of the 1992 Information Resources Management Association International Conference*, pp. 29-37, Harrisburg, Pennsylvania, May 1992.

6. Bret Johnson, Steve Ornburn, and Spencer Rugaber, "A Quick Tools Strategy for Program Analysis and Software Maintenance," *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, November 1992.

7. Stephen B. Ornburn and Spencer Rugaber, "Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs," *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, November 1992.

8. Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr., "Recognizing Design Decisions in Programs," *IEEE Software*, vol. 7, no. 1, pp. 46-54, January 1990.

9. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595-609, September, 1984.

IMPROVING PROGRAM COMPREHENSION OF OBJECT-ORIENTED SOFTWARE SYSTEMS WITH OBJECT-ORIENTED DOCUMENTATION

Johannes Sametinger

Institut für Wirtschaftsinformatik
CD Laboratory for Software Engineering
Johannes Kepler University of Linz
A-4040 Linz, Austria

Object-oriented programming has brought many advantages to the software engineering community. Especially, the reuse of existing software components and application frameworks has improved the productivity in software development considerably. Now, the object-oriented programming paradigm has advanced in years and increasingly object-oriented software systems have to be maintained. Program comprehension plays a major role in software maintenance. Additionally, the increased reuse of software components, which is propagated and supported by object-oriented programming, necessitates the understanding of existing software during development and, thus, program comprehension becomes even more important.

Very often the only information a maintenance programmer can trust is the source code. It is the only accurate, complete and up-to-date representation of a program. However, source code listings are hardly suited to representing design decisions, the global system structure, or the interactions among different system components. System documentation is necessary to enable reuse and maintenance of software components. It should remain valid as long as the software is being used. Nevertheless, system documentation is often inadequate and out of date, and therefore unreliable and misleading.

Good (system) documentation should be complete, current, and consistent in style. We apply object-oriented technology to documentation in order to improve its quality by better reflecting the logical structure of a system. This way of organizing software documentation eases program comprehension of object-oriented systems.

Class Libraries and Application Frameworks

Typically, object-oriented software systems are extensions to class libraries or application frameworks. This characterization should become true for the documentation as well. Hence, such documentation should not describe an entire system from scratch; instead, it should contain a description of all extensions and modifications of the reused components and describe all system-specific parts as well. It is assumed that separate li-

brary documentation is available which—similar to the code—should build the base for the entire documentation.

With the object-oriented concepts of inheritance, information hiding, polymorphism, and dynamic binding software components have become reusable and extensible without the need to make any changes in the source code of these components. The reuse of whole collections of classes, called class libraries is a major step in increasing the productivity of software engineers. However, class libraries and application frameworks have strong impacts on the comprehension process. Comprehension of a software system being based on a class library depends on the documentation of the class library itself and the documentation of the application specific source code. In order to guarantee complete and consistent documentation of the whole software system, the documentation—similar to the code—has to be easily extended and modified without making changes to the original documentation.

Inheritance of Documentation

Inheritance can be viewed as both extension and specialization (see [Mey88]). A class X inherits from one or more superclasses A,B,C. The features of the superclasses are a subset of the features of class X, i.e., X heirs and thus provides whatever A, B, and C provide plus its own (extension). On the other hand, inheritance is used to realize an is-a relation. For example, a rectangle (X) is a special visual object (A) with the features of a visual object but specialized behavior (specialization). Inheritance is a means of better organizing the source code of a software system, because the logical structure of the software is getting closer to the structure of the part of the real world to be modeled.

In order to apply the inheritance mechanism to documentation, we divide the description of classes into subsections that can be modified and extended in subclasses. Thus, the documentation of a class is a combination of class specific descriptions plus the inherited subsections of the superclasses.

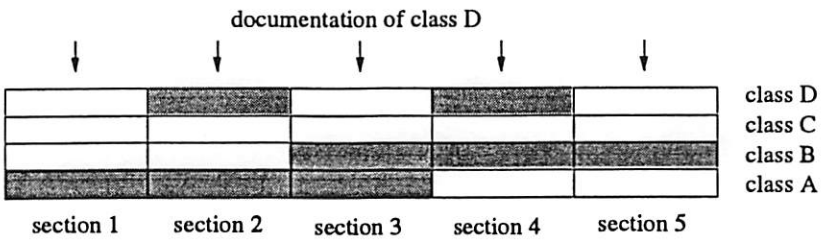

Fig. 1: Inheritance in the documentation of a class

Figure 1 contains the structure of the documentation of classes A, B, C, and D. The documentation of class A consists of 3 sections, and classes B, C and D have five documentation sections. Class D inherits section 1 from class A, sections 3 and 5 from class B, and has sections 2 and 4 of its own. Please note that the documentation of class C consists of five parts, though not an extra line of documentation has been written for this class.

The documentation of methods is organized the same way as that of classes. It is worth mentioning that there might be classes that do not implement a certain method. Naturally, they do not contain any documentation for this method. However, both the method and its documentation are available in these classes through inheritance.

Conclusion

In our research projects we use the public domain application framework ET++ (see [Wei89]). Detailed documentation for the most important classes and methods of this class library is available. Unfortunately, it is rather cumbersome to get relevant information because the data is usually spread over the descriptions of several classes (superclasses). Therefore, we divided the documentation into sections (e.g., short description, instance variables, methods, example) to be used with our programming environment DOgMA, that supports object-oriented documentation (see [Sam92]).

Although the documentation of ET++ had not been written with inheritance in mind, the benefits of applying this mechanism has been enormous. The possibility to get the part of the documentation that is relevant for using a special class or method, even when it is spread over many superclasses, made reusing a complex class library much easier, and was especially esteemed by our students.

References

[Mey88] Meyer Bertrand: Object-oriented Software Construction, Prentice Hall, 1988.

[Sam92] Sametinger J.: Object-oriented Documentation, submitted for publication, 1992.

[Wei89] Weinand A., Gamma E., Marty R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming Vol. 10, No. 2, 1989.

Reengineering for Porting Transaction Processing Applications

Larry Van Sickle and Michael Ballantyne
EDS Research, Austin Laboratory
1601 Rio Grande, Suite 500
Austin, Texas 78701
lvs@austin.eds.com amb@austin.eds.com

Abstract

The Reverse Engineering group at EDS Research has been developing representations and software tools to mechanically assist in understanding and reengineering transaction processing applications written in COBOL. A division of EDS approached us to help them convert a very large minicomputer application to run under CICS on an IBM mainframe. These two platforms provide very different environments. The user interacts with the minicomputer one field at a time, but interacts with CICS a full screen at a time. This and other major differences between the two environments demand that any successful mechanical conversion strategy employ sophisticated feature extraction and restructuring techniques. In the remainder of this paper we describe in more detail the nature of this large commercial problem and the tools and techniques being applied to solve it.

Background

The Reverse Engineering group at EDS Research has for the last several years been developing tools with the goal of extracting "high level" descriptions of existing applications from the source code. Examples of high level descriptions include data models, data integrity constraints, and standard data processing paradigms such as "update a file" or "sum a column".

Tools and Representations

A COBOL program is first translated to a set of Prolog clauses. The Prolog representation provides an abstract syntax tree which can be manipulated for program restructuring and also provides direct access to every COBOL statement using Prolog's indexing mechanisms. A Prolog program computes a control flow graph and some data flow information. Prolog rules for extracting high level descriptions, or plans [APU, Hartman], refer to the abstract syntax tree, the control paths, and the data flow information. Plans that are recognized are condensed to Prolog terms. The toolset also contains a symbolic evaluator used for slicing [Weiser] and generating weakest preconditions.

Project Description

EDS has a contract with a federal agency to consolidate many applications onto an IBM mainframe. Several large applications comprising over two million lines of COBOL must be moved to the mainframe under CICS. The converted applications must appear as unchanged as possible to the end user. As part of the acceptance criteria, EDS must provide test data which will force execution of 70%+ of the basic blocks in each program. For the sake of efficiency it is important to minimize the degree of "conversational" as opposed to "pseudo-conversational" programming. In the following sections we describe the main differences between the source and target environments and the difficulties these difficulties impose on the conversion.

User Interface

A minicomputer application program interacts with the user one field at a time. The program can display a prompt string at any location on the screen and can display multiple strings at multiple locations with a single operation. The program can then place the cursor at any location on the screen and accept values typed by the user. The program can respond to each value that the user types. The program could, for example, use a value that user typed in to look up a record and display the contents of the record before the user types any other values. Each value that the user types in can be checked for validity immediately, and the user can be forced to type a valid value before going on to any other fields.

On the mainframe using CICS, in contrast, the program interacts with the user a full screen at a time. The user types in all the values in all the fields, then transmits the entire screen contents to the program. The program then checks the values for validity. The program can report errors to the user in a number of ways, but a common method is to highlight all fields that are in error and display a message describing the first error .

Program Control Structure

The minicomputer program begins execution and continues execution until final program termination. Conceptually the program is always executing, although in reality it may be interrupted and swapped out while waiting for input. During an interruption for user input the variables retain their values. Figure 1 shows a fragment of minicomputer COBOL code that accepts user input.

Mainframe programs are written in what is called *pseudo-conversational* style. A pseudo-conversational program exits completely whenever it interacts with the user. Pseudo-conversational programming is the

recommended style for mainframe transaction processing applications because it allows the most efficient use of the mainframe and therefore the largest number of simultaneous users. When a pseudo-conversational program stops execution to accept user input, the values of variables are lost. When the user transmits a screen full of data, the pseudo-conversational program begins execution with initial values for all variables. The program can store a block of values before it stops execution and can recover that block of values when it starts up execution again. Figure 2 shows in pseudo-code the usual structure of a pseudo-conversational program.

Restructuring

The differences in user interface and control structure of the two types of programs preclude any simple syntactic or local translation of the COBOL code. Instead, automated tools must recognize the function of large pieces of the minicomputer program, translate the recognized pieces, and place the translated pieces in the proper place in the new program. In some cases a recognized piece of the minicomputer program must be split into separate pieces and this requires a control restructuring of the pieces.

```
 1 A-IN.
 2   DISPLAY A.
 3   ACCEPT A ON ESCAPE GO TO END-PGM.
 4   IF A < 1 OR A > 10
 5      DISPLAY ERROR-MSG-A-1
 6      GO TO A-IN.
 7 B-IN.
 8   DISPLAY B.
 9   ACCEPT B ON ESCAPE GO TO END-PGM.
10   IF B < 1 OR B > 10
11      DISPLAY ERROR-MSG-B-1
12      GO TO B-IN.
13   COMPUTE C = A * B + 3.
14 C-IN.
15   DISPLAY C.
16   ACCEPT C ON ESCAPE GO TO END-PGM.
```

Figure 1 Fragment of minicomputer COBOL program

Creating screens

One problem in porting the minicomputer program is determining the format of the screen for the mainframe program. The minicomputer program can display any string at any location on the screen, and can determine at run time which strings to display, as in Figure 3. In this example the string displayed in line 5 overlaps the two strings displayed in lines 8 and 10, so the mainframe program would require two full screens or else the original fields must be rearranged.

The tools must determine what full screens are needed in the mainframe version of the program. To do this, the tools must analyze the control paths that display prompts, determine which prompts and input fields physically overlap, and determine how many unique screen images can actually be generated by the program.

```
 1   IF FIRST-TIME
 2     PERFORM INITIALIZATION
 3   ELSE
 4     RECEIVE SCREEN FROM USER
 5     RESTORE SAVED VARIABLES
 6     SWITCH (SAVED-STATE)
 7      CASE 1:
 8        PERFORM CASE-1-PROCESSING
 9        MOVE 2 TO SAVED-STATE
10        GO TO SEND-SCREEN
11      CASE n:
12        PERFORM CASE-n-PROCESSING
13        MOVE m TO SAVED-STATE
14        GO TO SEND-SCREEN
15     END-SWITCH
16   END-IF.
15 SEND-SCREEN.
16   SAVE SELECTED VARIABLES
17   SEND SCREEN TO USER.
18   GOBACK.
```

Figure 2 Pseudo-code generic mainframe program

```
 1   DISPLAY "NAME: " AT LINE 3 COL 2.
 2   DISPLAY "SSN:  " AT LINE 4 COL 2.
 3   DISPLAY "ADDR: " AT LINE 5 COL 2.
 4   IF COUNTRY = "USA"
 5     DISPLAY "ST:        ZIP: "
 6       AT LINE 6 COL 2
 7   ELSE
 8     DISPLAY "PROVINCE:"
 9       AT LINE 6 COL 2
10     DISPLAY "POSTAL CODE: "
11       AT LINE 6 COL 18
12   END-IF.
```

Figure 3 Fragment of minicomputer COBOL program

Analyzing control paths through a typical program with several thousand statements is combinatorially explosive. We have developed techniques for extracting a control subgraph consisting only of those statements that affect the screen. We then analyze this reduced graph. The techniques are similar to slicing [Weiser].

Recognizing User Interaction

Identifying blocks of statements in the minicomputer program that interact with the user is an important recognition task. For efficiency, the mainframe program should have the minimum number of sends and receives of screens. We need to identify the largest block of user interactions in the minicomputer program that can be

combined into a single send and receive of a screen in the mainframe program. This requires recognizing blocks of code that validate values typed by the user, and recognizing when a data dependency or control dependency requires that a send and receive be executed.

Determining Variables to be Saved

The values of all variables are reinitialized when the mainframe program receives a screen of data from the user. The mainframe program can store a block of values before it stops execution and can recover that block of values when it starts up execution again. The porting tools must determine which variables to save. We accomplish this with a form of live variable analysis.

Conclusion

Many of the application programs to be converted are quite simple. Some, however, are very complicated. Several application programs have over 500 conflicting pairs of screens. Analyzing these conflicts without any mechanical aid would be a daunting challenge. It would be similarly difficult to determine solely by manual inspection which variables must be saved across program invocations.

The tools we have developed have proved to be adaptable and effective in attacking the conversion problem described here. The conversion problem has, in turn, driven the development of more refined tools to understand and analyze source code. A knowledge-based approach to recognition and analysis has proved vital to success in this project.

References

[APU] Van Sickle, Larry and Hartman, John E., 1992. Introduction to the First Workshop on Artificial Intelligence and Automated Program Understanding, *Notes of the Workshop on Artificial Intelligence and Automated Program Understanding*, Tenth National Conference on Artificial Intelligence, San Jose, CA.

[Hartman] Hartman, John E. 1990. Automatic Control Understanding for Natural Programs. Ph.D. dissertation, Dept. of Computer Sciences, University of Texas at Austin.

[Weiser] Weiser, Mark, July 1984. Program Slicing, IEEE Transactions on Software Engineering SE-10:352-357.

From Code Comprehension Model to Tool Capabilities

A. von Mayrhauser
A. M. Vans

Department of Computer Science
Colorado State University
Fort Collins, Co 80524

1 Introduction

A major part of the maintenance effort is understanding the existing code. If we can define and present the maintenance programmer with information that best helps to understand the code, we can significantly improve quality and efficiency of program understanding and thus maintenance. Our research goal is to develop a tool capability model based on an integrated code comprehension model. This will lead to tools that support maintenance tasks more effectively.

2 Integrated Code Comprehension Model

Our integrated code comprehension model consists of four major components, (1.) Program model, (2.) Top-down model, (3.) Situation model, and (4.) Knowledge base. The first three reflect comprehension processes. The fourth is necessary for successfully building the other three models. Program, situation, and top-down model building are the three processes that together construct an understanding of code. Each component represents both the internal representation of the program being understood (or short-term memory) as well as a strategy used to build this internal representation. The knowledge base either furnishes the process with information related to the comprehension task or stores any new and inferred knowledge.

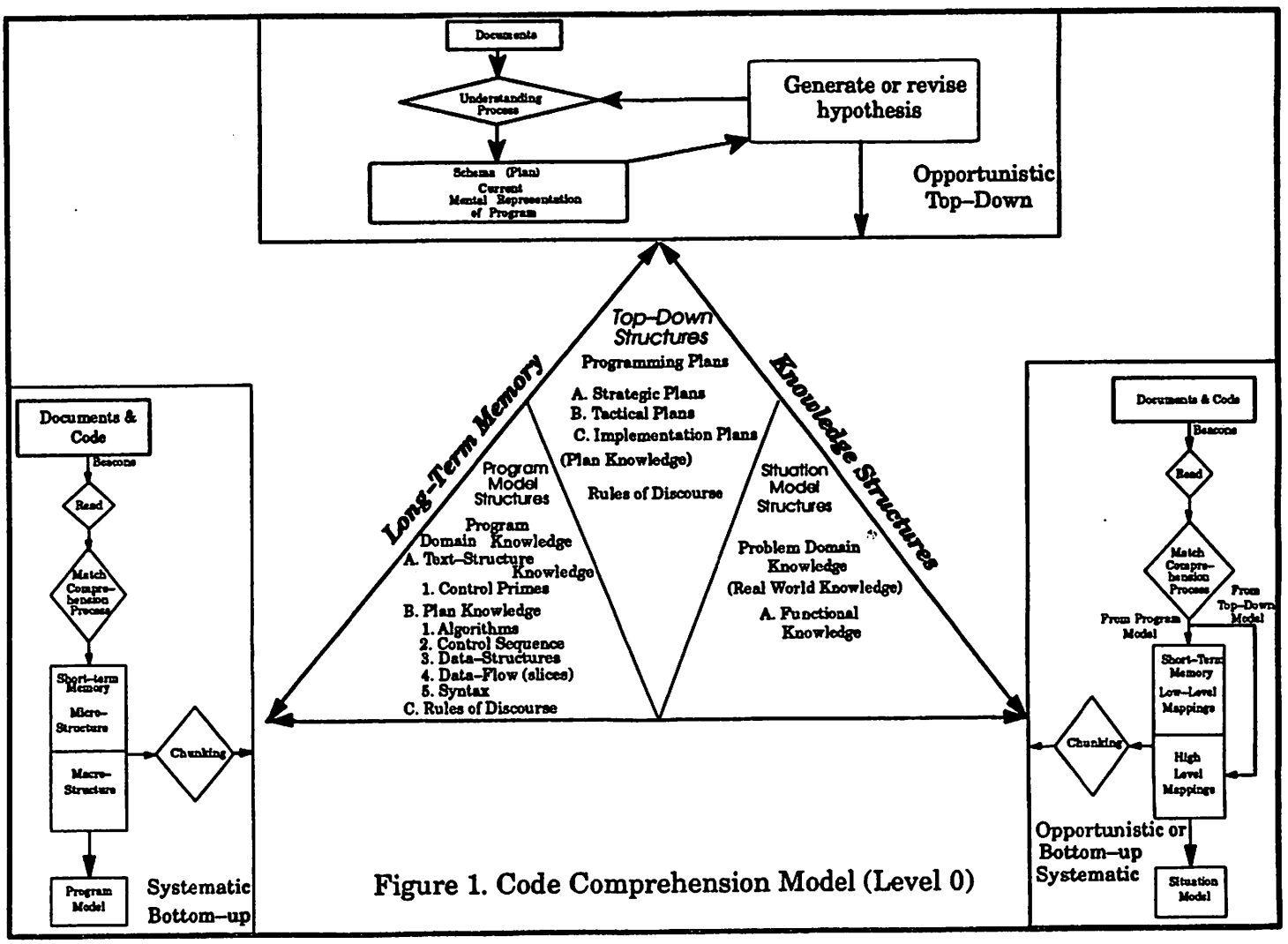

Figure 1. Code Comprehension Model (Level 0)

The *Top Down* model of program understanding is based on [2]. This process is typically invoked during the comprehension process if the code or type of code is familiar. When code is completely new to the programmer, Pennington found that the first mental representation programmers build is a *program model* consisting of a control flow abstraction of the program [1]. Once the program model representation is constructed, a *situation model* is developed. This representation, also built from the bottom up, uses the program model to create a data-flow/functional abstraction. The *knowledge base* , also known as long-term memory, is usually organized into schemas. Schemas are grouped into partitions specifically related to the comprehension processes. For example, knowledge of distinctive algorithms is used by the *program model building* process.

As Figure 1 illustrates, any of the three sub-models may become active at any time during the comprehension process. For example, during program model construction a programmer may recognize a beacon indicating a common task such as sorting. This leads to the hypothesis that the code sorts something, causing a jump to the top down model. The programmer then generates subgoals (e.g. I need to find out whether the sort is in ascending or descending order) and searches the code for clues to support these sub-goals. If, during the search, he finds a section of unrecognized code, he may jump back to program model building. Structures built by any of the three model components are accessible by any other; however, Figure 1 shows that each model component has its own preferred types of knowledge.

3 Tool Capabilities

While a comprehension model allows us to understand how programmers go about comprehending code, it does nothing to make this process more productive. Only tools that support and help to speed up the process would. Obviously, such tools must quickly and succinctly answer programmer questions, extract information the programmer asks for without extraneous clutter, and represent the information at the level at which the programmer currently thinks [3]. Above we have identified three such levels. Thus tool information should be available at the program model, situation model, and top-down model level. Tool information should accommodate the relevant knowledge structures as well. In addition, tools should aid in switching between components of the code comprehension model.

Unfortunately, our current tools fall far short in these respects. They either emphasize code analysis capabilities like control and data flow (components useful for the program model), or stay at higher levels of abstraction (some of our CASE tools). Even for each single modeling component, we frequently do not see all relevant aspects tool supported (such as defining hypotheses about the code, or identify a strategy on how to recognize code). Nor is the information represented succinctly as an answer to a specific question (e. g. a full dataflow table as compared to a showing dataflow of a specific variable through code highlighting and/or code elision). Connections between the three models of code comprehension are not commonly tool supported.

In ongoing research at CSU, we developed an integrated code comprehension model that unifies all three existing approaches. Now we are using it to derive tool capabilities that high quality maintenance tools should have. Several experiments with industry-strength code support the validity of our integrated model. With the information gained from the model and from the experiments we have started to develop a tool capability matrix. It relates comprehension tasks to individual tool capabilities. Figure 2 shows some of our preliminary results. We plan to use these insights in several ways:

- define and develop extensions to the AMT Maintenance Toolkit [4]. In particular we will use our findings to develop cognition focused user interfaces and new tools that support program understanding such as ways to represent hypotheses and cognitive strategies, and connections between program model, situation model, and the top-down model along with the information they use.
- define and develop an assessment procedure for tools that identify which aspects of code understanding they support and how well.

Master Goal	*Sub-goal Level 1*	*Sub-goal Level 2*	*Sub-goal Level 3*	*Tool*
Develop Program Model (Systematic Strategy)	Build Micro-Structure	Read Intro & Related Documents	Read pieces that are physically separated in the same document	On-line documentation with windows
			Look for references to module in other documents.	Cross reference modules to other Documents
		Determine Next Module to Examine	Systematic Search through code	Fusion Book Paradigm
		Examine Next Module in Sequence	Understand syntax of constructs	Syntax querying On-line programming language manuals
			Annotate listing with learned information	Annotation editing/reporting capabilities
			Highlight Begin..ends	Customizable reformatting

Figure 2: Paradigm Utilization Table(Partial)

References

[1] Nancy Pennington, **Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs**, In: *Cognitive Psychology* , 19(1987), pp. 295-341.

[2] Elliot Soloway, Beth Adelson, and Kate Ehrlich, **Knowledge and Processes in the Comprehension of Computer Programs**, In: *The Nature of Expertise* , Eds. M. Chi, R. Glaser, and M.Farr, ©1988, ALawrence Erlbaum Associates, Publishers, pp. 129-152.

[3] von Mayrhauser, **Should CASE Care about Software Maintenance or Why We Need Code Processing**, *Procs. CASE 90*, Dec. 1990, Irvine, CA, p. 20-22.

[4] von Mayrhauser, **AMT - The Ada Maintenance Tool Chest**, *Procs. Triada 91*, Oct 21-25, San Jose, CA, p. 294-303.

Research In Program Comprehension: Some Analogies and Directions

Norman Wilde, Department of Computer Science
University of West Florida, Pensacola, Fl 32514
904-474-2548, wilde@cs.uwf.edu

The exploding size and complexity of computer software undoubtedly create unique problems for its maintainers. Still, software professionals might gain insight by considering how other professions have managed to progress despite the great complexity of the systems they deal with. To develop a research agenda, we might consider some interesting analogies from the practice of medicine[1].

Medical doctors also need to diagnose or "understand" each patient, at least well enough to be able to propose effective treatments for the patient's ailments. The human body is a physical and chemical system made up of organs that interact in very complex and imperfectly understood ways. Individual differences are great and "domain knowledge" about the patient's ancestors, job, life style and mental state can be very important for diagnosis. The symptoms that the patient presents may often be confusing or ambiguous and may represent a combination of several ailments. Yet the medical profession has unquestionably made great progress in the last 50 years in diagnosis and treatment of many of humanity's ills.

A medical doctor can make use of literally thousands of diagnostic tests or studies: blood and urine tests, X-rays, electrocardiograms, CAT scans, and so on. Just a few of these will be selected for any particular patient, based on the symptoms presented and the results of earlier tests. Each test has its strengths, limitations, and ambiguities. Given the complexity of medical problems, there is no simple algorithm for arriving at many diagnoses[2].

By contrast, the tools available to the software maintainer are limited indeed. While it may be doubted that Software Engineering should follow slavishly in the tracks of the medical profession, the differences may provide some insights into productive research directions in program comprehension. Three notable differences are:

1. *The medical doctor knows what he is looking for.*

The medical profession has a fairly well established taxonomy of disease. The purpose of diagnosis is thus to go from "this patient has something wrong with his stomach" to "this patient has *amoebic dysentery*." The taxonomy of disease has several benefits, not least of which is the ability it gives doctors to communicate

[1] The medical analogy can certainly be carried too far. Many of the requests that maintainers receive would be the medical equivalent of "Please give this patient an extra arm, and while you are at it, increase his IQ by 20 points."

[2] The case studies in Michael Crichton's book *Five Patients* (Bantam Books) provide interesting illustrations of the diagnostic process.

with each other. By contrast, our vocabulary for describing software is very vague and incomplete; an experienced maintainer can have very great difficulty communicating his or her knowledge of a program to a colleague.

As a first research topic I would suggest that we need to identify and classify what it is about a program that maintainers really need to know. There are very few studies of how maintainers actually work and what kinds of information they use[3]. Additional case studies would be most useful, especially if some way can be found to compile and classify the results.

If we can identify the understanding needs for each kind of maintenance task, we will be in a stronger position in requesting documentation from developers and we will be better able to identify needed maintenance tools. As a possible starting point for discussion I would suggest that, depending on the task at hand, a maintainer may need to:

- Understand system architecture
- Understand system functionalities
- Understand system algorithms
- Understand system data
- Understand exception behavior

2. *Medical tests are standardized in purpose and interpretation.*

A medical doctor knows what each test should be able to do. The tests have been standardized with respect to the taxonomy of disease so that possible alternative diagnoses lead to specific tests being ordered which may confirm or negate each of the alternatives.

The maintainer has no such guidelines to follow. While many program understanding tools have been proposed, there is little guidance as to when to request a call graph, when to take a program slice, how to interpret metrics, and so on. We have not thought through the way each of these tools should be used.

Thus I suggest that a second research topic should be to evaluate the effectiveness of different tools in the context of specific maintenance tasks. Just as the doctor knows the margin for error in each diagnostic test, the limitations of each software tool should also be established.

[3] Notable exceptions are the papers by Soloway and his co-workers and by Koenemann et al [LETO.86, SOLO.88, KOEN.91]. The author has also been working on more informal studies focused on object-oriented programs [WILD.91].

3. *The range of medical diagnostic tools is very wide.*

Finally, it is clear that the medical profession has a much wider range of diagnostic aids than the software professional. Instead of the thousands of diagnostic procedures, in common use we may find cross-referencers, browsers, debuggers, call-graph or flowchart generators, debuggers, and occasionally data flow analyzers. Other tools such as slicers, regression test selectors and reverse engineering tools have appeared in the literature but are only rarely encountered in practice.

We must recognize that, just like medical diagnostic aids, each program understanding tool will only be usable in a limited range of circumstances. Normally the maintainer will, like the doctor, have to integrate information collected from a number of sources. Thus it is important to have a large number of complementary tools available.

The last research area I would propose is thus the development of new tools, that complement the information obtainable from the tools we now use. I would hypothesize that many of the new tools will be quite specialized. A maintainer will have one tool for analyzing recursive subroutines, another for displaying the behavior of a particular kind of data structure, a third for identifying a specific kind of program abstraction, and so on. The expert maintainer of the future will show his virtuosity by applying a wide range of tools to get each job done quickly and accurately.

References

[KOEN.91] Koenemann, Jurgen and Robertson, Scott P., "Expert Problem Solving Strategies for Program Comprehension," *Proceedings of the Conference on Human Factors in Computing Systems*, ACM Press, pp. 125-130, May 1991.

[LETO.86] Letovsky, S. and Soloway, E., "Delocalized Plans and Program Comprehension," *IEEE Software*, Vol. 3, No. 3, May 1986, pp. 41 - 49

[SOLO.88] Soloway, E.; Pinto, J.; Letovsky, S.; Littman, D.; Lampert, R., "Designing Documentation to Compensate for Delocalized Plans," *Communications of the ACM*, Vol. 31, No. 11, November 1988, pp. 1259 - 1267.

[WILD.91] Wilde, N.; Chapman, A.; Matthews, P.; Huitt, R., "Describing Object-Oriented Software: What Maintainers Need to Know," Software Engineering Research Center report SERC-TR-54-F, CIS Department, University of Florida, Gainesville, FL 32611, December 1991.

Position Paper:

Program Comprehension

Horst Zuse

Technische Universität Berlin (FR 5-3)
Franklinstraße 28/29
1 Berlin 10 (Germany)
Phone: +49-30-314-73439
Fax: +49-30-314-21103
e-mail: BITNET: ZUSE at DB0TUI11
Internet: ZUSE TUBVM.CS.TU-BERLIN.DE

1 Introduction

During the last years much attention has been directed toward the comprehension of software. Program comprehension is close related to software complexity. Software complexity is similar to program understanding and program comprehension.

We think, one important aspect, among others, is to formalize the term comprehension. However, the difficulty is that comprehension is an empirical term. Empirical statements cannot be formalized by mathematics.

The research area of software metrics tries to formalize the empirical term comprehension/complexity because the comprehension of programs should be represented by numbers. Measurement is a mapping of empirical objects to numerical objects by a homomorphism. However, the many existing software metrics show that there is no uniform view what programs make complicated to understand or to comprehend. In order to select appropriate software metrics for program comprehension it is necessary to have more clearness about the terms comprehension and complexity.

For this reason we think that measurement theory can help to get hypothesis about the term program comprehension. Roberts /ROBE79/ writes in the introduction of his book about measurement: *"A major difference between a "well-developed" science such as physics and some of the "less well-developed" sciences such as psychology or sociology is the degree of which things are measured".*
We agree to this statement of Roberts which says that measurement requires a more precise thinking of empirical statements.

Although program comprehension is an empirical term and cannot be defined mathematically, measurement theory can help to make hypothesis about reality. Measurement theory deals with the "connection" of the empirical world with the numerical world. It gives hypotheses about reality in form of axioms. Axioms are empirical conditions and these conditions are appropriated to discuss about the term program comprehension in a more precise way.

2 What is Measurement?

We want to introduce measurement as it is seen by Roberts /ROBE79/, Krantz et al. /KRAN71/ and Luce et al. /LUCE90/ very briefly. Measurement is a mapping of empirical objects to numerical objects by a homomorphism. Krantz et al. (/KRAN71/, p.33, line 13) write the following in the introduction of their book: *Here, by contrast, we are concerned almost exclusively with the qualitative conditions under which a particular representation holds.*
That means, measurement is based on a homomorphism between the empirical und numerical relational systems related to a measure. It gives qualitative (empirical) conditions for the use of measures. Representation means, that, for example, program comprehension is represented by a homomorphism into numbers.

In 1988 Kriz /KRIZ88/ gave a good explanation of the benefits of measurement in general. Kriz introduced the following picture.

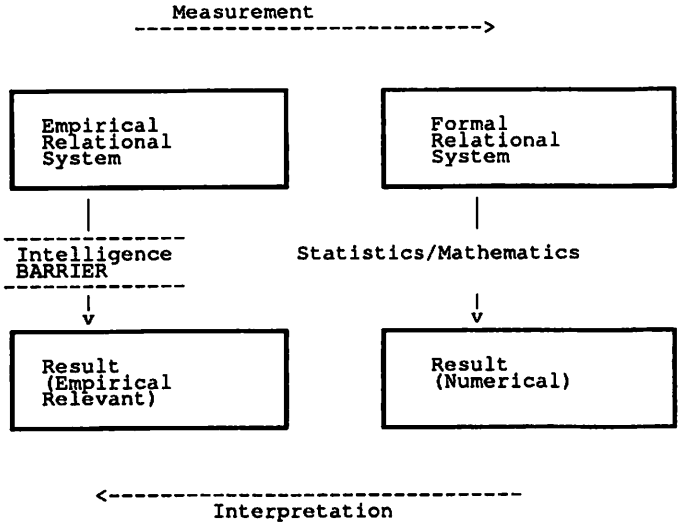

Figure 2.1: The measurement process as presented by Kriz /KRIZ88/. The empirical and formal relational systems are explained below.

Users want to have relevant empirical results of problems in reality. For example, users want to have relevant empirical statements about the complexity of programs. However, our human brain, in many of the cases, is not able to produce directly relevant empirical results. An exception is, for example, the length of wooden boards. In this case humans can make clear relevant empirical statements. However, considering the complexity of programs, the human brain is very often not able to make such statements. The relevant empirical statements related to software complexity can change over the time and people have different ideas of complexity. In many cases the human brain is unable to make relevant empirical decisions. Kriz calls this problem the "intelligence barrier". That means, in many cases, the human brain is not able to reduce informations without certain help.
In order to overcome the problem of the intelligence barrier measurement is introduced. Measurement is a mapping of empirical objects ("Empirical relational system") to numerical(mathematical) objects ("Formal relational system") by a homomorphism. Mathematics is used to process the informations. Doing this we get mathematical results ("Result numerical"). Now, the important step is to give the mathematical results an empirical meaning (empirical interpretation). The most important point of measurement is to give an interpretation of the numbers. In this case without an interpretation of the numbers it is not possible to make empirical statements. Measurement theory, as presented by Roberts /ROBE79/, Krantz et al. /KRAN71/ and Luce et al. /LUCE90/ gives the (relevant) empirical interpretation of the numbers by the empirical relational system.

In order to give an empirical relevant interpretation of the numerical results, we introduce measurement theory.

3 Empirical Conditions for the Term Comprehension

We now present a list of empirical conditions which are discussed in measurement theory /ROBE79/, /ZUSE91a/. Similar conditions can be also found by Weyuker /WEYU88/ and Fenton /FENT91/. Most of the empirical conditions are based on concatenation of objects. For this reason we introduce a concatenation operation for programs. Weyuker /WEYU85/, p.2, discusses the concatenation of programs, we cite it:
One way to think of a program is an object made up smaller programs. Certainly this is the perspective used in our definition of a program body, or any recursive definition. Using this point of view, the basic operation in constructing programs is composition. Since each of the programming language constructs a single entry and single exit, it makes sense to speak of concatenation of two program bodies. P;Q is the program body formed by appending the program body Q immediately following the last statement of P. We shall say that P;Q is composed from P and Q.

We call this type of concatenation of program components BSEQ.

We now give some empirical rules related to the concatenation operation BSEQ. Let **P** be the set of all flowgraph, P,P1,P2,P3,a,b,c,d∈**P**, $\bullet\geq$ an empirical relation like "equal or more complex", and o the concatenation operation BSEQ ($\approx$ means equally complex).

Condition C1:

$a \approx b \Rightarrow a \circ c \approx b \circ c$, and $a \approx b \Rightarrow c \circ a \approx c \circ b$, for all a, b, c ∈A.

Condition C2:

$a \approx b \Leftrightarrow a \circ c \approx b \circ c \Leftrightarrow c \circ a \approx c \circ b$, for all a, b, c ∈A.

Condition C3:

$a \bullet\geq b \Rightarrow a \circ c \bullet\geq b \circ c$, and $a \bullet\geq b \Rightarrow c \circ a \bullet\geq c \circ b$, for all a, b, c ∈A.

Condition C4:

$a \bullet\geq b \Leftrightarrow a \circ c \bullet\geq b \circ c \Leftrightarrow c \circ a \bullet\geq c \circ b$, for all a, b, c ∈A.

Conditions E (Extensive Structure):

A1': (P,$\bullet\geq$)	is a weak order
A2': P1 o (P2 o P3)≈(P1 o P2) o P3,	axiom of weak associativity
A3': P1 o P2 ≈P2 o P1,	axiom of weak commutativity
A4': P1 $\bullet\geq$ P2=> P1 o P3 $\bullet\geq$ P2 o P3	axiom of weak monotonicity
A5': If P1 $\bullet>$ P2 then for any P3, P4 there exists a natural number n, such that nP1 o P3 $\bullet>$ nP2 o P4,	Archimedian Axiom

The list above is appropriated to talk about the term complexity/comprehension in a more precise way. We will explain this very briefly with the Metric of McCabe.

4 Program Comprehension behind the Metric of McCabe:

We now apply the conditions above to the Measure of McCabe. For example, using the Metric of McCabe for analyzing program comprehension/complexity, the meaning of the Metric MCC-V=|E|-|N|+2 of McCabe is the following. P and P' are flowgraphs.

e1: If P results from P' by inserting an edge, then P is more complex than P'.

e2: If P results from P' by inserting an edge and a node, then P and P' are equally complex.

e3: If P results from P' by transfering an edge from one location to another location, then P and P' are equally complex.

The conditions e1, e2, and e3 describe the ordinal property of the Metric of McCabe related to the term complexity or comprehension. The are a prerequisite for the conditions C1-C4 and E. That means, using the Measure of McCabe for analyzing program comprehension, the term comprehension behind the Metric of Mccabe is described by the conditions e1, e2 and e3.

Because the Metric of McCabe is additive related to the concatenation operation BSEQ, the conditions C1-C4 and E are also assumed by this metric.

This concept shows that the term comprehension or complexity behind the Measure of McCabe can be described by empirical conditions from measurement theory. More informations about this approach can be found in /ZUSE91/ and /ZUSE91a/.

5 Future

We mean that the application of conditions of measurement theory can help to understand the term program comprehension in a better way. The term program comprehension can be described by empirical relational systems consisting of axioms.

6 References

/KRAN71/ Krantz, David H.; Luce, R. Duncan; Suppes; Patrick; Tversky, Amos
Foundations of Measurement - Additive and Polynominal Representation, Academic Press, Vol. 1, 1971

/KRIZ88/ Kriz, Jürgen
Facts and Artefacts in Social Science: An Ephistemological and Methodological Analysis of Empirical Social Science Research Techniques. McGraw Hill Research, 1988

/LUCE90/ Luce, R. Duncan; Krantz, David H.; Suppes; Patrick; Tversky, Amos:
Foundations of Measurement, Vol 3, Academic Press, 1990

/WEYU88/ Weyuker, Elaine J.
Evaluating Software Complexity Measures IEEE Transactions of Software Engineering Vol. 14, No. 9, Sept. 88.

/ROBE79/ Roberts, Fred S.:
Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences Encyclopedia of Mathematics and its Applications Addison Wesley Publishing Company, 1979

/ZUSE89/ Zuse, Horst; Bollmann, P.
Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics Sigplan Notices, Vol. 24, No. 8, pp.23-33, August 89.sp;.of 10;.cp 5;

/ZUSE91/ Zuse, Horst:
Software Complexity: Measures and Methods, DeGruyter Publisher 1991, Berlin, New York, 605 pages, 498 figures.

/ZUSE91a/ Zuse, Horst; Bollmann, Peter:
Measurement Theory and Software Measures. In: Proceedings of the International BCS-FACS Workshop (Formal Aspects of Computer Software), May 3, 1991, South Bank Polytechnic, London, UK", by T.Denvir, R.Herman and R.Whitty (Eds.), ISBN 3-540-19788-5, will appear in October 1992, Springer Publisher, Springer Verlag London Ltd, Springer House, 8 Alexandra Road, Wimbledon, London SW19 7JZ, UK.

ANALYZING AND CONTROLLING INSTALLED SOFTWARE

Nicholas Zvegintzov
Software Management News
141 Saint Marks Place, Suite 5F
Staten Island NY 10301 USA
(+1-718-816-5522 or fax +1-718-816-9038 or email 72050.570@COMPUSERVE.COM)

Cries for help

Cries for help. That is our name for telephone calls that come in to our office asking questions about how to manage real systems.

Sometimes the answers are easy (" '17' is always a good answer", says Richard Donnelly of the National Security Agency). But, more seriously, these cries for help often reveal how difficult managing a real system is, and how this evolving difficulty stays uncomfortably ahead of the techniques and tools that we deploy to deal with it.

Case in point...

Controlling the software of a bank

"I am calling from a medium-sized bank. We have a centralized IBM mainframe system, with a number of applications, many built around packaged software. Each of our functional groups understands its own system fairly well, but our subsystems are becoming more and more linked.

Now we are scared that we don't understand the system as a whole any more.

Are there CASE tools that can help us? (We have been thinking of diagramming the data flow of our system.)

Or other tools?"

A real time system

You are modeling a real time system.

You need to understand that a bank is a real time system – a network of communicating asynchronous processes – just as much as an airplane or a telecommunications network. Each depositor is a process, making changes at random times. The bank's own current position is a process, reflecting the accumulation of currently completed transactions. You need a model that is adequate for real time systems.

Real time modeling

There are real time modeling systems, originally aimed at military and engineering systems.

They describe:

- The processes that make up the system.
- The links between the processes.
- The behavior of the processes, e.g., Under what circumstances does process P communicate with Process Q?

Some systems will "play out" the behavior of the system. The following is from a description of Team*work*/SIM from Cadre Technologies Inc.:

> SIM executes a token-based symbolic simulation, i.e., a token on a data flow represents a data packet whose structure conforms to the data dictionary definition of the flow and a token on a store represents an instance of a record in the store. As a simulation proceeds, tokens arrive at processes, and, depending on the current state of the process and the number of tokens on other data flows in the process, tokens are consumed by the process. After a delay representing the execution of the process, tokens are produced on some of the output data flows of the process. The simulation executes interactively, or in batch, collecting statistics on contention, queue build-up, performance, etc.

Slow real time systems

Slow real time systems are harder than fast real time systems.

A bank is a slow real time system – a depositor may not have a transaction for weeks at a time, but a depositor may stay around for decades. A telecommunications system that goes down may lose all its current calls, but it will be back up with new calls within seconds. A bank that loses transactions is in trouble.

A slow real time system is also dangerous to update.

Why?

Because many of the elements of its transactions are locked up (effectively, hidden) in queues ("The ATMs at Branch XYZ lost contact with the mainframe due to a telephone failure, so they queued their transactions", etc.). We would be willing to bet that some of your nervousness is caused by bad system updates.

(**Cold comfort:** This is a major research frontier in the software maintenance field.)

Connectivity, interaction, and implementation

You need to model the connectivity, the interaction, and the implementation.

You need to understand how the system is connected ("The ATMs feed into the demand deposit accounting system", etc.), what the interactions are ("Demand deposit accounting takes messages off the ATM queues under the following circumstances...", etc.), and how the system is built ("Demand deposit accounting is built around a package from PQR, with the following shell modules...", etc.).

Statemate, from i-Logix Inc., has the right idea in offering three representations of the system:

- Activity-chart – representing control and data flow.
- Statechart – representing state transitions, and allowing for hierarchical and concurrent connections.
- Module-chart – representing the physical grouping of the control and data structures represented by the above charts.

Able to read real data

Your model must be able to read real data.

Many "CASE" tools have only a graphical representation, as if their only function was to let a human play with a screen and a mouse.

Warning: A CASE tool that only accepts graphical representations is no better than a video game. Why? Because...

Collecting real world data

You need to collect real world data.

The good news is that all the information described above is already "in" the system. The connectivity and the implementation is available to the system in the programs, in the data structures, and in the instructions for putting programs and data together (JCL, scheduling algorithms, network configurations, etc.).

The bad news is that it is tough to find, tough to read, and tough to piece together.

Connectivity

How many ways could procedures and/or data be hooked together?

- Modules compiled and linked to other modules.
  - –Look at source code.
  - –Look at compile- and link-time JCL.
- A module writes a file that another module reads.
  - –Could be explicitly mentioned in JCL.
  - –Could be visible in data dictionary.
  - –Could be visible in a scheduling algorithm.
  - –Could be dynamically accessed.
- A module communicates with another module via the operating system ("cross memory services", "pipes", short cuts within the telecommunications handler, etc.).
- Module M thinks it is talking to user U_M, module N thinks it is talking to user U_N, but in fact they are talking to each other.
- A module communicates with another module via a network.
  - –Could be visible in network directories.
  - –Could be dynamically linked.
- Other?

Never underestimate the tricks past programming gurus may have played!

Vendors

Look to Candle Corporation, Computer Associates International, Inc., International Business Machines Corporation, and LEGENT Corporation for help in finding information on the IBM mainframe. The mainframe usually functions as a giant network server – has visibility of distant peripherals and networks.

Novell, Network General, HP, etc., for network monitoring.

"Operations"

Warning: Many of the above vendors present themselves as doing business with "Operations", and Operations at your installation may keep a tight lid on its rice-bowl. You will need support at the management level above Operations to use these resources.

Parsing technology

Bad news: You may have access to this information, but can you understand it? It is in multiple formats and multiple languages.

Therefore you need parsing technology. You need to be able to read various language formats and bring together what you read.

Possible resources are

- InterPort Software Corporation
- SEEC, Inc.

Both companies came out of the conversion / language translation business.

Possible dark horse:

- LBMS Inc and LBMS Plc (swallowed a company called META Systems that pioneered this kind of creative reading).

A place to put the information

You need a place to put your information.

You need a large, robust store to keep the information you have discovered and to inquire on it.

The undisputed champion here is the Maestro II software management system from Softlab GmbH and Softlab, Inc., which runs as a network of workstations using an entity-relation database server.

Second choice: InterPort Software Corporation's PC-based code management database.

Configuration management

Control is the last step. In principle, you would like to control your updates from your model; in practice, if you are far from an adequate model, you are even further from control.

Suggestion: Use update queries as tests of the model's adequacy. As you build the model, query it for update advice ("If I change this module, what else do I need to change? What else do I need to freeze?"). After this advice begins to make sense is the time to begin to link the model into control.

Summary

- Aim for a real time model.
- Access real data.
- Learn to read real data.
- Store the data and inquire on it.
- Control is the last step.

An engineering problem

Conclusion: You have an engineering problem.

You may say "Isn't all that pretty elaborate to answer my question?"

But you may have (wild guess) 10 subsystems each with 1,000,000 lines of code and 5,000 modules. Boeing's new generation passenger transport, the 777, will have 132,500 distinct engineered parts, with about 3,000,000 actual parts ("including rivets, screws, and other fasteners"), and Boeing has eight IBM mainframes and more than 2,000 workstations to design and track them.

Boeing knows an engineering problem when it sees one. So should you.