Proceedings

IEEE Second Workshop on Program Comprehension

wpc'93

Proceedings

IEEE Second Workshop on Program Comprehension

July 8-9, 1993
Capri, Italy

Sponsored by
IEEE Computer Society
Technical Committee on Software Engineering

In cooperation with
Project SICP-CNR, Consiglio Nazionale delle Ricerche
Consorzio Campano di Ricerca per l'Informatica e l'Automazione Industriale, (CRIAI)

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

Published by the
IEEE Computer Society Press
10662 Los Vaqueros Circle
PO Box 3014
Los Alamitos, CA 90720-1264

IEEE Computer Society Press Order Number 4042-02
Library of Congress Number 93-77039
IEEE Catalog Number 93TH0551-2
ISBN 0-8186-4041-3 (microfiche)
ISBN 0-8186-4042-1 (case)

Additional copies can be ordered from

IEEE Computer Society Press Customer Service Center 10662 Los Vaqueros Circle PO Box 3014 Los Alamitos, CA 90720-1264	IEEE Service Center 445 Hoes Lane PO Box 1331 Piscataway, NJ 08855-1331	IEEE Computer Society 13, avenue de l'Aquilon B-1200 Brussels BELGIUM	IEEE Computer Society Ooshima Building 2-19-1 Minami-Aoyama Minato-ku, Tokyo 107 JAPAN

Production Editors: Regina Spencer Sipple and Edna Straub
Cover: Joseph Daigle/Schenk-Daigle Studios
Printed in the United States of America by Braun-Brumfield, Inc.

The Institute of Electrical and Electronics Engineers, Inc.

Message from the General Co-Chairs

Welcome to the Second Workshop on Program Comprehension!

This is the second workshop dealing with the dynamic new field of program comprehension. The first workshop was held in Orlando, Florida in conjunction with the IEEE Conference on Software Maintenance. Another one-day workshop on Artificial Intelligence and Program Understanding was held under the auspices of the American Association of Artificial Intelligence, in conjunction with AAAI '92. Based on the favorable response from these workshops, the organizers decided to hold the Second Workshop on Program Comprehension.

Although very young, the field of program comprehension offers tremendous potential. The evidence from the "trenches" suggests that program comprehension — or lack of it — is a major contributor to the cost of software. On the one hand, we have the huge legacy systems that have to be maintained, and program comprehension is a major component of the maintenance cost. On the other hand, the development of the new systems is more and more dependent on reuse of software components, which again have to be comprehended in order to be usable. The fundamental nature of program comprehension is illustrated by the success of two important software technologies: structured programming and object-oriented programming. Both were advocated on the grounds that they improve program comprehension. The Second Workshop on Program Comprehension presents a comprehensive overview of the current research in the field. We hope that the participants will find many stimulating ideas in the papers and the discussions. A tools fair will present state-of-the-art software tools facilitating program comprehension.

Our thanks go to all volunteers on both sides of Atlantic who made this workshop possible. They devoted many hours to the organization and preparation of the meetings. We also want to thank all cooperating organizations for their financial support. We want to thank the staff of the IEEE Computer Society for their support and professional help.

We wish all participants a pleasant stay on the beautiful island of Capri and an abundance of innovative and exciting new ideas for their future work.

Bruno Fadini
Co-Chair
Project Information Systems and Parallel Computation — CNR

Vaclav Rajlich
Co-Chair
Wayne State University

Message from the Program Co-Chairs

We take great pleasure in welcoming you to the Second Workshop on Program Comprehension (WPC '93). Program comprehension has come to be recognized as a fundamental activity in software engineering tasks such as maintenance, reuse, and re-engineering that involve the study and adaptation of existing programs. These tasks make up a very large proportion of the work load of most software engineers in today's world. Unfortunately, in spite of the enormous need for support of a systematic approach to program comprehension, we still find a lack of theories, models, methods, and techniques in this area. WPC '93 is intended to contribute to overcoming this lack. By providing an environment for fruitful discussion by those involved in research and experimentation — in both the academic and the industrial worlds — we hope to encourage the exchange of experiences, ideas, and results in this vital area. We believe that WPC '93 is a significant step in the evolution of this series of workshops whose success can be seen from the increasing number of researchers and projects related to program comprehension. The fact that WPC '93 is based on the presentation of full papers, (rather than position papers as in the past), as well as the introduction of a formal review process for the selection of papers, is clearly a sign of this important evolution.

The workshop is structured into six sessions: Models and Methodological Proposals for Program Comprehension (two sessions), Experiments in Program Comprehension, Program Representations for Program Comprehension, Integrating Documents and Code for Program Comprehension, and Tools for Program Comprehension. A panel discussion will explore the stimulating and innovative viewpoints on program comprehension by a number of highly qualified experts from industry. The keynote address by Keith Bennett, "Understanding the Process of Software Maintenance," will surely contribute to the success of this dynamic workshop.

We thank all of the authors who submitted their work to the workshop, the speakers, the panelists, and the session chairs for their tremendous contributions to the success of WPC '93. We also wish to thank all of the members of the Program Committee for their time commitments to carefully review the submissions. Special thanks are due to Regina Spencer Sipple and Edna Straub of the IEEE Computer Society Press and Janet Harward-Colopy of the IEEE for their assistance with the proceedings and program.

We wish you an interesting and enjoyable WPC '93!

Aniello Cimitile
Program Co-Chair
University of Naples, Italy

Norman Wilde
Program Co-Chair
University of West Florida, USA

WPC '93 Committees

General Co-Chairs

Bruno Fadini
Project SICP-CNR, Italy

Vaclav Rajlich
Wayne State University, USA

Program Co-Chairs

Aniello Cimitile
University of Naples, Italy

Norman Wilde
University of West Florida, USA

Tools Exhibition Chair

Guiseppe Visaggio
University of Bari, Italy

Program Committee

Paolo Benedusi (*Italy*)
Gianluigi Caldiera (*USA*)
Frank Calliss (*USA*)
Sylvie Cochinal (*France*)
Ugo De Carlini (*Italy*)
John Foster (*UK*)
Daniel Herlemont (*France*)
Wojtek Kozaczynski (*USA*)
Paul Layzell (*UK*)
Panos Livadas (*USA*)
Loredana Mancini (*Italy*)
Malcom Munro (*UK*)
Phil Newcomb (*USA*)
Alex Quilici (*USA*)
Harry Sneed (*Germany*)
Genny Tortora (*Italy*)
Axel van Lamsweerde (*Belgium*)
Larry Van Sickle (*USA*)
Giuseppe Visaggio (*Italy*)
Horst Zuse (*Germany*)

Table of Contents

* Paper not received in time for publication in proceedings.

Keynote Speaker: Keith H. Bennett

Understanding the Process of Software Maintenance

Understanding the Process of Software Maintenance

K.H.Bennett

School of Engineering and Computer Science, University of Durham, South Road, Durham DH1 3LE, UK

Abstract

An extensive programme of research has been in progress at Durham University for the past seven years to investigate how formal transformations may be used to assist the process of software maintenance, particularly addressing the issues of understanding and representing existing large software systems. In this paper, the results from the research are drawn together in order to assess the success and problems of the approach.

1 Formal Transformations

A formal transformation on a program, design or specification is an operation which alters the form of the operand, but not its semantics. As a very simple example, using the conventional semantics for assignment:

y:=1; x:=y+2

may be transformed by assignment merging and then addition to:

x:=3

The concept is not only applicable at the program code level. An algebraic design for the 'stack' data type may be transformed into an implementation, in terms of an array or linked list, using an appropriate sequence of transformations. This illustrates how formal transformations may be used to cross levels of design abstraction - in the above example, from design to code. It is necessary to identify a mapping from all design instances into the equivalent code instances.

This approach may also be used to transform requirements specifications, represented for example in Z or VDM, to designs.

The formal transformation movement has argued that executable software should be developed from specifications by the application of a sequence of suitable semantics preserving (or refining) transformations. A good survey is included in Partsch and Steinbrugen [2] and in Yang [3].

Since transformations are equivalences, the possibility exists at least in principle of using them in the 'opposite' direction to obtain designs and specifications from existing code. This is a problem that has received little attention by those working in program transformations, but has been the subject of a major research programme in the Centre for Software Maintenance at the University of Durham.

Reverse engineering is the analysis of existing software to obtain representations at the same or higher levels of abstraction. This definition embodies two ideas central to transformation theory: firstly the choice of suitable representations, and secondly, transformations to enable the software engineer to convert between representations without change of semantics.

Many existing programs only exist in the form of source code which typically has been heavily modified and extended following years of corrective, adaptive and perfective maintenance. Where documentation or high level design information is available, it is typically out of date and inconsistent with the source code. Software maintainers then have little confidence in software other than the source code.

Reverse engineering has the objective of taking such source code, and producing from it simpler, more understandable code. More ambitiously, there is the potential of extracting design and even requirements specifications. This can be done by inspection by skilled software engineers, but the problem seems well suited to computer based support, and transformation theory would seem to provide a sound basis.

At Durham, we have constructed a prototype tool called the Maintainer's Assistant to allow us to explore this problem and this paper draws together our experience of building and using this tool.

2 The Maintainer's Assistant

The Maintainer's Assistant uses an internal wide spectrum language (WSL) which can be used equally well to express non-executable specifications and to express low level program code. All transformations are expressed in terms of WSL. Such an approach is essential when moving across levels of abstraction. This means that there is an initial translation process to convert the source language code into the equivalent WSL. When a transformation project is complete, it may be useful to translate the WSL into a notation such as Z; again this is a once-only activity.

The maintainer proceeds by selecting a WSL construct, and then choosing a transformation from a library of pre-proven transformations. There are over 600 such transformations in the library, but the tool only presents those which are applicable to the selected WSL construct, thereby reducing the choice substantially (but at the cost of complex pattern matching involving unification).

0-8186-4042-1/93 $03.00 © 1993 IEEE

Considerable attention has been paid to providing a good user interface, and further details of this can be found in the literature [7,4]. WSL and its rationale is described in [5,6].

3 Results

Our description of the tool presents an image of a maintainer rapidly selecting a series of transformations to acquire the design and even the specification of an existing, heavily modified software system. The objective of this section is to assess the extent to which this has been achieved.

3.1 Initial Translation to WSL

Throughout most of the project, the translation of source code to WSL was seen as a simple implementation activity of little research interest. In practice, this naive view turned out to be wrong for two reasons:

- the correctness of the whole system depends critically on the correctness of the initial translator.
- difficulty was found in deciding to what extent all the information existing in the source code could be retained. What would be important for subsequent understanding by the maintainer?

The first issue was addressed by adopting a very simple table driven translation process so that the mapping was available for human inspection, and at least in principle, proof. The main test bed has been IBM 370 Assembler, and the semantics of a typical instruction, in terms of potential side effects, is often complex. As a result, the size of WSL emerging from the translation was typically three times that of the source code. However, this was remedied by writing a formal transformation which could remove the considerable redundancy from the WSL, bringing the WSL back to a similar size to the original source.

The second issue has proved much more serious. It can be argued that even non-functional properties of source codes (such as timing in a real time system) should be represented. Even if only functional issues are considered, programming techniques such as self modifying code, complex addressing modes and dynamic addressing mean that the level of WSL is so low that it is far harder to understand than the original source. This is not an attribute of Assembler only; C programmers are well know for abusing pointers, using first class procedure values and so on.

It was realised that this initial stage is not really a translation, but a modelling stage, in which the WSL models those aspects of the source which are important for the maintainer's subsequent understanding and ignores others. For example, most Assembler programmers do not use self modifying code, and use of this knowledge leads to a much better (i.e. high level) model of the source.

3.2 Abstraction levels

It was a design aim of the tool that when using it at low levels of abstraction (where the WSL is at code level) much of the transformation activity could be automated. At the other end of the spectrum, a considerable amount of input and interaction with the user would be needed when transforming at requirements specifications level.

In practice, this aim has been met. The general problem of obtaining a specification from code is undecidable (there are an infinite number of specifications). Design decisions are implicit, and no automatic system can recover them. For example, it is extremely difficult to envisage a general system which could take some code and recognise it as e.g. a "sort" procedure, or which could take a collection of operations on an array and recognise the abstract data type underlying them. At a higher level, recognition of application domain objects would be even harder.

Cognitive modelling has provided useful input to thinking about this issue. Such models show that during development, expert software engineers are needed to convert specifications into designs; applications domain experts are needed in the first place to capture the requirements specifications.

In our case study work, it is clear that the tool is most effective in the hands of such experts, and it is not very helpful to the inexperienced maintainer. This adds weight to the view that skilled maintainers need a high degree of expertise at all levels of software abstraction. At low levels of abstraction, the tool largely but not completely automates reverse engineering; understanding by the maintainer has to be minimal. At higher levels, the maintainer tries out hypotheses to reinforce the progressive revealing of the understanding of the system. Such hypotheses may turn out to be dead ends, and as a result the tool provides an undo and history facility to make exploration easy.

3.3 Reduction in complexity

Typically, heavily maintained code is highly unstructured. If this is input to the Maintainer's Assistant, then the WSL will be similarly awful.

Reduction in complexity is an important technique in helping humans to understand a system, and at low levels of abstraction there are four main areas in which this may be done: control; data; procedural and performance.

Techniques for automatic control restructuring have been available for some time, but these tend to lead to the introduction of new flag variables, and often the program is lengthened considerably. In the Maintainer's Assistant, restructuring is achieved by the maintainer selecting a series of transformations but he or she is able to use the understanding of what the program does to avoid both the above pitfalls of automatic restructurers. This has been demonstrated in all our case studies.

Data type transformation typically involves discovering the most appropriate data type for the problem, and this is a key step, undertaken after the control structure has been simplified. If data types are transformed, then changes to the operations on them are also needed.

The third technique for improving the comprehensibility of a program is to have the appropriate procedural structure. In the Maintainer's Assistant, a set of powerful transformations is available to fold and unfold procedures, undertake static analysis to convert a parameterless procedure to one with parameters, to recognise similar sections of code etc.

The maintainer typically proceeds by unfolding procedures, restructuring, folding and then parameterising. Automation of this would be unlikely to succeed, but in our tool, human input means that the process is remarkably effective. In a number of case studies, large unstructured (4000 line) assembler modules have successfully been reorganised into a small main program which calls short single lexical level procedures. This has even been achieved for a real time programs including a despatcher. This capability alone makes the tool useful for some users, but it demonstrates the differing capabilities of full automation against the effectiveness of using human understanding at the appropriate points. Another powerful form of procedural abstraction is to replace iteration by recursion for problems which are more naturally expressed in recursive form. Many problem-solving techniques use a "divide and conquer" approach and recursion introduction has been found to be widely applicable in most case studies.

Experiments have been undertaken to demonstrate this in a different way. The tool can compute a series of standard complexity metrics [1]. By applying all transformations to all syntactic constructs, it is possible to obtain the effect of a fully automatic system. Complexity metrics were used as a characteristic function. By allowing the system to run, and attempt to reduce complexity, it was found that the reduction could only proceed so far; at that point, if further reduction was to be achieved, human input was needed. Typically this involved making a temporary sacrifice in order to make larger gains later. The final form of abstraction involves removing the efficiency improvements which are typically scattered through an implementation. This amounts to discarding information about the implementation and gaining information about the underlying design. A number of common techniques are used e.g. loop invariant introduction and recursion introduction.

3.4 Data Abstraction

Data abstraction is a powerful reverse engineering technique, and is central to system understanding. Much work has been undertaken in recognising candidate abstract data types in languages such as COBOL and Pascal by several research groups.

Transformations provide a means formally of converting data from one representation to another while preserving semantics. Fundamentally. this is very similar to data reification techniques used in program proofs. A 'retrieve' or 'abstraction' function must be identified which maps values of one data type into a value of the other such that the semantics of an operation on both is preserved (the retrieve function must therefore equivalently map the results of the operations). Identification of suitable retrieve functions needs human understanding. The choice of a retrieve function represents increasing human comprehension of the problem.

More recently, experiments have been undertaken to try to extract data designs from data intensive programs (typically those written in COBOL). The design representation we chose was the Entity-Relation-Attribute diagram. A number of tools have been described which take COBOL data and produce data design diagrams. Unfortunately, these are little more than diagramming tools. When ERA diagrams are converted into code, only part of the information is converted to data; the remainder ends up as code. A good example is the use of foreign keys to represent relationships. Design information has thus become distributed across possibly large sections of program.

If the maintainer is aware of this, then transformations can be used involving both code and data to retrieve the ERA model. In our experiments this as resulted in an interesting form of abstraction; much of the coding detail has been discarded but the foreign key operation itself must be retained.

3.5 Algebraic transformations

The transformations available in the tool library only involve very limited facilities for algebraic manipulation. In practice, the user needs access to the full range of basic algebraic axioms and rules of inference, and a number of theorems from number theory have been found to be useful. When analysing loops, induction techniques are important.

This issue represents the most serious shortcoming of the tool, and will be the subject of future research.

3.6 Precision

Many reverse engineering tools are designed as aids to human understanding. It is often not crucial if they do not handle "hard" cases, difficult language constructs etc. It is the human that takes final responsibility, and although the tool may well be judged on its capabilities, it may not be essential even for it to be correct.

This weak approach is not applicable to the Maintainer's Assistant, and the constraints on its design are hence much more severe. The maintainer expects the semantics to be preserved and is very intolerant of any shortcomings. For example, it is not acceptable to support only a source code language subset. Although the tool can be used for simple restructuring through to specification recovery, confidence of the user drains away rapidly if it does any of these fallibly.

Although the tool is a prototype, the design has had to take this constraint into consideration [1], as problems would have invalidated any sensible experimentation. In practice the main problem has proved to be the front end as explained in 3.1.

3.7 Is Reverse Engineering Viable?

Large software systems are usually derived from some model of the real world application domain, even if that model was never explicitly recorded or represented. By the time such a system needs reverse engineering, or simply comprehending, it has typically been heavily modified and extended. It is hence an open question whether this degraded system represents any sensible real world model. If it does not, this will severely limit the extent to which it can be understood, let alone reverse engineered.

Our experiments give us some grounds for hope that heavily modified systems can be reverse engineered to sensible, compact requirements specifications. Although no real world application domain model for a system may actually exist, in practice the software reflects a set of changes made in the application domain. For example, a tax program may be heavily modified over the years to reflect the current tax law. Although the program may be unstructured, the tax laws it reflects are not necessarily so.

Perhaps a deeper problem lies in the form of design representations chosen during reverse engineering. For example, a number of researchers are undertaking experiments to reverse engineer COBOL programs to object oriented design notations. The age of the source systems is such that OOD would not have been used in the original design activity. We have undertaken only very limited experiments on this issue. There is a danger that OOD may be chosen as it is fashionable, and other better approaches such as abstract machines may be rejected. This is a topic for further research. It would be attractive to retrieve components in this form, as they could potentially form reuse candidates. However, the boundaries around such components is not immediately obvious.

4 Other Work

Understanding real-time programs involves an additional complexity of comprehending timing relationships and constraints. Much of this is implicit in the source code. Considerable progress has been made in this area and the results are reported in another paper at this Conference.

Transformations may also be used to derive an efficient executable program from a formal non-executable specification. This offers the advantage of a derivation history in which the sequence of transforms used is recorded. Perfective maintenance can then be undertaken not in term of the program code, but using specification and design representations.

Work is underway at Durham to explore the effectiveness of this approach, and early results are encouraging. One fruitful approach is to transform the specification to an executable but inefficient program in as few steps as possible. As a second stage, this may then be transformed to an efficient version. Separating the issues seems to make perfective maintenance easier.

5 Conclusions

The development of a prototype formal transformation method and tool, together with a substantial programme of case studies, has demonstrated the practical feasibility of using this approach to help software maintainers. Users must think about software in a new way, as an operand to transformation operations. The approach helps users both to discover and then represent their understanding; progressively more of the user's software engineering and application domain knowledge must be employed as levels of abstraction are crossed.

6 Acknowledgements

The author would like to express his thanks to other members of the formal methods team. Particular thanks are due to Martin Ward, who undertook the research which has led to the theoretical foundations for the project. My thanks are also due to Tim Bull, Hongji Yang, Nigel Scriven, Brendan Hodgson, Dave Hinley, Dave Walton, Eddy Younger, Joan Carranca, and Jenny Newton.

The author would like to acknowledge the financial contribution of IBM UK, the UK DTI and SERC. He also thanks the team members at IBM Hursley for the contribution, particularly Ron Cain.

References

[1] Bennett, K.H., Bull, T. and Yang, H, *"A Transformation System for Maintenance - Turning Theory into Practice"*. Proc. IEEE Conf. on Software Maintenance, Orlando, Florida, November 1992.

[2] Partsch, H. and Steinbrugen, R., *"Program Transformation Systems"*, Computing Surveys, vol. 15, no. 3, Sept., 1983.

[3] Yang, H., *"How does the Maintainer's Assistant Start?"*, Durham University, Durham, Technical Report, 1989

[4] Yang, H., *"The Supporting Environment for a Reverse Engineering System - The Maintainer's Assistant"*, presented at the IEEE Conference on Software Maintenance - 1991, Sorrento, Italy, 1991

[5] Ward, M., *"Transforming a Program into a Specification"*, Durham University, Durham, Technical Report, 1988

[6] Ward, M., *"A Catalogue of Program Transformations"*, Durham University, Durham, Technical Report, 1988

[7] Bull, T., *"An Introduction to the WSL Program Transformer"*, presented at the IEEE Conference on Software Maintenance - 1990, San Diego, California, 1990

Session A: Models and Proposals for Program Comprehension

Chair: Alex Quilici

Criteria for Program Comprehension Derived from Software Complexity Metrics

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
Internet: zuse at tubvm.cs.tu-berlin.de

Abstract

Program comprehension is close related to program complexity. In order to analyze program complexity much effort has been spent to measure the complexity of programs. For this reason hundreds of software complexity measures were proposed. In this paper criteria/conditions for program comprehension from software complexity measures are derived. It is also shown that using measurement theoretic numerical conditions from software complexity measures can be translated back to empirical conditions. That means the term comprehension can be described by empirical axioms. This approach makes it easier to talk about the term program comprehension.

1 Introduction

During the last years much attention has been directed toward the comprehension of software. Program comprehension is close related to software complexity. Behind software complexity program understanding or program comprehension is hidden. The term "software complexity" goes back to the sixties and seventies. Myers (Myers, 1976) and Yourdon et al. (Yourdon, Constantine, 1979) discuss the term "complexity". Myers writes about software complexity (p.37): *Complexity, being a principal underlying cause of translation errors, is one of the major causes of unreliable software. Complexity is both difficult to define precisely and to quantify. However, we can say that the complexity of an object is some measure of the mental effort required to understand that object.*
Myers characterizes complexity as (p.37): *In general, the complexity of an object is a function of the relationships among the components of the object. For instance, to some extent the complexity of an external design of a software product is a function of the relationships among all of the external interfaces of the product, for example, relationships among user commands and the relationships established between outputs of the system and inputs to the system. The complexity of a system architecture is a function of the relationships among subsystems. The complexity of a program design is a function of the relationship among the modules. The complexity of a single module is a function of the connections among program instructions within a module.*

In Yourdon et al. (Yourdon, Constantine, 1979) we also can find statements about complexity of software or of software designs. Yourdon et al. are saying that most of our problems in programming occur because human beings make mistakes and that human beings make mistakes because their limited capacity for complexity. This leads to the obvious question: What is that humans consider complex? In specific terms that means, what aspects of system design and program design do programmers consider complex? And, by extension, what can we do to make systems less complex.

These statements from the seventies and early eighties show that the discussion of complexity of software took place verbally. However, in the middle of the seventieth researchers began to propose software complexity metrics. The most famous software complexity metric is the Metric of McCabe (McCabe, 1976). Behind software complexity metrics empirically ideas of program complexity/comprehension are hidden.

In this paper criteria/conditions for program comprehension, which are derived from software complexity measures, are presented. Measurement theory is used to translate numerical criteria back

0-8186-4042-1/93 $03.00 © 1993 IEEE

to empirical criteria. Wakker (Wakker, 1989) also describes the advantage of measurement theory with the following statement: *So, representation theorems 'translate' theoretical statements into empirical statements. They give criteria for verification/justification or falsification/criticism.*

In detail we discuss the following. In Section 2 we discuss measurement and measurement theory very briefly. In Section 3 we present an overview of empirical conditions which are derived from numerical conditions. The independence conditions, the axioms of the extensive structure, the Weyuker conditions for program complexity, the axioms of Bache and the idea of program complexity/comprehension behind the Metric of McCabe are discussed. In Section 4 conclusions are given and Section 5 contains the used list of literature.

2 Measurement and Program Comprehension

In this Section we want to introduce the central ideas of measurement and a brief introduction in measurement theory which deals with the connection of empirical and numerical conditions by a homomorphism. We present measurement as it is seen by Roberts (Roberts, 1979), Krantz et al. (Krantz, Luce, Suppe, Patric, Tversky, 1971) and Luce et al. (Luce, Krantz, 1990) very briefly. In (Zuse, 1991) and (Zuse, 1992) the application of measurement theory to software metrics is described in detail.

2.1 What Is Measurement?

Measurement is a mapping of empirical objects to numerical objects by a homomorphism. Krantz et al. ((Krantz, Luce, Suppe, Patric, Tversky, 1971), 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. As empirical conditions are considered conditions which can be falsified or verified. Representation means, that, for example, program comprehension is represented by a homomorphism into numbers.

In 1988 Kriz (Kriz, 1988) gave a good explanation of the benefits of measurement in general. Kriz introduced the following picture.

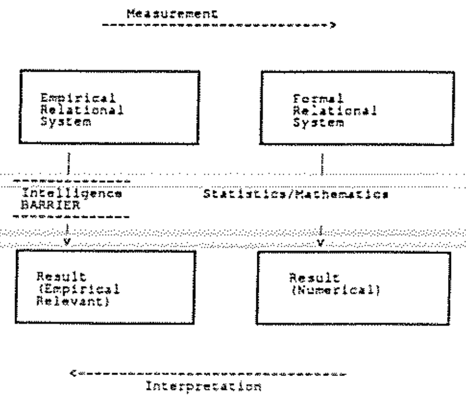


Figure 2.1: The measurement process as presented by Kriz (Kriz, 1988). The empirical and formal relational systems are explained below.

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 (Roberts, 1979), Krantz et al. (Krantz, Luce, Suppe, Patric, Tversky, 1971) and Luce et al. (Luce, Krantz, 1990) 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 and translate numerical conditions back to empirical conditions.

2.2 Basic Concepts of Measurement Theory

First of all we want to introduce the notion of an empirical, a numerical relational system and a scale. Let

$\mathbf{A} = (A, R_1, \ldots, R_n, o_1, \ldots, o_m)$.

be an empirical relational system, where A is a non-empty set of empirical objects, R_i are k_i-ary empirical relations on A with i=1,..,n, and o_j, j=1,..,m, are binary operations on A.

According to Luce et al. (Luce, Krantz, 1990), p.270, we assume for an empirical relational system **A** *that there is a well-established empirical interpretation for the elements of* **A** *and for each relation Si of* **A**. We also assume the same for the binary operations.

Let further

$\mathbf{B} = (B, S_1, \ldots, S_n, \bullet_1, \ldots, \bullet_m)$,

be a formal relational system, where B is a non-empty set of formal objects, for example numbers or vectors, S_i, i=1,..,n, are k_i-ary relations on B, and $\bullet_j$, j=1,..,m, are closed binary operations on B. We also include the case that there are no relations or no operations.

A **measure** μ is a mapping μ:A -> B such that the following holds for all i=1,..,n; j=1,..,m and for all $a, b, a_{1i}, \ldots, a_{ki} \in A$:

$$R_i(a_1, \ldots, a_{ki}) <=> S_i(\mu(a_1), \ldots, \mu(a_{ki}))$$
and
$$\mu(a\ o_j\ b) = \mu(a) \bullet_j \mu(b)$$

Then the Triple (A, B, μ) is called a **scale**. According to this definition we see that measurement assumes a homomorphism. More precisely, we consider the following problem in measurement theory following Krantz et al.: Let be (A, $\bullet\geq$,o) a relational system, where A is a non-empty set of objects, $\bullet\geq$ a binary relation on A and o a binary operation on A. The problem is which conditions (axioms) have to hold on (A, $\bullet\geq$,o) in order to have a mapping μ: A->$\Re$, where $\Re$ are the real numbers, such for that for all $a,b \in A$

$$a \bullet\geq b <=> \mu(a) \geq \mu(b)$$
and
$$\mu(a\ o\ b) = \mu(a) + \mu(b)$$

hold.

The answer is given in Krantz et al. ((Krantz, Luce, Suppe, Patric, Tversky, 1971),Chapter 3.2) and Roberts ((Roberts, 1979), Chapter 3.2).

Given two relational system **A** and **B** we can ask whether there exists a measure μ such that (**A**, **B**, μ) is a scale. This problem is called the representation problem.

2.3 Ordinal Scale

We now introduce the conditions for the use of software measures as an ordinal scale. In order to describe a measure as an ordinal scale we introduce the **weak order** which is a binary relation that is transitive and complete:

$P \bullet\geq P'$, $P' \bullet\geq P'' => P \bullet\geq P''$ transitivity,
$P \bullet\geq P'$ or $P' \bullet\geq P$ completeness (connectedness),

for all P, P', P" $\in$ **P**, where **P** is the set of flowgraphs and $\bullet\geq$ is a binary empirical relation, like "equal or more complex".

In (Roberts, 1979), p.110, we find the following theorem which we can apply directly to flowgraphs.

Theorem 2.1:
Suppose (**P**, $\bullet\geq$) is an empirical relational system, where **P** is a non-empty countable set of flowgraphs and where $\bullet\geq$ is a binary relation on **P**. Then there exists a function μ: P->$\Re$, with

$$P \bullet\geq P' <=> \mu(P) \geq \mu(P')$$

for all P, P' $\in$ **P**, iff $\bullet\geq$ is a weak order. If such a μ exists, then

$$((\mathbf{P}, \bullet\geq), (\Re, \geq), \mu)$$

is an ordinal scale.
◆

2.4 Measurement Theory and Program Comprehension

As mentioned above measurement theory deals with the connection of empirical conditions with numerical conditions via a homomorphism. This is expressed more formally with the statement

$$a \bullet\geq b <=> \mu(a) \geq \mu(b),$$

where the left side describes an empirical ranking order and the right side the numerical ranking order. The empirical relation $\bullet\geq$ can be interpreted as "equal or moe complex" or "equal or better to comprehend". More empirical conditions can be found if we consider the equation

$$\mu(a\ o\ b) = \mu(a) + \mu(b)$$

where o is a concatenation operation. In order to investigate additive measures the extensive structure is used. We always assume an ordinal scale which includes the conditions transitivity and completeness.

3 Empirical Conditions for the Term Program Comprehension

As shown in Section 2 it is possible to give numerical results an empirical meaning under the condition that a homomorphism exists. The double implication <=> allows to translate numerical conditions back to empirical conditions.

We now present a list of empirical conditions which are discussed in measurement theory (Roberts, 1979), (Zuse, Bollmann, 1992), (Zuse, 1992). Similar conditions can be also found by Weyuker (Weyuker, 1988) and Fenton (Fenton, 1991). These empirical conditions describe program complexity or program comprehension. As an empirical relation we introduce $\bullet\geq$ which means "equal or more complex" or "equal or better to comprehend". That means, for example, having two programs P1, P2$\in$P, where P is the set of all programs,

$$P1 \bullet\geq P2$$

is interpreted that P1 is equal or more complex than P2 or P1 is equal or more complicated to comprehend than P2.

3.1 Empirical Conditions Based on Concatenation of Objects

Many of the empirical conditions are based on concatenation of programs. In order to show these conditions we have to define a concatenation operation. Weyuker, for example, (Weyuker, 1988) p.1359, gives a definition of a concatenation operation: *A program can be uniquely decomposed into a set of disjoint blocks of ordered statements having the property whenever the first statement in the block is executed, the other statements are executed in the given order. Furthermore, the first statement of the block is the only statement which can be executed directly after execution of a statement in another block. Intuitively, a block is a chunk of code which can be always executed as a unit.*
In Bollmann et al. (Bollmann, Zuse, 1985), (Zuse, Bollmann, 1989) and Zuse (Zuse, 1991) (Zuse, 1992) the sequential concatenation operation for flowgraphs was introduced and denoted with BSEQ.

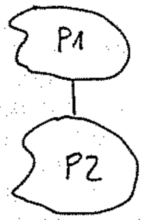


Figure 3.1: Sequential concatenation operation BSEQ=P1 o P2.

Two arbitrary programs P1 and P2 or flowgraphs are sequentially concatenated with BSEQ. The arbitrary programs or flowgraphs are so-called primes because they have only one entry and one exit. Weyuker calls this chunks or blocks. For more informations see Fenton et al. (Fenton, Hill, 1993) (Chapter 13).

We now consider empirical conditions for program complexity/comprehension based on concatenation operations.

3.2 Independence Condition and Program Comprehension

We now give some empirical conditions rules related to the concatenation operation BSEQ. Let P be the set of all flowgraph, a,b,c,d$\in$P, $\bullet\geq$ an empirical relation like "equal or more complex" ($\approx$ means equally complex) and o the concatenation operation BSEQ. The conditions C1-C4 are denoted as independence conditions and are described in detail in (Zuse, 1992).

Condition C1:

a $\approx$ b => a o c $\approx$ b o c, and a $\approx$ b => c o a $\approx$ c o b, for all a, b, c $\in$P.

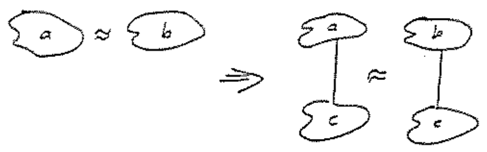


Figure 3.2: Empirical condition C1.

Condition C1 is the weakest empirical condition for independence between two components of a program. Independence means that the complexity of an entire software system can be derived from the complexity of the components.

We formulate this more formally related to software complexity measures by the following statement (Zuse, Bollmann, 1992). Exists an F such that

$$\mu(P \text{ o } P') = F(\mu(P), \mu(P'))$$

holds? This formal statement is not yet discussed by authors in literature. However, it is discussed by verbally formulated statements, for example, by Fenton (Fenton, 1991), who writes: *The complexity of a sequential flowgraph should be uniquely determined by the complexities of the components.* Generally, we can say that the independence condition C1 for program complexity is not widely accepted.

P'∈**P**. Weyuker requires this axiom here in a weaker form (•≥ instead of •>). The idea behind this axiom is that Weyuker means adding something to a program makes it more complex or more difficult to comprehend.

3.5.2 Rejection of Condition C1

Weyuker rejects the independence condition C1 and requires

$$P \approx Q \Rightarrow \neg(R \circ P \approx R \circ Q).$$

Doing this Weyuker also rejects the conditions C2-C4 and the extensive structure as conditions for program complexity/comprehension.

3.5.3 Rejection of Weak Commutativity

Weyuker rejects the axiom of weak commutativity which is also an axiom of the extensive structure. The axiom of weak commutativity is defined as

$$P \circ Q \approx Q \circ P,$$

for all P, Q∈**P**.

Weyuker requires

$$P \circ Q \neg\approx Q \circ P,$$

Rejecting the axiom of weak commutativity means also rejecting the extensive structure.

3.5.4 Requiring the Extensive Structure

Weyuker requires, among others, the extensive structure because she discusses the statement

$$\mu\,(P1 \circ P2) = \mu\,(P1) + \mu\,(P2),$$

where μ is a software complexity measure. With this statement Weyuker requires the extensive structure. And, Weyuker requires independence because the conditions C1-C4 are included in the extensive structure (Zuse, 1992).

There is a contradiction since Weyuker rejects weak commutativity and requires the extensive structure. Both is not possible. This shows that conditions which are derived from software complexity measures can be interpreted from a measurement theoretic view.

3.5 Required Properties by Bache

Bache (Fenton, 1991), p.218, suggests axioms of program complexity. Bache formulates his axioms numerical, but using measurement theory it is possible to translate the numerical conditions back to empirical conditions by the implication =>. Again, o is the sequential concatenation operation BSEQ. Bache denotes F, H, G as arbitrary flowgraphs and P1 as an trivial flowgraph consisting of one edge and two nodes. For his considerations Bache assume that holds

$$\mu\,(H) > \mu\,(G) \Rightarrow H \bullet\!> G.$$

That means flowgraph H is always more complicated or less to comprehend than flowgraph G. μ is a software complexity measure. We always show the original condition and then the translation back to an empirical condition by an implication =>.

Axiom 1:
$F \neq P1 \Rightarrow \mu(F) > \mu(P1) \Rightarrow F \bullet\!> P1$

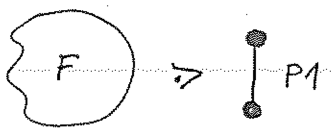


Figure 3.12: Axiom 1.

Axiom 1 means, that every arbitrary flowgraph F is more complex than a trivial flowgraph P1.

Axiom 2:
$\mu(F \circ G) > \max\ (\mu(F),\mu(G)) \Rightarrow F \circ G\ \bullet\!>\ \max\ (F,G)$

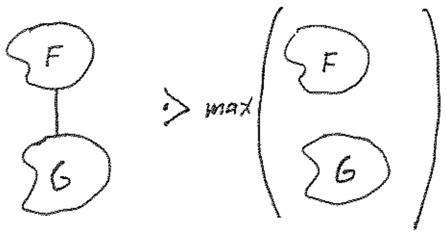


Figure 3.13: Axiom2.

Axiom 2 says that a sequence is more complex than the maximal single component.

Axiom 3:
$\mu(F \circ G) = \mu(G \circ F) \Rightarrow F \circ G \approx G \circ F.$

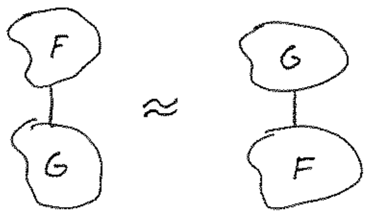


Figure 3.14: Axiom 3.

Axiom 3 is the weak axiom of commutativity. It is required by Bache.

strongest condition for concatenation operations. We show the conditions of the extensive structure in the next Section.

3.4 Conditions for Program Comprehension from the Extensive Structure

In measurement theory (Krantz, Luce, Suppe, Patric, Tversky, 1971), p.74, (Zuse, 1991), (Zuse, Bollmann, 1992), (Zuse, 1992) we can find empirical conditions which are denoted as the extensive structure. They also can be interpreted as conditions for program comprehension. The extensive structure consists of the following axioms:

A1': (P,•≥) 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 •≥ P2=> P1 o P3 •≥ P2 o P3 axiom of weak monotonicity
A5': If P1 •> P2 then for any P3, P4 there exists a natural
number n, such that nP1 o P3 •> nP2 o P4, Archimedian Axiom

We now consider the empirical conditions A1'-A5' in detail. The weak order (A1') describes the ranking order of objects (See Section 2.3). The prerequisites for the weak order are the axioms of transitivity and completeness.

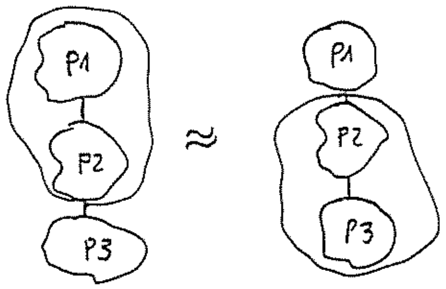


Figure 3.8: Axiom of weak associativity.

The axiom weak associativity is not discussed in literature as a condition for program comprehension/complexity. This empirical condition is not questionable because both flowgraphs are identical.

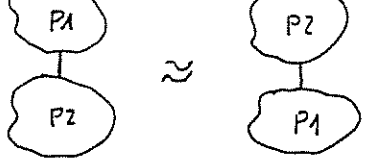


Figure 3.9: Axiom of weak commutativity.

This empirical condition is questionable because many author require this condition and other authors reject this condition for program complexity/ comprehension (Zuse, 1991), p.534.

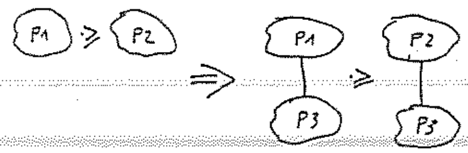


Figure 3.10: Axiom of weak monotonicity.

This empirical condition is similar to C3.

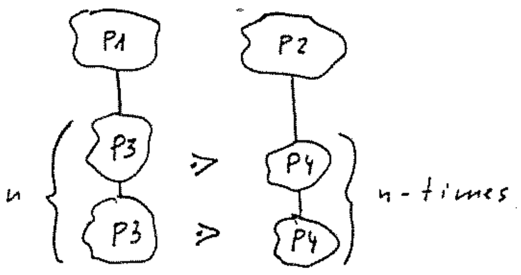


Figure 3.11: Archimedian axiom.

The axioms of the extensive structure are very strong axioms. As shown in Figure 3.7 the axioms of the extensive structure include the conditions C1-C4.

3.5 Weyuker's View of Program Comprehension

In this Section it is shown that there are more ideas of program complexity or program comprehension. Weyuker (Weyuker, 1988) discusses desireable properties for software complexity measures. Following our approach (Zuse, Bollmann, 1992), (Zuse, 1992) we translate most of the Weyuker properties back into empirical properties by an implication =>, where o is again the sequential concatenation operation. We only discuss the four most important requirements of Weyuker.

3.5.1 Weak Positivity

Weyuker requires weak positivity. It is defined as

P ≤• P o Q,

and

Q ≤• P o Q.

Positivity is defined as (Krantz, Luce, Suppe, Patric, Tversky, 1971) (p.73): P o P' •> P, for all P, P'∈**P**. Weyuker requires this axiom here in a weaker form (•≥ instead of •>). The idea behind this axiom is that Weyuker means adding something to a program makes it more complex or more difficult to comprehend.

3.5.2 Rejection of Condition C1

Weyuker rejects the independence condition C1 and requires

$$P \approx Q \Rightarrow \neg(R \circ P \approx R \circ Q).$$

Doing this Weyuker also rejects the conditions C2-C4 and the extensive structure as conditions for program complexity/comprehension.

3.5.3 Rejection of Weak Commutativity

Weyuker rejects the axiom of weak commutativity which is also an axiom of the extensive structure. The axiom of weak commutativity is defined as

$$P \circ Q \approx Q \circ P,$$

for all P, Q∈**P**.

Weyuker requires

$$P \circ Q \neg\approx Q \circ P,$$

Rejecting the axiom of weak commutativity means also rejecting the extensive structure.

3.5.4 Requiring the Extensive Structure

Weyuker requires, among others, the extensive structure because she discusses the statement

$$\mu(P1 \circ P2) = \mu(P1) + \mu(P2),$$

where μ is a software complexity measure. With this statement Weyuker requires the extensive structure. And, Weyuker requires independence because the conditions C1-C4 are included in the extensive structure (Zuse, 1992).

There is a contradiction since Weyuker rejects weak commutativity and requires the extensive structure. Both is not possible. This shows that conditions which are derived from software complexity measures can be interpreted from a measurement theoretic view.

3.5 Required Properties by Bache

Bache (Fenton, 1991), p.218, suggests axioms of program complexity. Bache formulates his axioms numerical, but using measurement theory it is possible to translate the numerical conditions back to empirical conditions by the implication =>. Again, o is the sequential concatenation operation BSEQ. Bache denotes F, H, G as arbitrary flowgraphs and P1 as an trivial flowgraph consisting of one edge and two nodes. For his considerations Bache assume that holds

$$\mu(H) > \mu(G) \Rightarrow H \bullet\!> G.$$

That means flowgraph H is always more complicated or less to comprehend than flowgraph G. μ is a software complexity measure. We always show the original condition and then the translation back to an empirical condition by an implication =>.

Axiom 1:
F≠P1 => μ(F) > μ(P1) => F •> P1

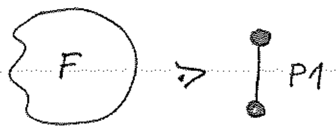


Figure 3.12: Axiom 1.

Axiom 1 means, that every arbitrary flowgraph F is more complex than a trivial flowgraph P1.

Axiom 2:
μ(F o G) > max (μ(F),μ(G)) => F o G •> max (F,G)

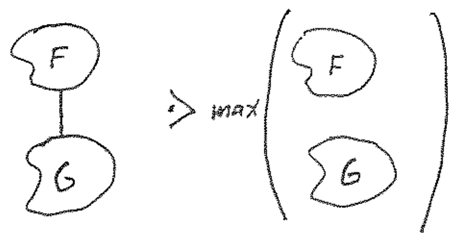


Figure 3.13: Axiom2.

Axiom 2 says that a sequence is more complex than the maximal single component.

Axiom 3:
μ(F o G) = μ(G o F) => F o G ≈ G o F.

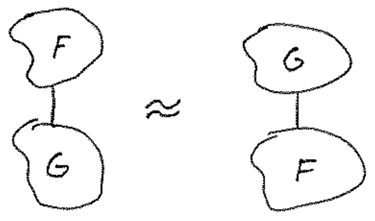


Figure 3.14: Axiom 3.

Axiom 3 is the weak axiom of commutativity. It is required by Bache.

Axiom 4:
μ(F o H) > μ (F o G) => F o H •> F o G.

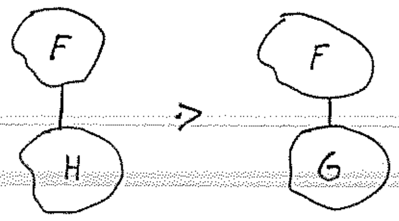


Figure 3.15: Axiom 4.

Axiom 4 is another type of the axiom of monotonicity than the axiom of weak monotonicity as defined in the extensive structure. Bache requires here > but does not consider the case of =.

Axiom 5:
μ(F(F1 on a)) = μ (F(F1 on b)) => F(F1 on a) ≈ F(F1 on b), where F(F1 on b) means F1 is nested in F on node b.

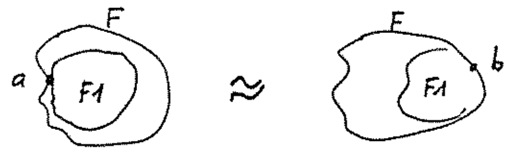


Figure 3.16: Axiom 5.

Axiom 6:
μ(F(H,F2,..,Fn)) > μ(F(G,F2,..,Fn))=> (F(H,F2,..,Fn)) •> (F(G,F2,..,Fn)),
where F(G......) means G is nested in F.

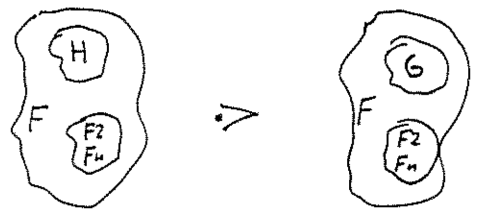


Figure 3.17: Axiom 6.

Axiom 6 says that the complexity/comprehension of a program is independent of the node where a flowgraph is nested in another one.

Axiom 7:
μ(H(F))>μ(G(F))=> (H(F)) • > (G(F)),
where H(F) means that F is nested in H on an arbitrary node.

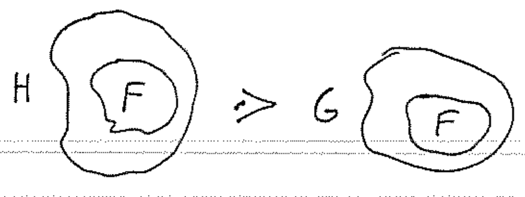


Figure 3.18: Axiom 7.

With axioms 6 and 7 Bache discusses nesting properties.

Axiom 8:
μ(F(G)) > μ (F o G), that immediately implies μ(F(G)) > max (μ(F),μ(G))=> (F(G)) •> max (F,G)

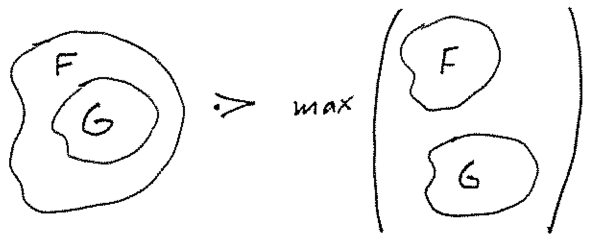


Figure 3.19: Axiom 8.

Axiom 8 says that a nested flowgraph is more complex than the maximum of the single parts.

Axiom 9:
μ(H(G)) > μ(G(H)) => H(G) •> G(H).

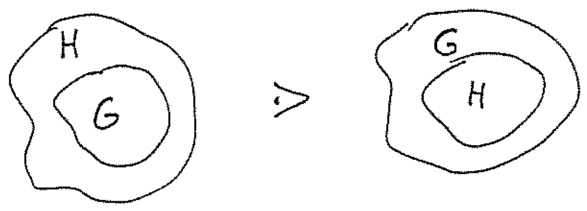


Figure 3.20: Axiom 9.

Axiom 9 is also a condition for nested structures.

The Axioms 1-9 show that program complexity can be described by empirical axioms. Bache (Fenton, 1991) presents the VINAP measures which fulfil the axioms above. It is important to notice that the VINAP measures also assume the extensive structure. This shows again that in the area of software metrics many empirical conditions about program complexity/comprehension can be found.

3.7 Program Comprehension behind the Metric of McCabe

We showed that in the area of software complexity metrics implicitly empirical conditions are discussed. Using measurement theory and assuming a homomorphism we can interpret the idea of program complexity/comprehension behind a software complexity measure. We illustrate this with the Measure of McCabe.

The idea of program comprehension/complexity behind the Measure MCC-V2=|E|-|N|+1 of McCabe, where E is the set of edges and N the set of nodes, is the following (Zuse, 1991), (Bollmann-Sdorra, Zuse, 1992). P and P' are arbitrary 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 ranking order of the Metric of McCabe related to the term complexity/comprehension.

Because the Metric of McCabe is additive related to the concatenation operation BSEQ, the Metric of McCabe also assumes the axioms of the extensive structure and the conditions C1-C4.

4 Conclusion

Program comprehension and program complexity are similar terms. Because program complexity is analyzed with software complexity measures we derive empirical conditions for the term program comprehension from numerical conditions of software complexity measures. This is possible because measurement theory allows under certain conditions (homomorphism, etc.) the translation of numerical conditions back to empirical conditions. It is shown that the term program comprehension can be discussed together with software complexity metrics. This approach helps to describe the empirical term program comprehension with empirical conditions (axioms).

5 References

Bollmann, Peter; Zuse, Horst:
An Axiomatic Approach to Software Complexity Measures. Proceedings of the Third Symposium on Empirical Foundations of Information and Software Science III, Roskilde, Denmark, October 21-24, 1985. Reprinted in: Empirical Foundations of Information and Software Science III, Edited by Jens Rasmussen and Pranas Zunde, Plenum Press, New York and London, 1987, pp.13-20.

Bollmann-Sdorra, P.; Zuse, Horst:
Supplement to: "Horst Zuse: Software Complexity - Measures and Methods" Proof of Theorems (Part I). Technical Report, Technische Universität Berlin, Franklinstraße 28-29, FR 5-3, 1 Berlin 10, Germany, 1992.

Fenton, Norman:
Software Metrics: A Rigorous Approach. City University, London, Chapman & Hall, 1991.

Fenton, Norman; Hill, Gillian:
Systems Construction and Analysis - A Mathematical and Logical Framework. McGraw Hill Publisher, 1993.

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

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

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

McCabe, T.:
A Complexity Measure. IEEE Transactions of Software Engineering. Vol. SE-1, No. 3, pp. 312-327, 1976.

Myers, G.J.:
Software Reliability - Principles and Practices. Wiley & Sons, 1976.

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

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

Wakker, Peter, P.:
Additive Representations of Preferences - A New Foundation of Decision Analysis. Kluwer Academic Publisher, 1989.

Yourdon, E; Constantine, L.:
Structured Design - Fundamentals of a Discipline of Computer Programs and Design. Prentice-Hall, 1979.

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 1989.

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

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. Springer Publisher, Springer Verlag London Ltd, Springer House, 8 Alexandra Road, Wimbledon, London SW19 7JZ, UK, October 1992.

Zuse, Horst:
Properties of Software Measures. Software Quality Journal, Vol 1, December 1992, pp. 225-260.

A Process Algebra Based Program and System Representation for Reverse Engineering *

E. Merlo[1,2], R. DeMori[1,2] K. Kontogiannis[1]

[1] McGill University
3480 University St., Room 318, Montréal, Canada H3A 2A7

[2]Centre de Recherche Informatique de Montréal
1801 McGill Collége, Bureau 800, Montréal, H3A 2N4, Canada

Abstract

A reverse engineering approach based on process algebras for system representation and understanding is presented. Process algebras offer both a formal framework for representing communicating processes and a proof theory for proving semantic equivalences between them. Programs and program fragments are denoted as concurrent agents and code behaviour is defined in terms of interactions among agents in a process algebra representation suitable for subsequent analysis. Semantic and behavioural equivalences between programming plans, which represent programming stereotypes, and code fragments can be defined in this formal system together with a deduction system to prove them. Several advantages and further research issues on the use of process algebra for reverse engineering and maintenance are identified and discussed.

1 Introduction

This paper describes an approach for program and system representation and understanding, based on process algebras to represent and analyze the behaviour of source code. Program understanding is a tedious and time consuming phase of program maintenance, particularly when the person involved is not the author. Impact/change analysis; intermodule communication; classification of what a given code fragment does; investigation of similarities between code fragments; and path analysis, were some of the problems identified by IBM researchers and programmers, and constitute the basis of our research objectives. Our research efforts focus first, on structural recognition of code and second, on formalisms and methods for behavioural recognition.

Structural recognition is achieved at lower levels of abstraction using the information provided by the abstract syntax trees, and by control and data flow graphs. The structure of these graph representations is the basis for program understanding, [14]. It provides essential information for representing code fragments as functional objects.

Behavioural recognition is achieved at a higher level of abstraction. The dominant structures at this level are algebraic languages for processes, interpretations of these languages, source code fragments viewed as processes, and well defined communication mechanisms between these processes. Program statements are denoted as processes [16], [3], and source code is represented in a process description language. Processes which correspond to source code fragments are linked through communication paths which describe control and data flow. Behavioural equivalences of processes can be defined within the possible interpretations of such an algebraic process description language. In [16] it is shown how a programming language is denoted into CCS constructs. The translation occurs in a phrase-by-phrase fashion, and it is compositional. In such a way, the programming language is given an unambiguous meaning. Extensions to include the process algebra denotation of programming constructs typical of a modular imperative language are given in [13] together with examples. The program fragments can be classified and recognized by matching or by proving their equivalence to predefined programming plans and specifications. At this level, process behaviour is captured by using labeled transition graphs and derivation trees. The recognition process becomes a problem solving activity for proving process equivalences.

*This work is funded by IBM Canada Ltd. Laboratory - Center for Advanced Studies (Toronto).

Because of complexity issues and limits of current theories, paradigms for program understanding do not usually cover all the situations which have to be analyzed. Partial recognition has to be performed and semantic distances, defined for different recognition plans, can be used to score the alternative recognition hypotheses. User interaction could also make feasible certain analysis algorithms whose complexity makes them otherwise unacceptable for programs and systems of substantial size. This approach is used for developing software maintenance tools for PL/AS [1] programs at IBM Toronto research laboratories.

2 Plan based program understanding techniques

A number of research teams have addressed the issue of program representation and understanding using plan based techniques. The basic idea in all of these approaches is to syntactically match some code description to a cliché description, and to perform abstractions using rewrite rules. Our approach replaces the syntactic matching of descriptions with proving equivalence relations. Program equivalence is proven by establishing the equivalence between processes using criteria such as observational equivalence and bisimulation. This results in dynamic matching based on the behavioural properties of the programs rather than the syntactic properties of their representation.

The Program Recognizer [22] uses a library of clichés to identify fragments and data structures which appear in the code. It is based on Plan Calculus [21] which represents source code at a higher level of abstraction. Plan calculus program representations are encoded as flow graphs which are parsed in order to produce design trees. Cliché recognition is achieved by identifying subgraphs in the Plan Calculus representations, and replacing them by more abstract operations.

CPU [15], uses lambda calculus to represent programs and reason about them. Source code is translated into lambda expressions in a preprocessing phase. Then using a knowledge base of rewrite rules, it applies all applicable transformations, until expressions can be no further transformed. The sequence and the nature of the transformations represent the design decisions and provide a hierarchical description of the causal connections of plans at different levels of abstraction.

Pat,[19] uses a Plan Parser to generate a set of rules that contain understanding, paraphrasing and debugging knowledge along with a deductive inference engine. A Program parser creates events or facts from source code. These are used to fire the rules that recognize the plans in the program. The whole understanding process is recorded by a justification-based truth maintenance system.

A program understanding method based on proper decomposition for large, unstructured, imperative programs is presented in [6]. Source code is represented by a hierarchical data flow / control flow graph, corresponding to an abstract model which is language independent. Plan instances and programming concepts are recognized by matching parts of the abstract program representation against a library of standard implementation plans. Decomposition principles are used to break complex problems into smaller subproblems.

The points that make our algebraic approach attractive are (1) the uniform way in which systems can be described, from simple programming statements, to operating systems and computing environments; (2) the compositional way the behaviour of any code fragment is defined in terms of its subcomponents; (3) the ability to define ordering relations between code fragments when equivalences can not be proven; and (4) the ability to denote recursive and interactive programs.

3 Process Calculus and Program Representation

Our approach uses the theory of Concurrent Communicating Systems (CCS) [16] which incorporates an algebraic language for processes, a deduction system for reasoning about processes based on a set of equations, and a class of equivalence relations.

The key point in this calculus is the notion of an *agent*. An agent models and describes a specific part of a complex system. Communication is achieved by *actions* that agents may perform. Each agent is capable of performing an action to interact with its neighbouring agents, which occurs independently and concurrently with other agents' actions. Communication between agents is achieved by linking input and output ports of agents. Handshakes through complementary ports are represented by a special action denoted by

[1] PL/AS is a copyright of IBM Corp.

the *silent action* τ. Thus, the behaviour of a system is defined in terms of its entire communication capabilities, or alternatively, in terms of what is observable in such a system. Equations are used to represent process behaviour. For example the equation A = b.B + c.C is interpreted as : *Agent A is capable of performing action b and then behaving like agent B, or A is capable of performing action c and then behaving like agent C.* The basic constructors for agent expressions are :

- The inactive agent 0
- Constant A *(the Agent A)*
- Prefix a.E *(do Action a and behave like Agent E)*
- Summation E1 + E2 *(behave like Agent E1 or Agent E2)*
- Composition E1 | E2 *(Agent E1 composed with Agent E2. The Agents may proceed independently or may interact through complementary ports)*
- Restriction E \ L *(Actions in the set L, are visible only within the scope of Agent E)*
- Relabelling E[f] *(Port renaming of Agent E according to function f. It is used for communicating Agents which share no complementary ports)*
- Substitution E/X *(Substitute X by E)*
- Recursion fix(X = E) *(the Agent X such that X = E)*

Equations are not the only means to represent an Agent's behaviour. *Labelled transition systems, process algebra flow graphs*, and *Transition trees* are used as well.

A *Labelled Transition System* is a triplet (S, A, $\xrightarrow{\alpha}$) where S is a set of process algebra *states*, A, a set of *actions*, and $\xrightarrow{\alpha}$, a state transition function.

Process Algebra Flow Graphs (PAFG) are tools to understand complex processes. PAFGs consist of nodes which correspond to processes, and arcs which join nodes. Each node has as many internal labeled ports as the corresponding actions emanating from this node. Each port may have external labels defined by renaming its internal labels. Arcs join two ports at different nodes with either no external labels or complementary labels. In Fig. 1, such a PAFG for basic programming language constructs is illustrated.

Nodes in Fig. 1 represent the following agents : *a)* identifier registers (Reg_i, Reg_x); *b)* identifier references (Denote[i]); *c)* function symbols (Denote[:=]) and finally; *d)* control (Done).

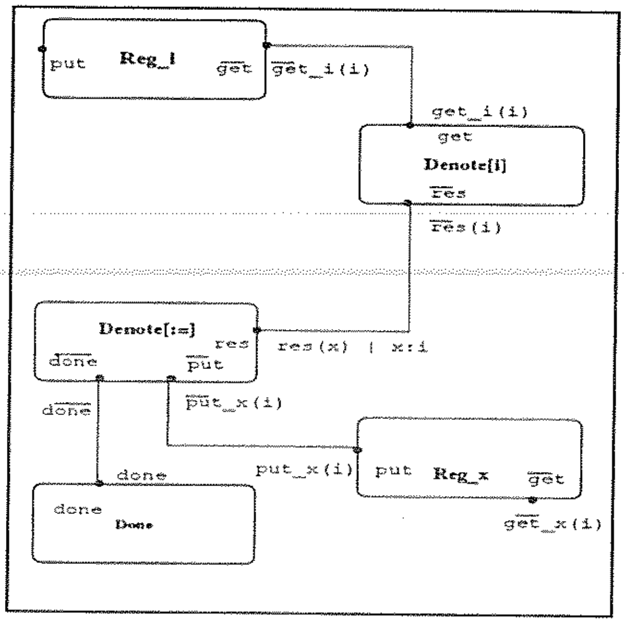


Figure 1: Process Algebra Flow Graph for the statement x := i

Arcs in Fig. 1 represent communication links between complementary communication ports. The value of register i is read through the get_i port and it is passed to the function := via its *arg1* port which is renamed as *res*. The := process updates the value of the register referring to identifier x by issuing a $\overline{put\_x}$ action through its renamed $\overline{res}$ port. The control process *Done* signals the termination of the statement *x:=i*.

Derivation Trees are tree structures which state all the transitions that may occur in a system. Whenever E $\xrightarrow{\alpha}$ E', E' is called an α-derivative of E. *Derivation trees* are of the most interest to our work as the meaning of an agent is a property of its derivation tree. Nodes correspond to processes, and arcs from each non-leaf node correspond to all possible actions that emanate from this node. A possible interpretation of the algebraic language in terms of derivation trees is shown in Fig. 2.

4 Equivalence Relations over processes

Basic CCS calculus defines equivalence relations for processes based on bisimulation. The intuitive idea behind bisimulation is that two processes can be characterized as different if an external observer can detect a difference between them.

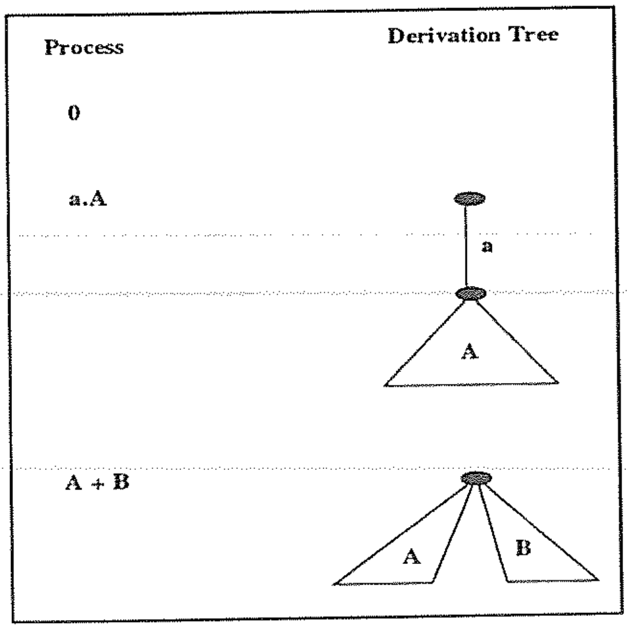


Figure 2: Derivation trees for basic language constructors

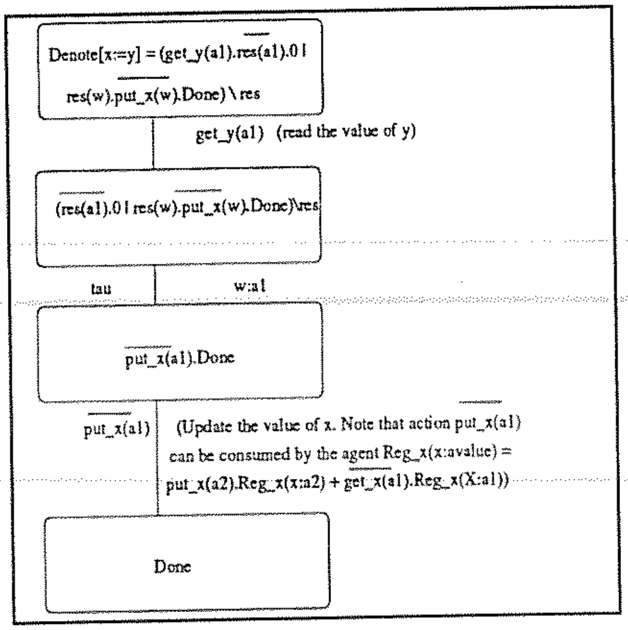

Denote[x:=y] = (get_y(a1).res(a1).0 | res(w).put_x(w).Done) \ res
get_y(a1) (read the value of y)
(res(a1).0 | res(w).put_x(w).Done)\res
tau
w:a1
put_x(a1).Done
put_x(a1)
(Update the value of x. Note that action put_x(a1) can be consumed by the agent Reg_x(x:avalue) = put_x(a2).Reg_x(x:a2) + get_x(a1).Reg_x(X:a1))
Done

Figure 3: Update/Initialize plan as a derivation tree for the expression x:=y

More formally, a bisimulation is a binary relation R over processes such that $(P,\ Q) \in R$ if and only if:

- whenever $P \xrightarrow{\alpha} P'$ then for some Q', $Q \xrightarrow{\alpha} Q'$ and $(P',\ Q') \in R$
- whenever $Q \xrightarrow{\alpha} Q'$ then for some P', $P \xrightarrow{\alpha} P'$ and $(P',\ Q') \in R$

Thus, two processes, P and Q, are equivalent if and only if for all $\alpha \in Act$ each α - derivative of P is equivalent to some α - derivative of Q, and conversely. If the silent action τ is treated as any other possible action in the system, we arrive at the notion of strong bisimulation (strong equivalence). If τ is treated as a special action (unobservable), then weak bisimulation (observational equivalence) is defined.

The relationship between bisimulation and testing equivalence is defined in [1] where it is shown how bisimulation equivalence relations can be based on testing.

Program understanding is an activity which can benefit from the semantics of the language in which the programs are written. There are several methodologies for representing the semantics of programming languages. These include denotational semantics, operational semantics [18], lambda calculus, Hoare's Logic [10], Dynamic Logic [9] and CCS [16].

5 Program and System Understanding using algebraic languages for processes

Our objective in system understanding is to provide a means and tools to a system maintainer to analyze a system's behaviour. At program level, two programs are considered equivalent if their corresponding representations in process algebra can be proven equivalent. In the context of program understanding, this approach involves proving the equivalence between process algebra descriptions of source code and programming plans. Programming plans can be expressed as process algebra derivation trees as well. For example the initialize/update plan is given as a process algebra derivation tree in Fig. 3. The expression $x{:=}y$ is analyzed as a process algebra agent. First the right hand side of the expression is evaluated (get_y($a1$)) and the result is passed via the *res* action to the agent which updates the value of register x, with the $\overline{put\_x}(a1)$ action. Process algebra equations, derivation trees and PAFGs can be extracted from the abstract syntax tree of the code being analyzed.

Derivation trees are of particular interest as bisimulation equivalence relations can be defined on them. An example of the use of observational congruence for proving that two code fragments are equivalent is shown in Fig. 4 and in Fig. 5. Two IF-THEN-ELSE statements are proven equivalent after they have been

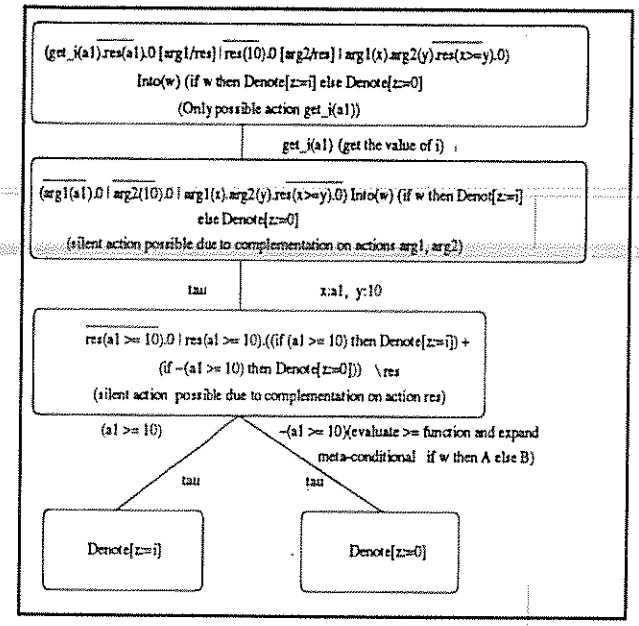

(get_i(a1).res(a1).0 [arg1/res] | res(10).0 [arg2/res] | arg1(x).arg2(y).res(x>=y).0)
Into(w) (if w then Denote[z:=i] else Denote[z:=0]
(Only possible action get_i(a1))
get_i(a1) (get the value of i)
(arg1(a1).0 | arg2(10).0 | arg1(x).arg2(y).res(x>=y).0) Into(w) (if w then Denot[z:=i]
else Denote[z:=0]
(silent action possible due to complementation on actions arg1, arg2)
tau
x:a1, y:10
res(a1 >= 10).0 | res(a1 >= 10).((if (a1 >= 10) then Denote[z:=i]) +
(if ~(a1 >= 10) then Denote[z:=0])) \res
(silent action possible due to complementation on action res)
(a1 >= 10)
~(a1 >= 10)(evaluate >= function and expand
meta-conditional if w then A else B)
tau
tau
Denote[z:=i]
Denote[z:=0]

Figure 4: Derivation tree for the expression IF ($i >=$ 10) THEN z:=i ELSE z: =0

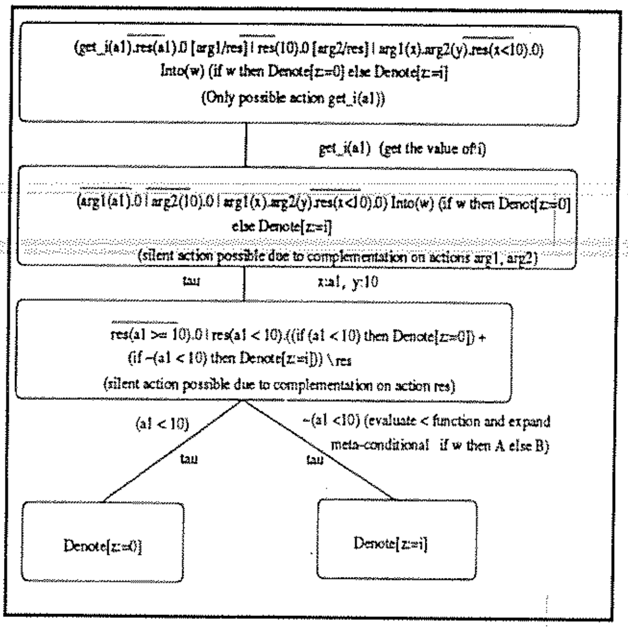

(get_i(a1).res(a1).0 [arg1/res] | res(10).0 [arg2/res] | arg1(x).arg2(y).res(x<10).0)
Into(w) (if w then Denote[z:=0] else Denote[z:=i]
(Only possible action get_i(a1))
get_i(a1) (get the value of i)
(arg1(a1).0 | arg2(10).0 | arg1(x).arg2(y).res(x<10).0) Into(w) (if w then Denot[z:=0]
else Denote[z:=i]
(silent action possible due to complementation on actions arg1, arg2)
tau
x:a1, y:10
res(a1 >= 10).0 | res(a1 < 10).((if (a1 < 10) then Denote[z:=0]) +
(if ~(a1 < 10) then Denote[z:=i])) \res
(silent action possible due to complementation on action res)
(a1 < 10)
~(a1 <10) (evaluate < function and expand
meta-conditional if w then A else B)
tau
tau
Denote[z:=0]
Denote[z:=i]

Figure 5: Derivation tree for the expression IF ($i < 10$) THEN z:=0 ELSE z:= i

analyzed as process algebra derivation trees.

At any given node of the tree, the actions emanating from that node can be bisimulated by the the actions of the corresponding node of the other tree.

For the trees in Fig. 4 and in Fig. 5 to be equivalent in the full value passing calculus, the proof of $(a1 < 10) \equiv \neg(a1 >= 10)$ must be incorporated as well. Another example for proving equivalence between a structured GOTO statement combined with an IF-THEN statement, and a WHILE statement is shown in Fig. 6 and in Fig. 7. The process algebra derivation trees are used to prove observational equivalence between these two statements. In Fig. 6, Denote[Cond] process, which can be represented as a process algebra derivation subtree on its own, terminates by issuing a $\overline{res(T)}$ or $\overline{res(F)}$ action. These actions are consumed by Denote[Body] and the *Done* process respectivelly. The GOTO is translated as a special node in the derivation tree called *Iterate*, pointing to the root of the tree, thus implementing the unfolding of the statement as indicated by the GOTO label. The Denote[Body] process can be represented as a process algebra derivation subtree as well. The last action it performs is a $\overline{done}$ action, which is then consumed by the *Iterate* process using the *before* operator.

The *before* operator links two processes in such a way that when the first terminates the second starts. Its process algebra equation is given in [16] and is defined as :

$$\textbf{A before B} \overset{\text{def}}{=} (\text{A[link/done]} \mid \text{link.B}) \setminus \{\text{link}\}$$

where *done* is the last action performed by process A and link is a new port introduced in process B.

In Fig. 7 the same *Cond* and *Body* statements are translated as process algebra derivation subtrees and linked according to the actions performed, resulting in a derivation tree which is observationally equivalent to the one in Fig. 6. The *fix* operator is used to represent recursive equations of the form A $\overset{\text{def}}{=}$ B, where B is of the form E{A/X} and E contains the variable X; that is A occurs in B. By setting A $\overset{\text{def}}{=}$ E{A/X}, one is looking for a unique solution to the equation X = E [16].

The advantages of considering equivalence relations and partial orders for program understanding within the framework of algebraic languages for processes are based on two criteria. The first criterion is related to the power and formality of process algebras. Such a formalism offers not only expressive power for program representation, but also a complete set of combinators and equational laws for reasoning on processes. It is possible to define equivalences between processes so that make program understanding and classification easier.

The second criterion deals with the application of process algebras for program understanding. Information hiding as well as composition are easily encoded by using the restriction and the composition opera-

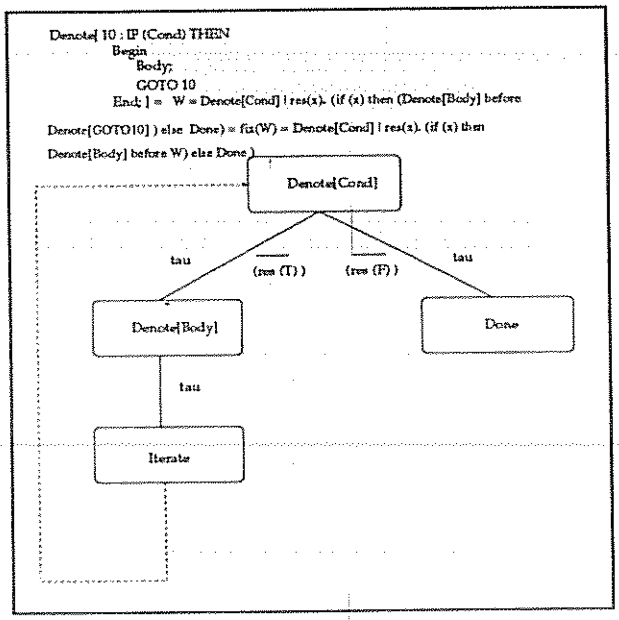

Denote[10 : IF (Cond) THEN
Begin
Body;
GOTO 10
End;] = W = Denote[Cond] ! res(x). (if (x) then (Denote[Body] before
Denote[GOTO10]) else Done) = fix(W) = Denote[Cond] ! res(x). (if (x) then
Denote[Body] before W) else Done)
Denote[Cond]
tau
(res (T))
(res (F))
tau
Denote[Body]
Done
tau
Iterate

Figure 6: Derivation tree for the statement : 10 : IF (Cond) THEN Begin Body; Goto 10 End;

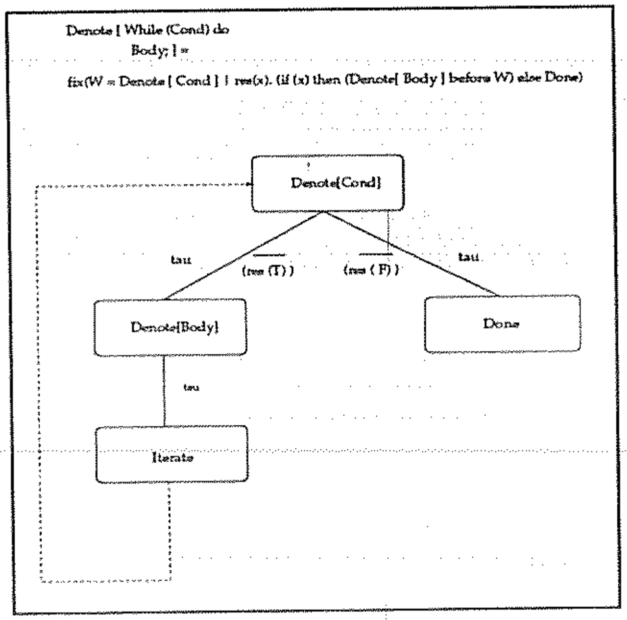


Figure 7: Derivation tree for the statement : WHILE (Cond) DO Body;

tors. Composition, in particular, is an interesting feature of the calculus since it allows program behaviour to be defined in terms of the behaviour of its subcomponents.

Finally, it is more natural to think of programs in terms of communicating processes than in terms of plain functions or structural descriptions. The process algebra approach is intrinsically scalable up and is uniform as well. The whole program environment - from the user or the operating system up to the actual source code (program fragments, procedures or modules) - is viewed and encoded as processes and agents.

The ability of building abstractions is an inherent attribute of the process algebra approach. Unnecessary information due to format, comments, or syntactic sugar is eliminated as the denotation of such information is uniform in all of its possible variations.

Behaviour based concept representations of programs are used for hierarchical recognition in which already recognized concepts as well as programming structures can be used for further recognition. Process algebra program representations are abstracted as higher level agents for which actions correspond to a set of roles which describe the abstracted concept. Programming structures, procedures, or functions can be represented as single agents performing certain communication actions characterizing the abstracted concept they define.

Transformations on abstractions are straightforward as process algebra language allows for such manipulation. Software represented as process algebra derivation trees, process algebra flow graphs, or process algebra equations, can be transformed by transforming these process algebra expressions.

The drawback of this approach is that recognition still relies on programming plans and on complex process algebra equations. In the case of scattered code, a plan may be computationally expensive to match. In such cases, heuristic methods could be used to locate the possible matching plans. Moreover, recursion and iteration have to be examined carefully as there are undecidable problems related to loop and recursion termination.

Implementation issues arise when such a theory is applied to real problems. We have chosen the Refine [2] tool as a workbench for our approach. Refine provides an environment with a programmable language parser/printer, a software analysis and transformation system, and an X windows-based graphical interface toolkit. Code is represented as annotated, abstract syntax trees in an object base. Moreover, a high level Lisp based language combining logic programming, rule based programming, functional programming, and object oriented programming is provided. This language is used to query and update the object base representing the source code under analy-

[2]REFINE is a trademark of Reasoning Systems Corp.

sis.

Within this framework, process algebra equations and derivation trees can be constructed directly from the abstract syntax trees. Currently, our research focuses on denoting PL/AS language in a process algebra formalism, and devising appropriate equivalence and partial order relations in order to prove similarities or equivalences between code fragments and programming plans. Specifically, we have devised a formalism to represent process algebra equations as trees in the Refine environment, and we have denoted basic PL/AS statements and expressions in this formalism. Moreover, we have implemented some simple equivalence relations for matching structurally different code fragments. Currently, we have denoted declaration statements, identifier references, assignment statements, while statements, switch statements, If-Then-Else statements, and a number of expressions such as addition, substraction, division, multiplication, $\leq$ predicates, $\geq$ predicates, etc. A simple behavioural pattern matcher has been implemented and is used to localize structurally different, but behaviourally equivalent code fragments. This simple behavioural pattern matcher can prove the similarity between the following statements : (a) an IF-THEN-ELSE statement and its reverse ; (b) a CASE statement and a sequence of nested IF-THEN-ELSE statements; and (c) a WHILE statement and a combination of a structured GOTO statement with an IF-THEN statement. Since process algebras focus on communication, additional questions on the way software components communicate can be answered as well. Within this framework we have devised a set of programs for selecting code fragments satisfying a set of input/output criteria such as variables used and variables updated. Moreover, symbolic values are used to implement the value passing mechanism in process algebra calculus when no constant values can be calculated. Towards this line of research, we are considering the use of flow analysis in order to perform more complex tasks, such as answering questions on reaching definitions and performing alias analysis.

Quality improvement can be defined in terms of simplifying, modularizing, and documenting the system.

6 Research Issues

Research topics to be investigated for system representation and understanding include: recursion and iteration handling; partial behavioural matching; the complexity of the derived process algebra expressions; and the complexity of the matching process.

Recursion and iteration introduce two basic problems. The first problem is related to criteria for proving termination. Recursive and iterative code fragments which have to be identified and matched with recursive and iterative plans need not only be proven to terminate, but must also be be proven as behaviourally equivalent processes. Proving termination is generally an undecidable problem. The solutions we consider are derived from the areas of partial evaluation, symbolic execution and flow analysis of programs [12], [17]. Moreover, the extension of system responses by allowing *Do not Know* responses in addition to *Yes / No* responses is currently under investigation in the framework of loop and recursion analysis. The second problem is related to the structures used to represent iterative and recursive processes. In [15] and [22], the authors show how iterative programming structures are transformed into sequential ones with the use of programming plans. This is an attractive approach because we wish to avoid using any graph structure to represent programs. The reason for this is that proving isomorphism between graphs (programming plan representations and source code representations) is a problem of unknown complexity.

Partial recognition and partial behavioural matching address the problem of recognizing plan instances even when source code with no relevance to any programming plan is interleaved with the code fragment being analyzed. In order to accomplish this, distances and metrics for partial matching must be defined. Semantic distances and metrics can be used to score program descriptions that partially match programming plans. Similar techniques can be found in [4] and [5], where metrics are used to describe relations between different processes.

Current program understanding techniques may require solutions whose complexity is unacceptable for systems of substantial size. Reducing the complexity of the top-level control strategy for guiding the matching process of plan and code representations is an open issue and it involves the investigation of several techniques which include reduction of the search space, guidance of the matching algorithm, and user interaction. Top-level control strategies for focusing on program parts and for selecting programming plans include the use of domain knowledge [2], top-down goal agendas [11], best first search (TALUS), depth first recursive search [15], repeated traversals [20], and exhaustive parsing [22].

Finally, the existence of normal forms for all processes is an issue in the equational theory. If an algorithm to derive normal forms can be found, then programming plans are reduced to these normal forms, and bisimulation equivalence is reduced to simple pattern matching. Such a normal or *standard form* exists, but only for processes that contain finite summations, no recursion, and finitely many distinct derivatives. Thus, the objective is to exploit techniques for selecting only those specific, behavioural properties in recursive and iterative processes that can be represented in standard form.

7 Conclusion

This paper describes an approach for program and system representation using process algebras and CCS. The use of this representation for system understanding offers several advantages. Firstly, program behaviour is compositional and is given in terms of the behaviour of its subcomponents. Formalisms used to support such a program representation are derivation trees and process algebra equations. Secondly, semantic equivalences between process algebra descriptions of programming plans and source code can be defined. Process equivalence can be based on bisimulation and on the observable behaviour of every process. We investigate the idea of using equivalence relations and partial order relations in order to show semantic equivalences between programming plans and source code representations. Partial order relations can be used to describe properties of a program fragment when no perfect, behavioural matching between plans and source code descriptions is possible due to syntactic variations, implementation variations, noncontiguousness. and unrecognizable code. Thirdly, interactive systems can be described and represented. Finally, it is a uniform, natural, and scalable approach since program statements, procedures, modules, programs, operating systems, and users are all viewed as communicating agents. Therefore there is no distinction between program and system understanding.

The critical problem of the completeness of programming plans seems to be overwhelming for efficient plan based program understanding. The number of programming plans required to cover all possible, realistic behaviours that can occur - let alone determine them - imposes a serious limitation towards automatic program understanding. An alternative solution could be incremental recognition in a goal oriented reverse engineering environment driven by an efficient query system. Complete plan recognition is difficult to achieve and reverse engineering should be an incremental and a goal directed process. Reverse engineering an entire system is expensive and this effort should be goal oriented. Goals can be set by the maintainer and expressed as a series of queries. Queries can be used to locate code fragments satisfying a specific behaviour without considering possible variations due to implementation and syntactic differences between code fragments. Such queries may seek code that updates a specific variable, interfaces with a given module, or performs a particular sequence of actions, implementing a specific algorithm. Thus, plan recognition becomes a goal driven, user assisted process.

Open research issues in this framework involve iteration and recursion handling for which termination proofs are required, as well as the definition of partial order relations which allow for partial behavioural matching. Alternative solutions based on static execution, induction, partial evaluation, and flow analysis are considered. This approach is being applied for analyzing and understanding PL/AS programs at IBM Toronto research laboratories.

Acknowledgement

We would like to thank E. Buss and J. Henshaw, from IBM CAS Toronto, for the deep insights our conversations gave us.

References

[1] S. Abramsky, "Observational Equivalence as a Testing Equivalence," *Imperial College Technical Report.*

[2] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, July 1989, pp. 36 - 49.

[3] T. Bolognesi, E. Brinksma, "Introduction to the ISO specification language LOTOS," *Computer Networks and ISDN Systems 14 (1987) pp25-59.*

[4] J. deBakker, J. Zucker, "Processes and the denotational Semantics of Concurrency," *Information and Control, 54, pp70-120.*

[5] G. Golsen, W Rounds "Connections between Two Theories of Concurrency : Metric Spaces and Synchronization Trees," *Information and Control, 57, pp102-124.*

[6] J. Hartman, "Understanding Natural Programs Using Proper Decomposition" *13th Interna-*

tional Conference on Software Engineering, 1991, Austin, Texas, pp. 62-73.

[7] M. C. Hennessy, "Algebraic Theory of Processes," *MIT Press, 1988.*

[8] M. C. Hennessy, G. Plotkin, "A term model for CCS" *LNCS 88.*

[9] C. A. R. Hoare, "Communicating Sequential Processes," *CACM, Vol.21, pp666-677, 1978.*

[10] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *CACM, Vol.12, pp576-580, 1969.*

[11] W. L. Johnson, "Understanding and Debugging Novice Programs," *Artificial Intelligence 42 (1990) pp.51-97.*

[12] J. King, "Symbolic Execution and Program Testing," *CACM, July 1976, Volume 19, Number 7, pp385-394.*

[13] K. Kontogiannis, "Toward Program Representation and Program Understanding using Process Algebras," *CASCON 92,* IBM Canada Ltd. Laboratory — Center for Advanced Studies, November 9-12 1992 , pp.299-317.

[14] W, Kozaczynski, J. Q. Ning, A. Engberts, "Program Concept Recognition and Transformation," *IEEE Transactions on Software Engineering Dec. 1992, Vol. 18, No. 12, pp1065 - 1075*

[15] I. S. Letovsky, "Plan Analysis of Programs," *Ph.D thesis YALEU/CSD/RR/662, Yale University, Dept. of Computer Science.*

[16] A. J. R. G. Milner, "Communication and Concurrency," *Prentice Hall, 1989*

[17] S. Muchnick, N. Jones, "Program Flow Analysis, Theory and Applications," *Prentice Hall 1981.*

[18] G. D. Plotkin, "A Structural Approach to Operational Semantics", *Lecture Notes, University of Aarhus, Denmark.*

[19] J. Q. Ning, M. T. Harandi, "Knowledge-Based Program Analysis," *IEEE Software January 1990, pp74-81.*

[20] A. Quilici, J. Khan, "Extracting Objects and Operations from C Programs," *Workshop notes AI and Automated Program Understanding, AAAI'92, pp.93-97.*

[21] C. Rich, R. C. Waters, "Intelligent Assistance for Program Recognition, Design, Optimization, and Debugging", *Memo, MIT AI Lab,* Jan 1989.

[22] L. M. Wills, "Automated Program Recognition: A Feasibility Demonstration," *Artificial Intelligence,* Vol. 45, No. 1-2, Sept. 1990.

Understanding Lolita: Program Comprehension in Functional Languages

J.E. Hazan*, S.A. Jarvis, R.G. Morgan and R. Garigliano
Artificial Intelligence Systems Research Group
University of Durham, DH1 3LE, UK

Abstract

Traditional arguments concerning the comprehensibility of functional programs have been illustrated with trivial examples. In this paper, we present the real-life example of a large system for natural language processing which has been programmed entirely in a lazy functional language. This system is undergoing constant change as new features are added to different areas. We present a series of case studies which illustrate various aspects of the maintenance task, including reuse of existing parts of the system and the integration of the new features. We explain how the choice of a functional language for programming the system has aided in the comprehension of the system by new programmers and how this in turn has led to the simplification of the maintenance task. We describe the ease with which new features have been integrated into the system and relate this to the careful design of abstractions within a functional programming framework.

1 Introduction

It has often been argued that it is easier to write in a functional programming language than in an imperative language. Higher-order functions and lazy evaluation allow new levels of modularity to be attained [1]; this in turn enables programs to be more easily read and understood. Lack of side-effects make the properties of the program easier to reason about. The similarity to mathematical notation can be considered an advantage to those with a knowledge of such notation. In addition, programmers do not need to concern themselves with storage management; the program is thus free from memory allocation statements and variable declarations.

In this paper, we shall study some of the features of functional programming which make it particularly suitable for developing large, real-life systems. In particular, we shall concentrate on the reasons why these features make the program easier to comprehend and hence simplify the maintenance task. In this introduction, we shall introduce our large-scale application and some of the difficulties facing its maintainers. In Section 2, some of the features of the functional programming language we have used are explained and in Section 3 we shall present several case studies in which these features of functional programming have aided the task of maintaining the application.

1.1 The Lolita system

The Artificial Intelligence Systems Research Group at the University of Durham has developed the Lolita system for natural language processing applications [2]. The system consists of 12,000 lines of source code (not including comments; with comments, the system consists of approximately 23,000 lines), equivalent to about 120,000 lines of imperative code [3]—divided between fifty modules. In addition there are around 450 data files. Although the system was initially developed by one person, a team of approximately ten people is currently engaged in developing various aspects of Lolita. The Lolita system is written entirely in Miranda[1] [4]. Miranda is a pure, functional programming language with non-strict semantics (i.e. lazy evaluation) and a polymorphic typechecking system. The features and syntax of the Miranda language will not be discussed in detail; instead, the reader is referred to [5], an introduction to functional programming in a language very similar to Miranda. In addition, Miranda possesses a mechanism for defining abstract data types (Section 2.3), a feature used extensively in the Lolita system, and a module system, allowing separate compilation and parameterized modules.

The Lolita system source code is continually be-

*This research is supported by grants from the Science and Engineering Research Council of Great Britain.

[1] Miranda is a trademark of Research Software Ltd.

ing changed. Many of the people involved in writing new pieces of code and changing existing code are new to the system and some have had no previous experience of computer programming whatsoever. In spite of these apparent obstacles, alterations to the system have been accomplished with surprising ease and very little disruption to other parts of the system and other Lolita developers' work. Novice functional programmers have relatively effortlessly incorporated their work into the system. People who previously had little or no idea of how the Lolita system worked have been able to commence with their modifications in a matter of days. Of course, the situation is not entirely perfect, but given such an unstructured, decentralised development model, maintenance of the Lolita system has been able to proceed remarkably smoothly.

Why has this been? A certain amount of this could be attributed to good design, but only up to a point; the Lolita system has not been built according to any formal design methodology and no specification exists, apart from the source code itself. The aim of this paper is to demonstrate, at least in part, that because Lolita has been written using a functional language, program comprehension has been greatly aided. In addition, we shall attempt to show that the maintenance task has in no small measure been eased by the same token. In the next section, we shall describe the various types of maintenance and which of these are currently being applied to the Lolita system.

1.2 Maintenance

Historically, the term 'maintenance' has been applied to the process of modifying a program after it has been delivered and is in use. These modifications may involve simple changes to correct coding errors, more extensive changes to correct design errors or drastic rewrites to correct specification errors or accommodate new requirements.

As Turski pointed out in [6], this is a great abuse of the term 'maintenance'. The addition of a new wing to a building would never be described as maintaining that building, yet adding new facilities to a program is described as a maintenance activity. However, as the term maintenance is in wide and general usage, it will be used here to mean changing the program in order to correct errors or, more often, to provide new facilities.

It is impossible to produce systems of any size which do not need to be maintained. Over the lifetime of a system, its original requirements will be modified to reflect changing needs. The system's environment will thus change and errors may emerge. Because maintenance is unavoidable, systems should be designed and implemented so that maintenance problems can be minimized.

Software maintenance can be divided into three categories:

- Perfective maintenance
- Adaptive maintenance
- Corrective maintenance

Perfective maintenance involves those changes demanded by the user or the system programmer which improve the system in some way without changing its functionality. Adaptive maintenance is due to changes in the environment of the program and corrective maintenance is the correction of undiscovered system errors. A survey by Lientz and Swanson [7] discovered that about 65% of maintenance was perfective, 18% adaptive and 17% corrective. Furthermore, they found that large organizations devoted about 50% of their total programming efforts to maintaining existing systems.

Particularly important are the principles of information hiding. It is a characteristic of any change that the original program structure is corrupted. The greater the extent of the corruption, the less understandable the program becomes and the more difficult it is to change; information hiding can alleviate this situation to some extent—such an issue becomes key in the development of research-based systems. The program modifier should try, as far as possible, to minimise the effect on program structure by using information hiding and concealing the low-level details of the implementation within carefully designed abstractions.

Maintenance costs can be governed by a collection of technical factors, some of which have been identified as:

1. Module independence. It should be possible to modify one program unit of a system without affecting any other unit.
2. Programming language. Programs written in high-level programming languages are usually easier to understand and hence maintain than programs written in a low level language.
3. Programming style. The way in which a program is written contributes to its understandability and hence the ease at which it can be modified.

When planning the perfective and adaptive maintenance of a large system such as Lolita, the emphasis on maintenance management principles provides worthwhile effectiveness and efficiency considerations.

In the remainder of this paper we shall examine some of the factors affecting comprehension of the Lolita system. We shall then demonstrate the effectiveness of these factors through a series of case studies selected from instances of maintenance of the Lolita system.

2 Factors aiding comprehension

In this section, some of the features of functional programming in general (and Miranda in particular) are discussed and their relevance to program comprehension are explained.

2.1 Referential transparency

Pure functional programming languages possess the property of *referential transparency*, a property also possessed by mathematical notation. This means that no side-effects are allowed in the language and thus rules out assignment statements and `goto`s. This also means that the value of an expression depends solely on the values of its subexpressions and there are no hidden effects influencing its value. Thus subexpressions may be substituted directly with another expression having the same value; also, different occurrences of the same name always have the same value, unlike in imperative languages, where a variable may be assigned several different values within an expression.

The property of referential transparency makes functional programs easier to understand; there is no extraneous information required in the program relating to memory allocation or storage of values.

2.2 Function application and currying

A factor which improves readability is the syntax of function application in Miranda. The operation of function application is represented by simple juxtaposition of the function and its arguments. Thus a function `f` applied to two arguments `x` and `y`, represented in most imperative languages as `f(x,y)` is represented in Miranda as `f x y`. This enables a program to use far fewer brackets. Associated with this is a device known as *currying*. Currying involves the replacement of structured arguments with a list of simple ones. We shall take the example of the function `plus`. This function gives the sum of two numbers. Consider the two definitions:

```
plus' (x,y) = x + y
```

and

```
plus x y = x + y
```

In an ordinary imperative language, the definition `plus'` would be used. However, Miranda also allows the definition `plus` to be written. The difference is that `plus'` takes the single, structured argument of a tuple of two numbers; the function `plus` takes two simple arguments. One can therefore write `plus 1 2` which is equivalent to the expression `1 + 2`. Function application in Miranda is left associative; `plus 1 2` is therefore interpreted as `((plus 1) 2)`. Thus the expression `(plus 1)` is a function in its own right—it takes a single argument and adds 1 to it. Without currying, the function to add 1 to a number would have to be written as a separate, new function. This simple but useful feature allows functions to be greatly simplified merely by leaving out arguments when they are not necessarily required, thus aiding readability. Currying therefore allows parameter hiding in abstract types. The abstract type mechanism is employed widely in the Lolita system and will be explained in the following section.

2.3 Abstract types

When using the mechanism of type definitions to influence a new type, we are in effect naming its values. With the exception of functions, each value of a type is described by a unique expression in terms of constructors. Using definitions by pattern matching as a basis, these expressions can be generated, modified and inspected in various ways. It follows that there is no need to name the operators associated with the type. Types in which the values are prescribed but the operations are not are called *concrete* types.

Abstract types operate in the reverse—an abstract type is defined not by naming its values, but by naming its operations. How values are represented is therefore less important than which operations are provided for manipulating them. The meaning of each operation has to be described by either algebraic specification, stating the relationship between the operations as a set of algebraic laws or by models, describing each operation in terms of the most abstract representation possible.

In order to implement an abstract type, the programmer must provide a representation of its values, define the operations of the type in terms of this representation and show that its implemented operations satisfy the prescribed relationships. Apart from these obligations, the programmer is free to choose between different representations on the grounds of efficiency or simplicity.

Important to the design of large programs such as the Lolita system is the concept of *abstraction barriers*, the mechanism of hiding the implementation of an abstract type so that the reference to the concrete representation is not permitted elsewhere in the program. In particular, this approach allows the representation to be changed without affecting the validity of the rest of the program. Programming of the system can in effect take place entirely at one of the predefined abstract levels and the maintenance of individual modules structured in terms of the abstract operations and types.

Abstract data types thus permit the specification of a data type together with operations which can be performed upon it. This, in effect, allows one to design a domain-specific language to tackle the given problem. This is demonstrated in the case study presented in Section 3.1. In an imperative language, a domain-specific language would normally be implemented with a separate program to parse this language and transform it into code which could then be handled with the imperative language compiler (the UNIX tool yacc [8] is an example). The advantage of this approach is that the programmer has precise control over the syntax of the new language. The disadvantages are that the tool is difficult to implement in the first place and is subsequently inflexible. It cannot be changed easily to accommodate new features or alter old ones and the syntax of the domain-specific language is completely different for each tool. These are problems which are addressed by the abstract type mechanism of Miranda; an example of how a domain-specific language may be defined in Miranda is given in Section 3.1.

Maintenance may therefore take place at three different levels. A change may be performed at the level of the abstract domain, requiring no change to the implementation. Alternatively, the domain itself may be altered, for example by adding new operations, thus changing the semantics of the domain-specific language. Lastly, the implementation of the abstract type may be altered without affecting the syntax of the domain-specific language. Maintenance at the first level is fairly easy to accomplish in imperative languages when using a tool such as yacc; using yacc, the other two are either difficult or impossible.

When writing abstract data types in Miranda, we have found two techniques to be particularly useful. The first of these makes use of currying to allow parameter hiding when using the abstract type. The hidden parameters are only ever referred to in the implementation of the signature functions and are omitted when using the abstractions, making the program much clearer and easier to read. The second technique involves the use of continuation functions to implement abstractions in which data is handed on from one part of the abstraction to the next.

The abstract type mechanism of Miranda works simply by breaking the type equivalence between the abstract type name and its representation. The only functions which may make use of the equivalence are those named in the type signature for the abstract type. An attempt by any other function to refer to the abstract type in terms of its representation will be reported by the typechecker as an error and the program will not compile. This means that there is no price to be paid, in terms of loss of efficiency, for using abstract data types in Miranda as they are enforced by the typechecking system and do not affect the implementation of the run-time system. Several aspects of the type system of Miranda relevant to the comprehension of large programs will now be discussed.

2.4 Type issues in Miranda

Miranda is a strongly typed language. Associated with each function is a type specification which gives the types of the parameters of the function and the type of the value that the function returns. Thus the type specification of the function `plus` is

```
plus :: num -> num -> num
```

which indicates that `plus` takes two numbers and returns a number. The type of the function `(plus 1)` is

```
(plus 1) :: num -> num
```

as it takes a single number and returns a number. A type specification in a functional language is more informative than in an imperative language as the former allows no side-effects. In a functional language, we know that the function takes some value or values and returns a value which depends on some operation which has been performed on these values. We know that no external factors can have affected the return value of the function; this is not the case in an imperative language, where the function could easily have changed a global value or relied on a global variable to influence the value returned. Moreover, in imperative languages, typechecking ensures that types are consistent within statements. However, imperative languages rely heavily on the sequence of statements being correct; typechecking cannot detect errors at this level. In functional languages, there is only one level to check— that of function application, which is checked by the typechecker. This not only means that the typechecking system in functional languages detects a greater proportion of errors than in imperative languages, but also that a type specification in a functional language is more informative about what a function will do than in an imperative language. For instance, a function which takes a list and returns a list can only perform a limited number of transformations on that list; with an imperative language, a function with an identical specification has a much greater range of possibilities, not all of which will involve simply transforming the list.

The type of a function in a functional language can therefore be used as a general guide to what the function will do. Thus if a function is required to check if a particular value exists in a list of numbers, it is sensible to search for the existence of a function which takes a list of numbers and a number and returns a boolean truth value. Although no such tool exists at present for Miranda, this idea has been explored by several researchers in the field [9, 10].

2.5 Higher-order functions

A higher-order function is one which can take another function as an argument. This allows functions such as `map` to be written. `map` takes a function and a list as arguments and applies that function to each element of the list. Thus instead of defining the recursive function `squares` to calculate the squares of a list of numbers (Figure 1), one may simply write `map square`. This is much easier to read than the recursive definition and, provided one knows what `map` does, easier to understand. This type of function is very common in Miranda and novice programmers often resort to writing functions employing such ad-hoc recursion; with a little more experience, however, the programmer learns to recognize this pattern of recursion and will use definitions employing `map` in preference. Figure 2 gives a definition of `map`—note the similarity to `squares` and how it has been generalized from `squares` by the addition of the function parameter `f`.

The ability to define higher-order functions in Miranda has allowed the parsers of Section 3.1 to be implemented as functions using the parser abstractions.

```
squares [] = []
squares xs = square (hd xs) : squares (tl xs)
```

Figure 1: A recursive definition of the function `squares`.

```
map f [] = []
map f xs = f (hd xs) : map f (tl xs)
```

Figure 2: The function `map`.

2.6 The Miranda interpreter

With Miranda, there is an additional factor which aids program comprehension. A Miranda script is a collection of functions, all of which are in scope when that script is loaded into the interpreter. This allows individual function definitions to be tested separately from each other and means that if one wishes to discover what a particular function does, one can simply call it directly from the interpreter.

In the following section, we shall relate some of the issues above to real-life case studies of maintenance tasks in the Lolita system.

3 Case studies

3.1 The syntactic parser

The parser was originally written to enable the parsing of the English language. The parser abstractions have been written in such a way as to enable a grammar to be written in Miranda simply by transcribing the BNF (Backus Naur form) definition of the grammar with very little alteration. The traditional way of doing this in an imperative language would be to use a parser generator (such as yacc) to interpret the contents of the grammar definition file and thus produce a parser.

Using abstractions in Miranda, the grammar itself can be written in Miranda with no need for a separate program to translate the grammar. As mentioned in Section 2.3, the abstract data type mechanism in Miranda allows one to devise a domain-specific language for a particular purpose; the parser is a prime example of this. In addition, this mechanism allows the grammar to be specified as a function rather than an algebraic data structure, which means that there is no

```
number ::= int ?fracpart ?exppart
int ::= '-' nat |
            nat
fracpart ::= '.' nat
exppart ::= 'e' int
nat ::= digit nat |
        digit
digit ::= '0' | '1' | ... | '9'
```

Figure 3: A BNF-like definition of a grammar for numbers.

```
number = int $t_ alt_ fracpart $t_ alt_ exppart
int = alt_ minus $t_ nat $o_
      nat
fracpart = point $t_ nat
exppart = e $t_ int
nat = digit $t_ nat $o_ digit
```

Figure 4: The numbers grammar in Miranda.

need for the grammar to be interpreted separately as it can simply be applied directly to the input.

We shall illustrate the simplicity and elegance of the parser abstractions with a very simple example. Suppose it was required to write a grammar which would accept a sequence of symbols representing a natural number in a computer programming language. The grammar can be described in a BNF-like form, shown in Figure 3. Our abstractions enable this to be translated into the Miranda code shown in Figure 4.

As can be seen, the BNF symbol `::=` translates to `=`, `?` translates to `alt_`, `|` translates to `$o_` and simple juxtaposition of symbols translates to `$t_`. The signature of the parser abstract type is shown in Figure 5. The operators `t_` and `o_` take two parsers and combine them to form a single parser. The function `alt_` takes a parser and returns a parser and the function `parse` takes a parser and an input stream and returns the result of applying that parser to the input stream. There is thus another interesting point to note about the grammar above: each of the grammar rules is itself a parser.

The parser abstractions are themselves implemented by the "list of successes" method, similar to that advocated by Wadler in [11]. This makes use of lazy evaluation to simulate a backtracking parser, which generates a list of all the possible results of the parse. If the parser fails to match all of its input, a list of incomplete parses is given.

Writing parsers in the way described above leads

```
abstype parser
with
    t_, o_ :: parser -> parser -> parser
    alt_ :: parser -> parser
    parse :: parser -> [char] -> parse_result
```

Figure 5: Parser abstract type signature.

to code which is declarative; the code tells us what the parser does rather than how it does it. There are several factors which enable one to write parsers in such a fashion in Miranda. The use of higher-order functions obviates the need for an intermediate data structure and, together with currying, allows for the parameters to the parser functions to be hidden. User-defined operators allow the use of a BNF-like notation. Lazy evaluation removes the need for a complicated backtracking mechanism. Abstract data types hide details of the implementation. Most of these features are lacking in imperative languages.

If we consider the implementation of a parser in an imperative language using a similar method, it is soon found that the lack of these features is a serious disadvantage. Since in most imperative languages we cannot define our own operators, the BNF-like operators of our Miranda parser become functions in the imperative language, with the attendant cumbersome notation of nested brackets. Since imperative languages do not permit higher-order functions, the parser functions would have to build a data structure to represent the grammar. This data structure would be in the form of a graph, which would need to be interpreted by a separate function which would perform the parsing, unlike in Miranda, where the functions which build the grammar also do the parsing. Additionally, imperative languages require storage control, so the code would also need to contain statements to accomplish this, unlike in a functional language.

We shall now examine a real-life application of the existing parser abstractions to a new problem.

3.1.1 A mixed Chinese/English grammar

Research is currently under way at Durham by Wang into the use of an intelligent tutoring system to eliminate errors of transfer by students studying the Chinese language [12]. This has involved writing a parser for Chinese in Miranda and integrating it with the existing parser for the English language grammar in such a way as to form a mixed grammar. This mixed grammar commences by applying the Chinese grammar rules to the sentence and then switches to using the English rules when the Chinese rules fail to match. Wang, who has no prior experience of functional programming, or indeed computer programming of any form, has successfully implemented a parser for this grammar.

Wang started by writing a grammar for Chinese. This grammar was written using Wang's own notation. The grammar was written entirely from abstract ideas, without any concern for how it could be translated into Miranda syntax. When this grammar had been written, it was an extremely simple matter to translate it into Miranda employing the abstractions which already existed. A section of the grammar in Miranda is shown in Figure 6. The mixed grammar required two new functions to be added to the abstractions to cope with switching from the Chinese grammar to the English grammar when no successful parse is found using the Chinese grammar. These functions, `oet` and `ioet` can be seen in Figure 6. Since the two functions were new, no alteration was required to the existing English language parser.

Thus can be seen the direct correspondence between the actual grammar itself and its implementation using the parser abstract type and its associated functions in Miranda.

3.2 The semantic parser

The semantic parser is a central feature of the Lolita system. The input to the semantic parser is a syntactic parse tree built at the previous level in the system. The output from the semantic parser is the corresponding semantic net structure. The fundamental task therefore is the transformation from the parse tree structure to the semantic net data type.

Each node in the parse tree is labelled with its grammatical construct. For instance the root node of the parse tree is labelled with *sen*, representing the complete sentence structure. Each of these labels has a corresponding semantic parse rule which will perform the transformation from the parse tree node to the abstract semantic structure. The semantics of a node in the parse tree is primarily determined by its label and the subtrees trees below it. However, contextual information is also required to maintain a list of referents[2] and the semantic net must be updated with the new nodes produced. This is achieved using an additional channel of information which traverses the parse tree from left to right in a depth-first fashion. This traversal is illustrated by dotted lines in Figure 7.

[2]Nodes which may be referred to in later pieces of text by pronouns (e.g. 'he' or 'it')

The main abstract data types of the semantic parser are *leaf rules* and *branch rules*. A leaf rule builds the semantics for a leaf in the parse tree and updates the context. A branch rule combines the semantics from the two branches at a node to give the semantics for that node. It also updates the context with new nodes added to the semantic net and any referents added to the list of referents.

In the semantic parse, a set of these abstract rules are passed as parameters. These rules are then applied to the appropriate parts of the tree to produce the final semantics. Each of these rules adds more detail to the structure of the semantic net. Using the abstract types, the details of the implementation of these operations have been hidden. Thus in the code for the semantic parser, only the semantic rules are given.

Figure 7 shows the results of the semantic parse for the sentence "Roberto owns a motorbike." The semantic parse begins by applying a leaf rule to the first leaf node ('ROBERTO'). The information is then passed back to the parent node. The application of the `proper noun` branch rule at the level above will create the first part of the semantic representation.

As an example, consider the following rule applied to the `transvp` node in the parse tree of Figure 7:

```
meta_branch na
  = labelboth Act Obj,
           if na $is_in ["transvp ","is_a ",
                "eq_is_a ", "hypsentverbs "]
```

This rule checks whether the name of the node `na` is one of either `transvp`, `is_a`, `eq_is_a` or `hypsentverbs`. Since this is the case, the rule then labels both the left and right sub-branches of the node `transvp`. The left branch, representing 'OWN', is labelled as as an action (`Act`) and the right branch, representing 'A MOTORBIKE', is labelled as an object (`Obj`). The identification of these branches as action and object values allows the subsequent construction of the corresponding part of the semantic net.

Using the rule abstractions, it is possible to create larger *meta-rules* by combining several smaller rules. The operator `$compose` takes two rules and applies the first rule to the result obtained from the application of the second. The following example creates a meta-rule for proper nouns by combining three smaller rules: `labelall`, `unique_newnodes` and `addrefs`.

```
meta_branch "full_propernoun "
  = addrefs
    $compose unique_newnodes mkobject
    $compose labelall Univ
```

```
location_sen = ((poss_ph $o_ prop_ph $o_ proper_name $o_ pron_per)
                $t_ (aux_loc_vp $o_ location_vp))
               $ioet_ quvb_sen
aux_loc_vp  = aux $t_ location_vp
location_vp = (location_ph $t_ (transvp $o_ act1 $o_ act_in))
               $oet_ (transvp $o_ act1 $o_ act_in) $t_ location_ph
```

Figure 6: Grammar after translation into Miranda

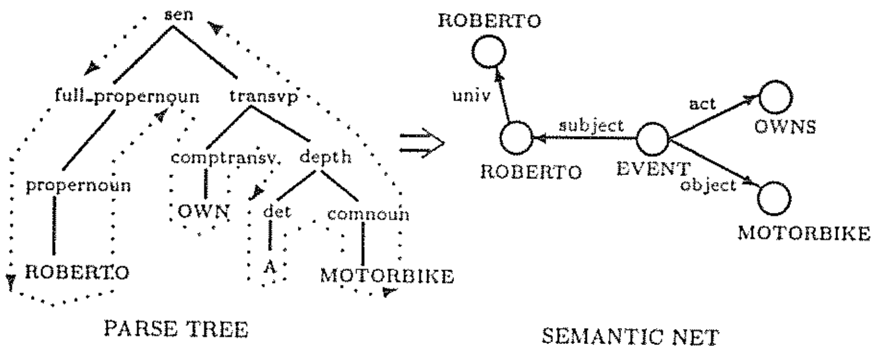


Figure 7: A fragment of the semantic net.

The initial application of `labelall` to the parse tree of Figure 7 creates a link to the universal `Univ` node which represents the set of all 'ROBERTO's. This node has been passed up from the generation of the semantics of `propernoun`. The rule `unique_newnodes` is then applied to this to give a node representing a unique 'ROBERTO', an instantiation of the universal 'ROBERTO'. This new node is thus the semantics of `full_propernoun`. The `addrefs` rule is then applied to add the resulting node to the list of referents.

The top-level semantic parse function has the following type specification:

```
sem_parse :: (nodelabel -> branchrule) ->
             (leaflabel -> leafrule) ->
              tree -> semantic_net ->
              semantic_net
```

Thus the function `sem_parse` takes four parameters: a function which takes a node label and returns the appropriate branch rule to apply at that node; a function which takes a leaf label and returns the appropriate leaf rule to apply to that leaf; the parse tree and the initial semantic net. It returns the transformed semantic net.

Multiple semantic nets

An interesting point concerns the way in which the semantic net is represented in Lolita. It can be seen from this method that it is necessary to traverse the parse tree whilst continually updating the corresponding semantic data type. The operation of the semantic net was originally visualised as an abstract state machine and would have been implemented as such in an imperative language with one copy of the net being accessed and altered by a set of functions. However, this implementation is not possible in a functional language owing to the lack of side-effects or state, features that an abstract state machine implementation would rely upon. In Miranda therefore, the state is passed explicitly—the semantic net is a large data structure which is passed as a parameter from one function to the next. Because of the way that functional languages are implemented, there is minimal efficiency overhead imposed in doing this. Additionally, it allows there to be multiple different copies of the semantic net at any given time with very little space overhead; the only additional memory used is in storing the alterations to the net as the duplicated parts are referenced by pointers in the implementation, which represents the program as a graph [13]. This would be difficult given the implementation of the semantic net as an abstract state machine in the imperative language; major alterations would have been necessary to accommodate extra copies of the net. Indeed, it was not initially envisaged that multiple copies of the net would be required in Lolita; however, this was required in the semantic analysis stage. The semantic analysis stage involves adding information to the semantic net and altering existing information. There are often a number of alternative analyses that must be explored and hence a difference semantic net is required for each. This additional requirement needed no alteration to the implementation of the semantic net whatsoever. So easy is this feature to incorporate, it is not even necessary to know whereabouts in the program multiple copies of the semantic net are being used; it is simply a facility which is taken for granted.

4 Conclusion

In this paper, we have shown how the use of Miranda, a lazy, functional programming language, has considerably helped in the task of maintaining a large, complex, real-life system. The ability to define domain-specific languages for particular parts of the system has been instrumental in aiding programmer comprehension of the Lolita system. The abstract data type mechanism of Miranda is central to this; however it is this mechanism used in conjunction with several other features which gives functional programming an advantage over imperative languages. These other features include higher-order functions, lazy evaluation, implicit storage management and user-defined operators.

These factors have allowed new features to be incorporated into the Lolita system with minimum effect on other parts of the program. They have also enabled programmers who had not previously worked on the system to familiarize themselves rapidly with the code. People who have had no experience of programming whatsoever have been able to write parts of the system with very little training. It is the use of Miranda for programming the Lolita system which has allowed much of this; however, many of the concepts discussed in this paper would be equally applicable to other lazy, functional languages.

Acknowledgements

The authors wish to thank Mark Smith for providing information on the semantic net and Yang Wang for information on the Chinese/English mixed grammar parser.

References

[1] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, no. 2, 1989.

[2] R. Garigliano, R. Morgan, and M. Smith, "LOLITA: Progress report 1," Tech. Rep. 12/92, Artificial Intelligence Systems Research Group, School of Engineering and Computer Science, University of Durham, Science Laboratories, South Road, Durham DH1 3LE, United Kingdom, 1992.

[3] D. Turner, "Recursion equations as a programming language," in *Functional Programming and its Applications* (Darlington, ed.), pp. 1—28, Cambridge University Press, 1982.

[4] D. Turner, "Miranda: a non-strict functional language with polymorphic types," in *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture* (J.-P. Jouannaud, ed.), no. 201 in Lecture Notes in Computer Science, (Nancy, France), pp. 1—16, Springer-Verlag, 1985.

[5] R. Bird and P. Wadler, *Introduction to Functional Programming.* Series in Computer Science, Prentice Hall International, 1988.

[6] W. Turski, "Software stability," in *6th ACM European Conference on Systems Architecture*, (London, UK), 1981.

[7] B. Lientz and E. Swanson, *Software Maintenance Management.* Addison-Wesley, 1980.

[8] S. Johnson, "Yacc—Yet Another Compiler Compiler," Tech. Rep. 32, Bell Labs, Murray Hill, New Jersey, USA, 1975. Also in UNIX Programmers' Guide.

[9] C. Runciman and I. Toyn, "Retrieving re-usable software components by polymorphic type," *Journal of Functional Programming*, vol. 1, pp. 191—211, Apr. 1991.

[10] M. Rittri, "Using types as search keys in function libraries," *Journal of Functional Programming*, vol. 1, pp. 71—89, Jan. 1991.

[11] P. Wadler, "How to replace failure by a list of successes," in *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture* (J.-P. Jouannaud, ed.), no. 201 in Lecture Notes in Computer Science, (Nancy, France), pp. 113—128, Springer-Verlag, 1985.

[12] Y. Wang and R. Garigliano, "An intelligent tutoring system for handling errors caused by transfer," in *Intelligent Tutoring Systems: Second International Conference*, no. 608 in Lecture Notes in Computer Science, (Montreal, Canada), Springer-Verlag, 1992.

[13] S. Peyton Jones, *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

Session B: Experiments in Program Comprehension

Chair: Panos E. Livadas

Experiments in Identifying Reusable Abstract Data Types in Program Code

G. Canfora*, A. Cimitile** M. Munro*** and M. Tortorella**

* DIIMA — Dep. of "Ingegneria dell'Informazione e Matematica Applicata"
University of Salerno, 84084 Fisciano (SA), Italy.

** DIS — Dep. of "Informatica e Sistemistica"
University of Naples, Via Claudio 21, 80125, Naples Italy

*** CSM — Centre for Software Maintenance
University of Durham, Durham DH1 3LE, UK

Abstract

In this paper the issue of program comprehension is addressed from the software reuse perspective. In particular the identification of abstract data types in existing program code is explored. A candidature criterion is presented and a prototype implementing it is described. The criterion is applied in an experiment that analyses five very different programs and the resulting output is discussed.

The work described forms part of the RE^2 *project that addresses the wider issues of software reuse through the exploration of reverse engineering and re-engineering techniques to identify and extract reusable assets from existing systems.*

1 Introduction

The comprehension of existing software systems plays a central role in many larger software engineering activities such as testing and debugging, validation, migration, maintenance and enhancement, re-engineering, and reuse. In this paper the issue of comprehending software is dealt with by the software reuse perspective.

An immediate problem to solve, in order to spread software reuse, is the finding of the reusable assets [1-3]. Reverse engineering and more general software comprehension techniques can help to solve this problem by facilitating the extraction of the reusable assets from existing systems. Extracting reusable assets involves accessing the existing systems, to identify reuse candidate components, and understanding their meaning in order to specify and catalogue them. Reverse engineering techniques can provide answers to some key questions such as:

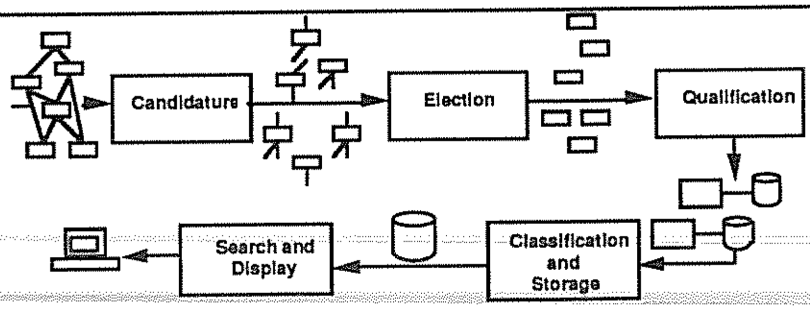
- what are the criteria for identifying the reuse candidate components ?
- what are the methods and formalisms to specify them ?
- what are the techniques to catalogue them in order to make their retrieval simple ?

Extracting reusable assets from an existing system also requires re-engineering techniques to decouple the reuse candidate components from the external environment and to package them into *easy-to-reuse* artefacts. In this paper we call reuse re-engineering processes the set of activities to populate a repository with reusable software components extracted and re-engineered from existing systems.

Reuse re-engineering processes are the main concern of the RE^2 project, a research project funded by the CNR (Italian National Research Council) and jointly carried out by the DIS (Dep. of Informatica e Sistemistica) of the University of Naples and the CSM (Centre for Software Maintenance) of the University of Durham. The aim of this two years project is to explore and single out the role of reverse engineering and re-engineering in the setting up of the reuse re-engineering processes. The RE^2 project has established a reference paradigm to implement reuse re-engineering processes. The key role of the paradigm is to allow the repetition of experiments, thus facilitating the learn process. It also defines a framework in which the available methodologies and tools can be used. Table I gives an overview of the RE^2 reference paradigm; a more detailed description is given by Canfora *et al.* [4].

The reuse re-engineering processes may be defined and set up to extract different types of reusable assets such as design documents, specifications, code, test data, business models. Extracting all these types of components allows software reuse to be placed into different stages of the processes of developing a new system, thus maximising its

This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the CNR under grant no. 91.00930PF69.



The reuse re-engineering paradigm RE^2 divides a reuse re-engineering process into five sequential phases each of which includes a set of homogeneous activities and is fully identified by the objects it produces. These five phases are:

- CANDIDATURE: This phase groups together the activities of source code analysis and produces sets of software components. Each one of these sets is a candidate to make up a reusable module when suitably de-coupled, re-engineered and possibly generalised.
- ELECTION: This phase groups together the activities of the analysis of the collection of software-component sets singled out in the candidature phase and produces a set of reusable modules.
- QUALIFICATION: This phase groups together the activities that produce the specifications of each one of the reusable modules obtained in the election phase. Both the functional and the interface specifications must be produced in this phase.
- CLASSIFICATION AND STORAGE: This phase groups together the activities that classify the reusable modules and related specifications according to a reference taxonomy. The aim is to define a repository system and populate it with the reusable modules produced.
- SEARCH AND DISPLAY: This phase groups together the activities that set up a front end user interface to interact with the repository system. The aim is to make finding the modules the user needs as simple as possible, for example by giving them visual supports to navigate through the repository system.

Tab. I: The RE^2 reference paradigm.

potential pay off. Currently, the RE^2 project deals with reusable components consisting of source code although the reuse of other artefacts is being considered. In this paper, therefore, the term reuse is used to mean "reuse of existing code" and the reuse re-engineering processes dealt with aim to produce reusable source code modules and related specifications.

Accessing the existing systems to identify reuse candidate modules is the major task of the CANDIDATURE phase. This phase involves three types of activities: (i) defining a candidature criterion, and the model to apply it (ii) defining and setting up a reverse engineering process to create an instance of the model (iii) applying the candidature criterion. The role of candidature criteria is to automatically produce a first approximation to the sets of components to be extracted from the system, each set being a candidate to create a reusable module. Examples of components that can be extracted include procedures, functions, subsystems, slices, primes, data structures, and user-defined data types. The definition of a candidature criterion entails the definition of the model of the system needed to apply it, and of the information to be reverse engineered to make up the model instances. The RE^2 project assumes the principle of abstraction [5] as a guideline to define the candidature criteria. For a set of components to be a candidate it must implement one and only one abstraction, i.e. one functionality, one object, one abstract data type, one interface, one abstract state machine. Changing either the type or the level of the abstraction to be clustered in a reusable module entails the change of the candidature criterion. For example, the criteria and models to look for functional abstractions (that refer to algorithms and can be specified by input/output relationships) are necessarily different from the ones to look for data abstractions (that refer to objects or classes) or control abstractions (that refers to the synchronisation of concurrent processes and the disciplining of accesses to shared resources). Canfora *et al.* [4] present a formalised approach to some different types of abstractions and discuss the reverse engineering techniques and tools needed to extract them from existing systems.

In this paper we focus on the family of data abstractions and in particular on the extraction of abstract data types. A candidature criterion, and the model to apply it, are illustrated and results from case studies are presented. The criterion is founded on logic and Prolog is used for the rapid prototyping of the case studies. The aim of these case studies is to validate the proposed candidature criterion and to assess its strength while understanding its limitations. They also aim to show the practical feasibility of a software tool for finding abstract data types.

The remainder of the paper is organised as follows. Section 2 defines the criterion to look for abstract data types and the model to apply it. The definition of the candidature criterion depends on the way in which abstract data types are implemented. This is why section 2 briefly recalls the concept of abstract data types and discusses how they can be implemented in traditional languages that do not present any *ad hoc* construct. Section 3 illustrates the rapid prototyping of the criterion using Prolog, and outlines the main characteristics and features of a software tool for finding abstract data types. Results from several case studies are presented and discussed in section 4, while section 5 contains some concluding remarks and considerations.

2 A candidature criterion for identifying abstract data types

Abstract data types allow a type to be axiomatically defined in terms of the operations that can be performed on the variables of that type [5, 6]. In practice, for the conventional programming languages, an abstract data type consists of a collection of user-defined data types and procedure-like components. The user-defined data types define the supporting data structure while the procedure-like components implement the operators.

A module implementing an abstract data type exports constants, user-defined data types and procedure-like components, while its implementation part does not contain any encapsulated variable. This defines the type of components' collections which a CANDIDATURE criterion looking for abstract data types must single out.

We assume that the procedure-like components use the user-defined data types in their headings, i.e. to declare formal parameters and/or return values. A module implementing an abstract data type must allow a designer to declare several different objects and to access and manipulate them by calling the procedure-like components. This requires the objects to be passed as formal parameters.

The CANDIDATURE criterion we propose is founded on logic. A set of direct relations — produced from code by static analysis — describes the relationships existing between the user-defined data types and the procedure-like components that use them in their headings. This set of relations defines the system's model needed to apply the criterion that, in turn, consists of a set of summary relations obtained by combining direct relations in expressions.

The set of direct relations we use describes a graph, the type-procedure-connection graph, which belongs to the family of the interconnection graphs proposed by Calliss [7]. The type-procedure-connection graph is a combination of the type-connection graph, used by Calliss and Cornelius [8] to detect and factor pot-pourri modules, and a slightly modified version of the reference graph proposed by Embley and Woolfield [9] to assess the quality of abstract data types implemented in Ada. If TT is the set of the user-defined data types in a software system and CC is the set of the procedure-like components, a type-procedure-connection graph is a directed graph G(N,E) with nodes $N \equiv TT \cup CC$ and edges $E \equiv \{(c_i,t_j) \mid c_i \in CC \wedge t_j \in TT \wedge$ 'c_i uses t_j in its heading'$\} \cup \{(t_i,t_j) \mid t_i, t_j \in TT \wedge$ 't_i is used to define t_j'$\}$. A path in G is a sequence of nodes $n_1, n_2,\ldots,n_k$ such that every couple (n_i,n_{i+1}) belongs to E. Let $\mu(n_i,n_j)$ denote a path connecting the nodes n_i and n_j, and let $\wp(G)$ denote the set of all the paths in G. It is worthwhile to point out that a node $n \in TT$ can only be the first element of a path, i.e. paths can only connect procedure-like components to user-defined data types, and user-defined types to user-defined types. In accordance with Liu and Wilde [10], we say that the user-defined data type t_i is a sub-type of t_j, denoted by $t_i \ll t_j$, if a path exists that connects t_i to t_j. We also say that t_j is a super-type of t_i. Obviously, if $t_i \ll t_j$ and $t_j \ll t_k$ then $t_i \ll t_k$.

A type-procedure-connection graph is fully represented by the two following direct relations:

- USED-TO-DEFINE $\subseteq TT \times TT$ defined as:
  $(t_i,t_j) \in$ USED-TO-DEFINE iff the user-defined data type t_i is used to define t_j.
- TYP $\subseteq CC \times TT$ defined as:
  $(c_i,t_j) \in$ TYP iff the procedure-like component c_i uses the user-defined data type t_j to define a formal parameter and/or a return value.

The candidature criterion to look for abstract data types on a type-procedure-connection graph G(N,E) is defined by the following four steps algorithm:

Step1.
From G(N,E) generate the graph G'(N',E') such that:
$N' \equiv N$ and
$$E' \equiv E - \{(c_i,t_j) \mid c_i \in CC \wedge t_j \in TT \wedge \exists\, t_k \in TT \bullet ((c_i,t_k) \in E \wedge \exists\, \mu(t_j,t_k) \in \wp(G))\}$$

Step2.
From G'(N',E') generate the graph G"(N",E") such that:
$N'' \equiv N'$ and $E'' \equiv E' - \{(t_i,t_j) \mid t_i, t_j \in TT\}$

Step3.
For each one of the isolated sub-graphs[1] in G" create a candidate abstract data type

Step4.
Recognise and establish any necessary interconnection (i.e. use and/or composed-of relationships) between the candidate abstract data types.

The above candidature criterion can be re-formulated in a logic based fashion through the following relations:

- ABTYP = (trans(STYP)STYP)*
- CCTYP = (trans(STYP)STYP)* trans(STYP)

where trans(R) and R* denote the transpose and the reflexive transitive closure of the relation R, and STYP is a binary relation on $CC \times TT$:

- STYP $\subseteq CC \times TT$ defined as:
  $(c,t) \in$ STYP iff the procedure-like component c uses the user-defined data type t to define a formal parameter, and c does not use any super-type of t.

[1] An isolated sub-graph of a graph G is a graph g such that: $g \subset G \wedge g \neq \emptyset \wedge \exists$ not-g $\subset G \bullet$ (not-g $\neq \emptyset \wedge g \cap$ not-g $= \emptyset \wedge g \cup$ not-g $= G$), where the symbol $\emptyset$ denotes the empty graph. This definition is due to Calliss [7], who also gives a formal definition of the sub-graph ($\subset$), graph intersection ($\cap$) and graph union ($\cup$) operations.

An algorithm to derive the relation STYP from the relations TYP and USED-TO-DEFINE is presented by Canfora *et al.* [11].

The relations ABTYP and CCTYP define, respectively, the supporting structure and the operands of the candidate abstract data type. In particular, for a given used-defined data type t, the relation ABTYP defines the set of the couples (t,t_i) such that the user-defined type t_i belongs to the same isolated sub-graph which t belongs to. The relation CCTYP defines the set of couples (t,c_i) such that the procedure-like component c_i and t belong to the same isolated sub-graph. The sets $T\_SET \equiv \{t_i \mid (t,t_i) \in ABTYP\}$ and $C\_SET \equiv \{c_i \mid (t,c_i) \in CCTYP\}$ define, respectively, the user-defined data types and the procedure-like components to be exported by a module implementing the abstract data type defined around t — actually, the one identified by the isolated sub-graph that contains t.

3 Outline of a tool for searching abstract data types

The candidature criteria formulated in a system based on logic have the advantage of being easy to prototype using a logic programming language, for example Prolog.

In order to easily evaluate the case studies, whose results are presented in the next section, we have developed a prototype tool that implements the proposed candidature criterion. Our aim is not to create a product to be released for external use but to have a tool that makes it possible to experiment with the candidature criterion on non toy programs in a reasonable time. The user community of the prototype is intended to be the research group within which the RE^2 project is being carried out. Therefore, issues like time/space performance and user-friendliness have scarcely been taken into account when designing the prototype. On the other hand, a tool that fits into a research environment must be particularly flexible and easy to evolve. For the logic based candidature criteria this means, for example, that the set of the summary relations implemented must be easy to tailor and/or extend. The possibility of defining new summary relations makes it possible to tune the candidature criteria to the type and the level of the abstractions to be looked for and to the characteristics of the environment in which the reuse re-engineering process is developing. Moreover, it allows different types of candidature criteria to be defined and experimented with. This is a matter of considerable importance because the knowledge and technologies in the field of reuse re-engineering are not stable but continuously changing. Therefore, for the methods and tools proposed to be useful they must be particularly versatile and easily tailorable so as to evolve on the basis of the new knowledge developed either within the environment in which they are used, or in the research community.

The implementation of a logic based criterion requires (i) a repository to store the direct relations (ii) a language to define the summary relations, and (iii) a query facility to specify the type of abstraction to be looked for. We use a Prolog program dictionary to record the direct relations and production rules to define the summary relations. Prolog queries are used to specify the abstractions to be looked for. Direct relations consist of a collection of same named facts of arity 2, each of which represent one of the couples of the relation identified by the name of the fact. Facts of arity 1 define the software components involved in the candidature criterion and state their type. The argument is the name of a component whose type is identified by the name of the fact. It is worth-while to stress that the use of Prolog to prototype the candidature criteria meets the needs for a set of summary relations that is easy to extend. Indeed, a kernel of production rules may be established on which new candidature criteria can be defined by adding new production rules or relating the existing ones in a different way.
The structure of the program dictionary is shown below:

```
proc(procedure_name).
func(function_name).
user_def_type(type_name).
proc_use_type_in_interface(procedure_name,
                           type_name).
func_use_type_in_interface(function_name,
                           type_name).
used_to_define(type_name_1,type_name_2).
```

We have implemented a static code analyser for automatically producing the program dictionary. The analyser has been written using Lex and Yacc [12], two standard Unix facilities for the implementation of lexical analysers and parsers. The current version analyses Pascal programs written according to the ISO standard. The main disadvantage is that the programs to be analysed have to be written in a unique compilable file. An extension is being designed in order to allow programs consisting of more than one module to be analysed.

We have also written the Prolog programs that implement the candidature criterion, i.e. compute the relations STYP, ABTYP and CCTYP. As an example Tab. II shows the program that computes the relation STYP according to the algorithm described by Canfora *et al.* [11]. For the sake of brevity we do not show the productions rules implementing the relations ABTYP and CCTYP. In order to simplify the interrogation of the system, the programs have been written that answer the queries `adt_strutt(T,T_SET)` and `adt_op(T,C_SET)` by producing, for a given user-defined data type, the sets T_SET and C_SET defined in section 2.

```
%+--------------------------------------------------------+
%| styp(C,T):-                                            |
%|   the procedure like component C uses the user         |
%|   defined data type T to declare a formal parameter    |
%|   AND C does not use any super-type of T               |
%+--------------------------------------------------------+
styp(C,T):-
  procedure_like_component_use_type_in_interface(C,T),
  setof(TS1,procedure_like_component_use_type_in_interface
        (C,TS1),TS1),
  set_of_supertype(T,TS2),
  intersect(TS1,TS2,TS3),
  TS3==[].

%+---------------------------------------------------------+
%| procedure_like_component_use_type_in_interface(C,T):-  |
%|   the procedure like component C uses the user          |
%|   defined data type T to declare a formal parameter     |
%+---------------------------------------------------------+
procedure_like_component_use_type_in_interface(C,T):-
  proc_use_type_in_interface(C,T).
procedure_like_component_use_type_in_interface(C,T):-
  func_use_type_in_interface(C,T).

%+-----------------------------------------------+
%| set_of_supertype(T,TS):-                      |
%|   TS is the set of the supertypes of the user |
%|   defined data type T                         |
%+-----------------------------------------------+
set_of_supertype(T,TS):-
  setof(TS,subtype(T,TS),TS),
  !.
set_of_supertype(_,[]).

%+---------------------------------------------------------+
%| subtype(T1,T2):-                                        |
%|   user defined data type T1 is a subtype of the         |
%|   user type T2                                          |
%+---------------------------------------------------------+
subtype(T1,T2):-
  used_to_define(T1,T2).
subtype(T1,T2):-
  used_to_define(T1,Ti),
  subtype(Ti,T2).
```

TAB. II: Prolog to compute the relation STYP.

With the current state of the art we do not believe that the process of finding the abstract data types can be totally automated. Human knowledge and heuristics are needed to identify the coincidental and/or spurious connections possibly existing among the components of a candidate abstract data type, thus improving its quality before electing it for reuse. If not correctly detected these connections can lead to the production of modules which are too large and of low quality. As an example of a coincidental connection consider a system using one STACK and one QUEUE, both obtained as instances of abstract data types, and suppose that it uses a unique procedure to initialise them once at the beginning of the execution. This procedure will force the criterion to candidate a module that implements both the abstract data types STACK and QUEUE. Similarly, a procedure that reverses the content of a STACK into a LIST is an example of spurious connection as it forces the abstract data types STACK and LIST to be clustered in the same candidate module. In the first case the correct way to separate the two abstract data types consists of slicing the initialisation procedure into two different ones on the basis of the formal parameters, thus obtaining the initialisation operations for the abstract data types STACK and QUEUE. In the second case the slicing does not produce any meaningful procedure and therefore the spurious connections has to be removed simply by deleting the reversing procedure.

A tool to effectively support the process of finding abstract data types should strictly interact with the user in order to allow him to identify the procedure-like components that create coincidental and/or spurious connections. It should also make it possible to choose the most appropriate actions — slicing or deletion — to improve the quality of the candidate modules. Below is the outline of the process such a tool should support:

1) From the source code automatically produce the Prolog program dictionary, and apply the candidature criterion;
2) Present the user with the results in both a textual and a graphical form — the latter consisting of a suitable layout of the type-procedure-connection graph;
3) Let the user select the procedure-like components that create coincidental/spurious connections and indicate the action to perform on each one of them.

It should be possible for the user to backtrack their choices (i.e. to re-insert deleted nodes or re-joint divided ones) and to explore different combinations of coincidental/spurious connections and slicing/deletion actions. Finally, the tool should access the source code, extract the components defining the candidature modules, and store them for future manipulations, for example the re-engineering operations in the ELECTION phase.

An important issue is the layout algorithm adopted to display the results of the candidature criterion (point 2). It must clearly identify the candidate modules, i.e. the isolated sub-graphs. Moreover, it must show the mainly internally connected sub-graphs[2] possibly existing within an isolated sub-graph, thus facilitating the identification of the coincidental/spurious connections. We are currently investigating the possibility of obtaining the above features by integrating the prototype tool for searching abstract data

[2]A mainly internally connected sub-graph of a graph G is a graph g $\subset$ G such that the number of edges that connect couples of nodes belonging to g is higher than the number of edges linking nodes in g to nodes in G-g. If a candidate module implements more than one abstract data type these are likely to be associated with mainly internally connected components of the isolated sub-graph that depicts the module.

types into VAPS (Visual Aids for Pascal Software comprehension) [14,15], a graphical browser of Pascal programs developed on the top of the visual environment Diagram Server [16].

4 Case Studies

In this section we illustrate the results obtained by applying the candidature criterion in a set of case studies. The aim is to assess the strength of the proposed criterion while identifying and understanding its major limitations. We also exemplify the nature of the human intervention needed to improve the quality of the candidate abstract data types before electing them for reuse. The study of the nature of the human intervention is a matter of fundamental importance as it paves the way towards the definition of heuristics capable of improving the quality of the candidate abstract data types automatically singled out from the existing code.

The set of case studies consists of five Pascal programs developed in different periods, and therefore the level of good design practices exploited is quite different. The difference is also in the skill and expertise of the programmers — these include under-graduate students and university personnel. A program taken from a text book on advanced programming in Pascal [17] has also been included in order to have a "best case" to be assumed as a term of comparison for the "real life" programs. All the programs analysed have a small-medium size, between 1000 and 2000 LOC, and the overall study relies on the analysis of nearly 10,000 LOC. Below, a brief description of the programs is given:

MiniCalc.pas
A simple spreadsheet taken from the text book [17]. The program is provided with an interactive user interface. The display is divided into cells, labelled A to H vertically and 1 to 5 horizontally. When the system is in command entry mode the user enters a command indicating that he wishes to enter the identifier of a cell and an expression or label for the cell. The system responds accordingly.

ExamMarker.pas
Marks multiple choices examinations. Set in Computer Science at the University of Durham. The number of questions, their alternative answers, their answer and a particular marking scheme are input together with the students answers. The program outputs the resulting marking in order of marks, college and name.

Editor.pas
A version of the Unix editor ed as presented in the Software Tools book [20]. The program is augmented with an additional set of procedures and functions that have to be written for a particular environment, to carry out operations such as opening files and detecting interrupts from the user.

Formatter.pas
An ancient public domain pretty printer program for Pascal that has undergone numerous changes by "anonymous" authors. The output layout can be parametrised to change the indentation, the line length, and control structure format.

Othello.pas
An old Pascal program (circa 1978), origin unknown, that plays the game of Othello. The program has a simple user interface and the board is presented using a non-graphic VDU screen.

As was expected, the criterion produced the best results for the program MiniCalc. Here five candidate modules were obtained implementing the following abstract data types:

Parsed_Expressions, that manipulates an already parsed expression. In particular, the data structure defining the parse tree and the operators for evaluating and printing out the values of expressions are clustered in this module.
User_Messages, that manages the messages forwarded to the user by decoupling their internal (tokenised) and external (textual) representations.
User_Commands, that interacts with the user to read and interpret their commands. Again, the internal and external representations are made independent by parametrisation.
Input, that is responsible for the implementation of low-level input-output operations on characters.
Cells, that actually manage the spreadsheet by updating the cell contents on the basis of the user commands and/or inputs.

While the first three candidate abstract data types were obtained straight away from the criterion's application, the last two were originally assembled in a unique candidate module. The decomposition of this pot-pourri module required a careful analysis of the relationships existing among the procedure-like components, and in particular a preliminary re-engineering of the declaration nesting on the basis of the procedure calls. This re-engineering was performed according to the algorithm proposed by Cimitile *et al.* [18]. More details are given by Canfora *et al.* [11], who also show how the candidate abstract data types can be clustered in reusable modules implemented as TurboPascal Units [13] according to the template proposed by Cimitile [19].

The application of the candidature criterion produced good results also for the program ExamMarker. Here the following candidate abstract data types were obtained:

```
T_SET: candidates,lines,papers
C_SET: checkavailablealternatives,
       checkcandidate,checkextradata,
       readandcheckanswers,writeparticulars
```

```
T_SET: listsizes,markfudge,markschemes,
       questnos,titles
C_SET: getpreliminaries,readtitle,writetitle

T_SET: strings
C_SET: readstring

T_SET: colleges
C_SET: newcollege,readcollege,writecollege

T_SET: listelements
C_SET: alphaprecede,collegeprecede,
       highermark,swap

T_SET: exams,lists,posint,seeds
C_SET: analyse,dice,dumptofile,getparticulars,
       getrandomnumber,histogram,listbycollege,
       listbymark,listbyname,listforstudents,
       listsortedresults,mark,permute,
       quicksort,summarise,validate
```

It was not too difficult to understand the meaning of, and to give semantics to, the first five candidate modules by inspecting the code and interacting with one of the authors of the program. The criterion produced one very simple general purpose module — namely the one implementing the type string as a packed array of characters and the unique operator `ReadString` to read a string up to a maximum length from a file — together with more application domain oriented modules. Exemplar is the case of the module clustering the type `Colleges` together with the operators to distribute the marks of the students on the basis of their belonging to colleges, and produce college marking reports. This module is heavily application oriented as it depends on the organisation of the University of Durham into different colleges, and it would not be useful in the implementation of an exam marker program for universities that do not adopt a college organisation. Interesting, is the module that clusters the type `ListElements` with the operators to compare students' records on the basis of different parameters such as the alphabetical order of their names, their marks or the college which they belong. It is simple to guess that this module is intended to provide a higher level module implementing a list with the operators needed to create different ordering (actually, it also includes a swap operator). Finally, the candidate modules defined around the T_SETs {`Candidates`, `Lines`, `Papers`} and {`ListSizes`, `MarkFudge`, `MarkSchemes`, `Questnos`, `Titles`} implement, respectively, the type CandidatePapers — with the operators for reading and checking the data of a student and his answers (paper), and printing out the resulting mark — and Script — with the operators to read the title of an exam, the total number of questions, the marking scheme, and produce a form echo-printed into a file.

Problems were raised by the understanding of the last candidate module, as it is quite large (16 procedure-like components) and it is not immediately obvious how to associate it with an abstract data type. The analysis of the source code led us to formulate the hypothesis that this module clusters together two different abstract data types, Exams and List. To confirm this hypothesis, and to identify the source of such a bad clustering, we drew the associate type-procedure-connection graph. This graph is shown in Fig. 1 (we use boxes and bubbles to represent procedure-like components and user-defined data types; dotted and solid lines depict the USED-TO-DEFINE and TYP relations). From this graph it is easy to identify that the procedures `DumpToFile` and `Permute` are the cause of the bad clustering. Indeed, `DumpToFile` forces the user-defined data type `Exams` to be clustered together with `Seeds` and `PosInt`, these being in turn clustered with `Lists` because of `Permute`. The analysis of the system's call-graph revealed that the procedure `DumpToFile` is the only one that calls `Permute`. Similarly, `Permute` is the only procedure that calls `Dice`, and `Dice` is the only one that calls `GetRandomNumber`. As a consequence the declaration nesting can be re-engineered in such a way that `Permute` is declared locally to `DumpToFile`, `Dice` to `Permute`, and `GetRandomNumber` to `Dice`. The user-defined data types `Seeds` and `PosInt` can also be made local to `DumpToFile`, this requiring a slight change to the procedure `DumpToFile` to allow it to read the first value of the seed used to generate random numbers with a uniform distribution on [0.0, 1.0]. These numbers are used to randomly rearrange the members of a class when writing into a file the marks obtained for each question in the examination, thus enforcing the anonymity of data. This re-engineering made it possible to split the candidate module into two different modules each of which implements an abstract data type. The modules obtained are linked by a USE relationship, as the abstract data type Exams imports the resource exported by Lists.

Applying the candidature criterion to the program Editor led to the candidature of three modules. The candidate module T_SET = {`TraceString`} and C_SET = {`Etrace`} was immediately discarded after realising that the procedure `Etrace` is actually a debugging routine. The module T_SET = {`LinePtr`} and C_SET = {`AlloLine`, `FreeLine`, `GetInd`, `GetNew`, `GetPak`, `GetTxt`, `LinkUp`} implements the abstract data type Lines. It provides primitives to operate on two lists of lines, the list of the used lines (the ones that currently contain text) and the list of the free lines. Below, is a brief description of the operators and their relationships:

`GetNew` — Allocate space for a new line record.

`AlloLine` — Allocate a line record. If the list of the free lines is empty then `GetNew` is called.

`GetInd` — Locate a line from its number.

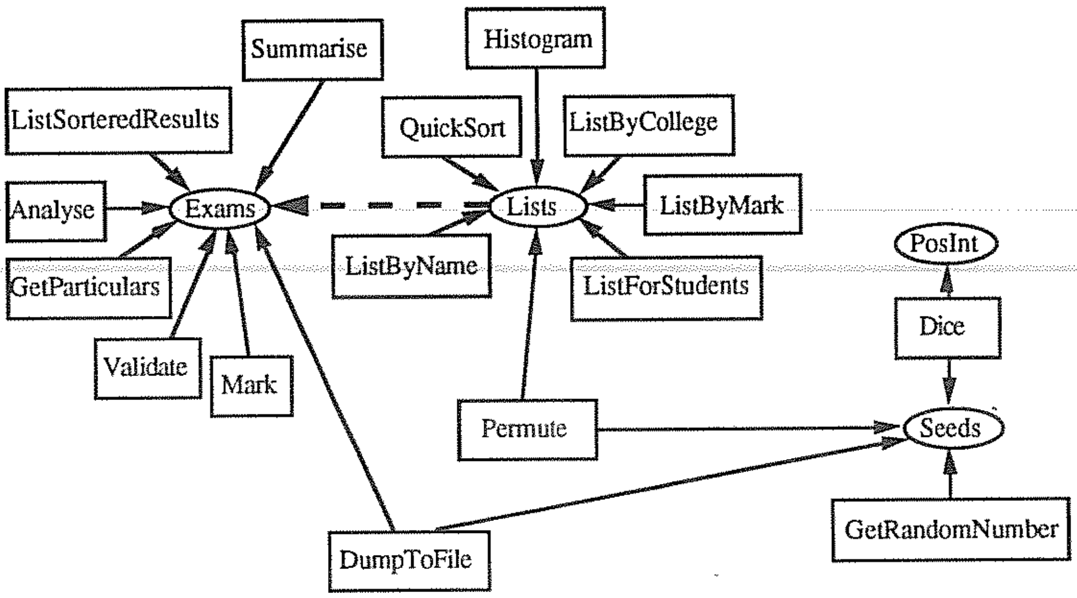


Fig. 1: A partial type-procedure-connection graph for the program Formatter.pas.

`GetPak, GetTxt` — Get the text of a line and its length. Both these functions call `GetInd`.

`LinkUp` — Receive the pointers to two lines, l1 and l2, and make l2 follow l1.

The third candidate module is quite a large pot-pourri module consisting of five user-defined data types and thirtysix procedure-like components. The analysis of the type-procedure-connection graph revealed that the main cause for such a large cluster was the used-defined data type `StatusRange`. It appears in the heading of a large share of the procedure-like components in the cluster. `StatusRange` codifies the possible states at the end of a number of different operations. No procedure-like components has the specific task of accessing or manipulating the status, this being always a consequence of the execution of some other operation. `StatusRange` aggregates procedure-like components on the basis of their control coupling. We decided to isolate `StatusRange` in a module that only defines and exports it. After that we re-applied the criterion to the cluster without taking into account `StatusRange`. This required the manual extraction of the Prolog program dictionary related to the cluster from the overall one produced by the code analyser, and the deletion of the facts that name `StatusRange`. As a result, all the procedure-like components in the cluster that use in their headings an user-defined data type other than `StatusRange` were re-arranged into three new candidate modules. These modules are shown below:

```
T_SET: filenamestring
C_SET: assignfile,doread,dowrit,getfn,open
```

```
T_SET: linestring
C_SET: addset,ctoi,esc,filset,inject,readcmd,
       readline
```

```
T_SET: argstring,patternstring
C_SET: amatch,catsub,dumppat,getccl,getrhs,
       locate,makpat,maksub,match,omatch,
       patsiz,stclos,subst
```

The first module implements the abstract data type Files, responsible for the operations on file streams. The following primitives are implemented: get the name of a file (`GetFn`), assign and open a file (`AssignFile`, `Open`), and write and read a given number of line of text to/from a file (`DoWrit`, `DoRead`). This module is machine dependent as it interacts with the underlying operating system, and in particular with the file system. The second candidate module implements the abstract data type LineStrings, with operators for reading a line from either the terminal (`ReadCmd`) or a file buffer (`ReadLine`), inserting (`Inject`) and modifying a piece of text (`Ctoi`, `Addset`, `Filset`), inserting an escape character (`Esc`). The third module implements a PatternMatcher, with operations for searching and substituting strings (`Subst`, `Catsub`, `Maksub`, `GetRhs`), and pattern matching (`Match`, `Amatch`, `Omatch`, `Locate`, etc.). PatternMatcher uses the module LineStrings. While this module is quite large, the data abstraction it realises is well identified. The large number of procedure-like components is essentially due to the implementation of slightly different versions of a same operation. An example is the pattern matching where a number of procedure-like components implement variants of a same operation (look for match at the beginning, everywhere, or at the end of lines, look forward, backward etc.).

The application of the candidature criterion to the program Formatter produced low interest results. Here three candidate modules were obtained, all of them with a very simple structure — two candidate modules consist of only one user-defined data type and one procedure-like component, the third includes two user-defined data types and one procedure-like component. Although the analysis of the code made it possible to understand the meaning of the candidate modules — for example the module T_SET = {Margins} and C_SET = {ChangeMarginTo} is responsible for implementing one of the layout parametrisations, namely the width of the indentation — their simplicity reduced the interest for electing them for reuse. The poorness of the results is probably due to the original design of the program which was not available to us. Nevertheless, the analysis of the program's call graph and nesting tree, the very few comments in the code, the naming convention, and the detailed study of some sample procedures led us to guess that the program had been designed according to a function decomposition approach rather than to the principle of data abstraction.

The worst results in the set of case studies were obtained for the program Othello. The application of the candidature criterion to this program produced four candidate modules, but the analysis of the code did not lead to the understanding of their meaning. The absence of the program's specification and design and the impossibility to get in touch with the author made it impossible to provide the candidate modules with semantics. The main problem was the lack of knowledge about the way in which the rules of the game have been encoded, and in particular about the linear programming techniques used to rate the possible moves and make the best one.

5 Conclusions

Extracting reusable assets from existing software systems is a promising approach to spread software reuse in the industrial software production environments. Reverse engineering and more general program comprehension techniques can help to search existing system for reusable assets. We have proposed a candidature criterion for producing reusable modules that implement abstract data types. The criterion is founded on logic and Prolog has been used for its rapid prototyping. The strength and the major limitations has been evaluated by presenting results from several case studies carried out within the RE^2 project.

Although the limited extent of the case studies does not allow definitive conclusions there is a number of considerations that can already been drawn. The proposed criterion produced a significant number of well-formed modules, i.e. modules that can be directly associated with abstract data types. When the modules produced were too large, the support the criterion gave to recognise the different abstract data types possibly clustered in a single module was considerable. The clusters produced provide sensible help to understand the structure of data in a software system, and how data are accessed and manipulated.

The quality of the candidate modules greatly depends on the original design of the system. The criterion produces the most satisfactory results if abstract data types have been used in the design, but modules have not been produced to implement them at the code level. For the hybrid systems, that means the systems designed using abstract data types but also including procedure-like components that directly access the supporting structure of abstract data types in order to implement system specific operations, the procedures and functions that create coincidental/spurious connections have to be detected and sliced/deleted. This can be facilitated by graphically showing the results of the candidature criterion through a suitable layout of the type-procedure-connection graph.

Finally, understanding the candidate modules may sometimes require a deep knowledge of the domain of the application, and of the way in which processes, rules, and constraints in the domain are encoded in the software solution.

A larger experimentation is needed to obtain a better understanding of both the strengths and weaknesses of the proposed candidature criterion, and to fully characterise the class of the systems for which it works well. We are currently in the process of producing a new version of the code static analyser to create the Prolog program dictionary. This new version will analyse multi-module programs, thus allowing the criterion to be applied to large systems.

References

[1] Biggerstaff, T. J. (1991) "An Assessment and Analysis of Software Reuse" MCC Technical Report STP-MT-119-91.

[2] Arnold, R. S. and Frakes, W. B. (1992) Software Reuse and Reengineering" CASE Trends, 4(2), pp. 44-48.

[3] Tracz, W. (1988) "Software Reuse Maxims" ACM SIGSOFT, Software Engineering Notes, 13(4), pp. 28-31.

[4] Canfora, G., Cimitile, A. and Munro, M. (1992) "RE^2: Reverse Engineering and Reuse Re-engineering" Computer Science Technical Report

8/92, University of Durham, School of Engineering and Computer Science; to appear on The Journal of Software Maintenance.

[5] Hoare, C. A. R. (1972) "Notes on Data Structuring" in Structured Programming, Academic Press, Inc., London.

[6] Dahl, O. J., and Hoare, C. A. R. (1972) "Hirerchical Program Structures" in Structured Programming, Academic Press, Inc., London.

[7] Calliss, F. W. (1989) "Inter-module Code Analysis Techniques for Software Maintenance" PhD Thesis, University of Durham.

[8] Callis, F. W. and Cornelius B. J. (1989) "Two Module Factoring Techniques" Journal of Software Maintenance, 1, pp. 81-89.

[9] Embley, D. W. and Woolfield, S. N. (1988) Assessing the Quality of Abstract Data Types Written In Ada" Proc of 10th International Conference on Software Engineering, Singapore, IEEE Comp. Cos. Press, pp.144-153.

[10] Liu, S. S. and Wilde, N. (1990) "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery" Proc. of Conference on Software Maintenance CSM'90, San Diego, California, IEEE Computer Society Press, pp. 266-271.

[11] Canfora, G., Cimitile, A. and Munro, M. (1992) "A Reverse Engineering Method for Identifying Reusable Abstract Data Types" Computer Science Technical Report 11/92, University of Durham, School of Engineering and Computer Science.

[12] Kernigham, B. and Pike, R. (1984) "The Unix Programming Environment" Prentice Hall, Inc., Englewood Cliffs, New Jersey.

[13] Turbo Pascal ver. 5.5 (1988) Reference Manual, Borland International.

[14] Canfora, G. and Vargiu, F. (1991) "Reverse Engineering, Reuse Re-Engineering and Visual Environments: the VAPS Project" Proc. of Workshop on Reverse Engineering, Portici (Naples), Italy, Ed. CUEN, pp. 142-175.

[15] Canfora, G., Cimitile, A. and De Carlini, U. (1992) "VAPS: Visual Aids for Pascal Software Comprehension" Position Paper, IEEE Workshop on Program Comprehension, Workshop Notes, Orlando, Florida, pp13-14.

[16] Di Battista, G., Gianmarco, A., Santucci, G. and Tamassia, R. (1990) "The Architecture fo Diagram Server" Proc. of IEEE Workshop on Visual Languages, Skokie, Illinois, pp. 60-65.

[17] Miller, L. H. (1986) "Advanced Programming: Design and Structure Using Pascal" Addison Wesley Publishing Company, Reading, Massachusetts.

[18] Cimitile, A., Di Lucca, G. A. and Maresca, P. (1990) "Maintenance and Intermodular Dependencies in Pascal Environment" Proc. of Conference on Software Maintenance CSM'90, San Diego, California, IEEE Computer Society Press, New York, pp. 72-83.

[19] Cimitile, A. (1992) "Towards Reuse Reengineering of Old Software" Proc. of 4-th International Conference Software Engineering and Knowledge Engineering SEKE'92, Capri, Italy, IEEE Computer Society Press, New York.

[20] Kernighan, B. and Plauger, P. J. (1981) "Software Tools in Pascal" Addison Wesley Publishing Company, Reading, Massachusetts.

Partial Evaluation as an Aid to the Comprehension of Fortran Programs

Sandrine Blazy
EDF DER
1, avenue du Général de Gaulle
92141 Clamart Cedex, France
Sandrine.Blazy@der.edf.fr

Philippe Facon
CEDRIC IIE
18 allée Jean Rostand
91025 Evry Cedex, France
facon@cnam.cnam.fr

Abstract

We describe a technique and a tool supporting partial evaluation of Fortran programs, i.e. their specialization for specific values of their input variables. We aim at understanding old programs, which have become very complex due to numerous extensions. From a given Fortran program and these values of its input variables, the tool provides a simplified program, which behaves like the initial one for the specific values. This tool uses mainly constant propagation and simplification of alternatives to one of their branches. The tool is specified in inference rules and operates by induction on the Fortran abstract syntax. These rules are compiled into Prolog by the Centaur/Fortran environment.

1. Introduction

Program understanding is the most expensive phase of the software life cycle. It is said that 40% of the maintenance effort is spent trying to understand how existing software works [21]. All maintenance problems do not require complete program understanding, but each problem requires at least a limited understanding of how the source code works, and how it is related to the external functions of the application. There exists now a wide range of tools to support program understanding [22].

Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. The slice of a program P on variable X at location i is the set of statements that influence the value of X at i. This is an executable program that is obtained by data flow analysis. Program slicing can be used to help maintainers understand and debug foreign code [11].

We have developed a complementary technique: reduction of programs for specific values of their input variables. It aims at understanding old programs, which have become very complex due to extensive modifications. From a given Fortran program and some form of restriction of its usage (e.g. the knowledge of some specific values of its input variables), the tool provides a simplified program, which behaves like the initial one when used according to the restriction. This approach is particularly well adapted to programs which have evolved as their application domains increase continually.

Partial evaluation is an optimization technique used in compilation to specialize a program for some of its input variables. Partial evaluation of a subject program P with respect to input variables $x_1...x_m$, $y_1...y_n$ for the values $x_1 = c_1...x_m = c_m$ gives a residual program P', whose input variables are $y_1...y_n$ and such that the executions of $P(c_1...c_m, y_1...y_n)$ and $P'(y_1...y_n)$ produce the same results [18]. Such a program is obtained by replacing variables by their constant values, by propagating constant values and simplifying statements, for instance replacing each alternative whose condition simplifies to a constant value (true or false) by the corresponding branch.

Partial evaluation has been applied to program optimization and compiler generation from interpreters (by partially evaluating the interpreter for a given program) [14]. In this context, previous works have especially dealt with functional [2] and logical languages [20]. The structure of the program may be modified (using loop expansion, subroutines expansion and renaming [9]) in order to optimize the residual code.

As far as imperative languages are concerned, partial evaluation has been used for software reuse improvement by restructuring software components to improve their efficiency [3-5]. Partial evaluation has been applied to numerical computation to provide performance improvements for a large class of numerical programs, by eliminating data abstractions and procedure calls [3].

Our goal is different. We remove groups of statements that are never used in the given context, but we do not expand statements. This does not change the original structure of the code. We transform general-purpose programs into shorter and easier to understand special-purpose programs. This transformational approach aims at improving a given program without disturbing its correctness when

0-8186-4042-1/93 $03.00 © 1993 IEEE

used in a given restricted and stable context. However, unlike [15], we do not aim at improving a program according to a performance criterion (e.g. memory), but at improving the readability of programs.

This paper is organized as follows. First, we justify our interest in scientific applications written in Fortran in section 2. Next, we present in section 3 the two main tasks of our partial evaluator: constant propagation and simplification. In section 4, we describe our partial evaluation as a set of inference rules, and we show how these rules combine constant propagation and simplification rules. Section 5 presents conclusions and future work.

2. Scientific programming

A number of scientific applications, written in Fortran for decades, are still vital in various domains (management of nuclear power plants, of telecommunication satellites, etc.). Even though, more recent languages are used to implement the most external parts of these applications. It is not unusual to spend several months to understand such applications before being able to maintain them. In a recent follow up about maintenance practices of scientific applications [12], we have noticed that understanding a 120 000 Fortran lines application took nine months. So, providing the maintainer with a tool, which finds parts of lost code semantics, allows to reduce this compulsory period of adaptation.

2.1. Characteristics

One of the peculiarities of scientific applications is that the technological level of scientific knowledge (linear systems resolution, turbulence simulation, etc.) is higher than the knowledge usually necessary for data processing (memory allocation, data representations). The discrepancy is increased by the widespread use of Fortran, which is an old-fashioned language. Furthermore, for large scientific applications at EDF Fortran 77, which is quite an old version of the language, is exclusively used to guarantee the portability of the applications on different machines (mainframes, workstations, vectorial computers) [1].

2.2. General purpose applications

Our study has highlighted common characteristics in Fortran programming at EDF. These scientific applications have been developed a decade ago. During their evolution, they had to be reusable in new and various contexts. For example, the same thermohydraulic code implements both general design surveys for a nuclear power plant component (core, reactor, steam generator, etc.) and subsequent improvements in electricity production models. The result of this encapsulation of several models in a single large application domain increases the program complexity, and thus amplifies the lack of structures in Fortran programming language.

This generality is implemented by Fortran input variables whose value does not vary in the context of the given application. We distinguish two classes of such variables:

- *data about geometry*, which describe the modelled domain. They appear most frequently in assignment statements (equations that model the problem).
- data taking a finite number of values, which can be represented by logical variables. These are either *filters* necessary to switch the computation in terms of the context (modelled domain), or *tags* allowing to minimize errors risks about the precision of the output values.

Figure 1 shows an example of program reduction. The code section of figure 1-a is extracted from one of the applications we have studied [19]. The partial evaluation of this code section according to the simplification criteria of figure 1-b yields the code section of figure 1-c. A maintenance team is used to update a specific version of the application. These people know some filters properties ($IC = 0$ and $IREX = 1$) as well as data about geometry ($DXLU = 0, 5$). Furthermore IM is a tag whose value is 20.

The knowledge of these values of input variables simplifies the code (as shown in figure 1-c). Because of the truth of the relation $IREX = 1$, two alternatives are simplified (1). The first alternative is simplified to its then-branch. In this branch, the variable DXLU is replaced by its value, which modifies the variables X(I) and DX(I) (2). The variable IM is replaced by its value too (3). The condition of the next alternative is simplified (4) since the value of IC is known. Furthermore, the relation $IC = 0$ simplifies the following alternative to its then-branch (5), which allows to compute the value of the variables ZERO, IREGU and IDECRI (6). Because the values of these three variables are constant values, the three corresponding assignments are removed from the code. Then other alternatives are simplified (7).

This reduction is specially important thanks to the high number of assignments and conditionals. This is the case for most subroutines implementing mathematical algorithms. For subroutines whose main purpose is editing results or calling other subroutines, the reduction is generally not so important.

In order to show how the simplified code has been obtained from the initial one, some links between both codes are shown in figure 1. The initial code which is left unchanged in the simplified code is italicized. Expressions which have been replaced by their value and which appear in the simplified code are written in bold type. The rest of the initial code is the code that has been removed in the simplified code. When using the tool, such links can be visualized in different colours.

```
(1)      IF ( IREX .NE. 0 ) THEN
(3)          DO 111 , I = 1 , IM
(2)              X(I) = XMIN + FLOAT(I-1) * DXLU
         111 CONTINUE
(3)          DO 112 , I = 1 , IM
(2)              DX(I) = DXLU * I
         112 CONTINUE
         ELSE
             READ (NFIC11,*, ERR=1103) X
             DO 121 , I = 1 , IM
                 DX(I) = X(I+1) - X(I)
         121 CONTINUE
         ENDIF
(4)      IF ( IMATSO .EQ. 0 .AND. IC .GE. 0) THEN
(6)          ZERO = 0.
(5)          IF ( IC .EQ. 0 ) THEN
(6)              IREGU = 1
             ELSE IF (IC .GE. IM) THEN
                 IREGU = 0
             ENDIF
(1)          IF ( IREX .EQ.2 ) THEN
                 READ (NFIC11,'(A)',ERR=5,END=5) L
(7)              IF ( INDEX (L,'I') .NE. 0 ) THEN
(7)                  IDECRI = 1
                 ENDIF
             ELSE
(6)              IDECRI = 2
             ENDIF
             IF ( IDECRI .EQ. 1 ) THEN
                 IF ( IREGU .EQ. 0 ) THEN
                     IMIN = 2
                     IMAX = IM
                 ELSE
                     IMIN = IM
                     IMAX = IM
                 ENDIF
             ELSE IF ( IDECRI .EQ. 2 ) THEN
                 IF ( IREGU .EQ. 0 ) THEN
                     JMIN = 2
                     JMAX = JM
                 ELSE
                     JMIN = JM
                     JMAX = JM
                 ENDIF
              ENDIF
         ENDIF
```

Fig. 1-a. Initial code

```
IREX = 1
IC = 0
IM = 20
DXLU = 0,5
IREX = 1
```

Fig. 1-b. Constraints on input variables

```
        DO 111 , I = 1 , 20
            X(I) = XMIN + 0,5*FLOAT(I-1)
111 CONTINUE
        DO 112 , I = 1 , 20
            DX(I) = 0,5*I
112 CONTINUE
        IF ( IMATSO .EQ. 0 ) THEN
            JMIN = JM
            JMAX = JM
        ENDIF
```

Fig. 1-c. Simplified code

Fig. 1. An example of program simplification

3. Two aspects of partial evaluation applied to imperative programs

Our partial evaluator performs two main tasks: constant propagation through the code and simplification of statements. The tool can give the result of one of the two independently of the other. For instance, if the user is a physicist who is familiar with the equations implemented in the code, he wishes to locate in Fortran statements these equations and their variables, as they appear in the formulae of these equations. But if the user is a maintainer who does not know the application well, he would rather visualize the code as simplified as possible. In all cases however and for an optimal partial evaluation, the tool performs both tasks.

3.1. Constant propagation

Constant propagation is a well-known global flow analysis technique used by compilers. It aims at discovering values that are constant on all possible executions of a program and to propagate forward through the program these constant values as far as possible. Some algorithms now exist to perform fast and powerful constant propagation [23].

We describe in this section our constant propagation process. It modifies most expressions by replacing some variable occurrences by their values and by normalising all expressions through symbolic computation. Presently, our tool propagates only *equalities between variables and constants*. Of course, that limits the precision of the analysis.

Substitution. Before running the partial evaluator, the user specifies numerical values for some input variables of the program (thanks to his personal knowledge of the application). Constant propagation spreads this initial knowledge supplied by the user. In a first stage, the partial evaluator replaces each specified variable by its value. Then, expressions whose operands are all constant values are computed and these resulting values are propagated forward through the whole program. This technique allows to remove from the code all occurrences of variables identifiers that are no more meaningful. The substituted values may be visualized differently from other values (with bold type as in the previous example or with a different colour).

Furthermore, the user can specify some variables that will not be substituted by their value. For instance, he can indicate that the variable PI will not be replaced by 3.1416. Expressions containing only such variables and numerical values are considered as constant values.

Normalisation. For any given numerical expression, we have to recognize if it reduces to a constant value (e.g. x+3-x reduces to 3). In the same way, for any given logical expression, we have to recognize if it reduces to a conjunction of equalities or to a disjunction of inequalities. In the first case (respectively the second case), we will be able to propagate equalities in the then- (respectively else-) branch of alternatives. To do this, we perform constant propagation. To propagate constant values as most as possible, our system normalises each expression into a canonical form: a polynomial form for numerical expressions and a conjunctive normal form for the logical expressions.

In a polynomial form, expressions are simplified by computing the values of the coefficients of the polynomial. Polynomial forms are written according to the decreasing powers order. When some terms of a polynomial have the same degree (e.g. z^2, x^2 and t.u), they are sorted according to a lexicographical order (e.g. $t.u < x^2 < z^2$). The canonical form of a relational expression is obtained from the canonical forms of its two numerical subexpressions. In normalized relational expressions, all variables and values occur on only one side of the operator.

Because of these modifications of expressions, overflow, underflow or round-off errors may happen. Therefore, the normalization of expressions may cause run-time errors. It is possible to obtain programs that will cause machine errors when compiled and executed. Conversely, some run-time errors may vanish thanks to the partial evaluation. As most partial evaluation systems [17], our tool ignores such problems. In this case only, the tool does not yield a program which behaves like the initial one.

3.2. Simplification

Simplification is an option of the partial evaluator. First, the expressions are simplified during the propagation as explained above. Then, the simplification process reduces the size of the code by removing both assignments of variables which can be evaluated to constants and statements which are never used for the specified values. This simplification includes the elimination of redundant tests and in particular the simplification of alternatives to one case thanks to the evaluation of their conditions. To simplify a statement means to remove or to modify it. Its components must be simplified, but in different contexts. This section defines the simplification for each statement.

A *write* statement is simplified by simplifying its parameters that are expressions. A *read* statement is simplified by removing its parameters whose values are known input values. If all its parameters have known values, the read statement is removed (replaced by an empty statement). Since the removed parameters do not appear in the residual program any more, their initialization is not missing in the code that is therefore still executable.

An *assignment* simplification consists in simplifying the assignment expression or in removing it in some cases.

An *alternative* is simplified into one of its branches when its condition has been evaluated to either true or false. Otherwise, the statements of the two branches are simplified. In this case a branch may become the empty statement.

Loops that are never entered are removed. When discovered, infinite loops are left unchanged. Otherwise, the only statements of the loop which are simplified by the knowledge of variables values, are those statements whose expressions are invariant. Thus, we do not expand loops because we want to keep the original structure of the code. Furthermore, Fortran loops are implemented using labels and goto statements. So, when a loop is removed, its label statement is kept when other statements contain goto statements to such labels (the label is left unchanged and the statement is replaced by a skip statement).

A *call* statement simplification consists in replacing its actual parameters whose values are known by these values. The identifier of the called subroutine is left unchanged in the current program (subroutines are not necessarily specialized) and the user has to run the partial evaluator on this subroutine code if he wants to simplify it too.

The partial evaluation does not simplify other statements. Let us notice that our tool yet can neither deal with inequalities nor with literal values.

4. Inference rules for partial evaluation

To specify the partial evaluation, we use inference rules operating on the Fortran abstract syntax and expressed in the natural semantics formalism [15], augmented by some VDM [13] operators. This section first presents rules defining on the one hand the constant propagation process and on the other hand the simplification process. Then, it details the rules for partial evaluation of statements. These new rules combine the propagation rules and the simplification rules. Let us notice that the techniques we implement are not new, but we specify and use them in a novel way.

4.1. Propagation and simplification rules

In the following, we use sequents such as $H \vdash I{:}H'$ (propagation), $H \vdash I \longrightarrow I'$ (simplification), and the combination of both $H \vdash I \longrightarrow I', H'$ (propagation and simplification). In these sequents:

- H is the environment associating values to variables whose values are known before executing I. It is modelled by a VDM-like map [13], shown as a collection of pairs contained in set braces such as {variable $\rightarrow$ constant, ...}, where no two pairs have the same first elements. Our system initializes such maps by the list of variables and their initial values, supplied by the user.
- I is a Fortran statement (expressed in a linear form of the Fortran abstract syntax).
- I' is the simplified statement under the hypothesis H.
- H' is H which has been modified by the execution of I.

The sequents such as $H \vdash I{:}H'$ express the propagation relation. The sequents such as $H \vdash I \longrightarrow I'$ express the simplification under hypotheses. It depends of the components of I, which are themselves simplified under other hypotheses. Thus, the definition of the simplification relation uses the definition of the propagation relation.

In the sequents, we use the map operators *dom*, $\cup$, $\dagger$, $\lhd$ and $\ntriangleleft$.

- The domain operator *dom* is the set of the first elements of the pairs in the map.
- The union operator $\cup$ yields the union of maps whose domains are disjoint (this operator is undefined if the domains overlap).
- The map override operator $\dagger$ whose operands are two maps, yields a map which contains all of the pairs from the second map and those pairs of the first map whose first elements are not in the domain of the second map.
- The map restriction operator $\lhd$ is defined with a first operand which is a set and a second operand which is a map; the result is all of those pairs in the map whose first elements are in the set.
- When applied to a set and a map, the map deletion operator $\ntriangleleft$ yields those pairs in the map whose first elements are not in the set. The example of figure 2 illustrates these definitions.

$$m = \{X \rightarrow 5, B \rightarrow true\}$$
$$dom(m) = \{X, B\}$$
$$m \cup \{Y \rightarrow 7\} = \{Y \rightarrow 7, X \rightarrow 5, B \rightarrow true\}$$
$$\{X, Z\} \lhd m = \{X \rightarrow 5\}$$
$$\{B\} \ntriangleleft m = \{X \rightarrow 5\}$$
$$n = \{C \rightarrow false, X \rightarrow 8\}$$
$$m \dagger n = \{B \rightarrow true, C \rightarrow false, X \rightarrow 8\}$$
$$n \dagger m = \{B \rightarrow true, C \rightarrow false, X \rightarrow 5\}$$

Fig. 2. Some map operators

We have written some rules to explain how sequents are obtained from other sequents. A rule is composed of a possibly empty set of sequents on the numerator, the rule premises, and of a sequent at the denominator, the conclusion of the rule. If the premises hold, then the conclusion holds.

The rules we present in figure 3 express the simplification of logical or numerical expressions. They belong to the *eval* system, which is a subsystem of the simplification system $\longrightarrow$. The first rule has no premise. It specifies that a variable X which belongs to the environment is simplified into a constant which is equal to its value C. Otherwise (second rule), the variable is not modified. To evaluate an expression E1 OP E2 to the value T, its two operands E1 and E2 must have been evaluated to E'1 and E'2 respectively, and the value T is the result of the computation of E'1 OP E'2 (through the *comp* system). If E'1 and E'2 are both constants (respectively N1 and N2), the computation of T is processed by the application of the *app* primitive to the operator OP and to its two operands N1 and N2.

eval

$$H \cup \{X \to C\} \vdash id(X) \longrightarrow C$$

$$\frac{X \notin dom(H)}{H \vdash id(X) \to id(X)}$$

$$\frac{H \vdash E1 \longrightarrow E'1 \qquad H \vdash E2 \longrightarrow E'2 \qquad \overset{comp}{\vdash} OP, E'1, E'2 : T}{H \vdash E1\ OP\ E2 \longrightarrow T}$$

$$\frac{Ei \neq number\,(N) \qquad Ei \neq bool\,(B)}{\overset{comp}{\vdash} OP, E1, E2 : E1\ OP\ E2} \quad \text{for } i=1,2$$

$$\frac{app\,(OP, N1, N2, T)}{\overset{comp}{\vdash} OP, number(N1), number(N2) : T}$$

$$\frac{app\,(OP, N1, N2, T)}{\overset{comp}{\vdash} OP, bool(B1), bool(B2) : T}$$

with

app (+, I, J, *number*(Z)) :- Z is I + J.
app (=, I, J, *bool*(true)) :- I = J.
app (=, I, J, *bool*(false)) :- not (I = J).
app (and, *bool*(false), C, *bool*(false)).
app (and, *bool*(true), C, C).
app (or, *bool*(true), C, *bool*(true)).
app (or, *bool*(false), C, C).
....

properties of logical expressions

by using C-Prolog like evaluation predefinite primitives.

Fig. 3. Simplification of expressions

The rule of figure 4-a expresses the propagation through a sequence of statements.

To propagate the environment H through the sequence of statements I1;I2, H is propagated through the statement I1, which updates H in H1. This new environment H1 is propagated through I2, which updates H1 in H2. H2 is the environment resulting from the propagation through the sequence.

$$\frac{H \vdash I1 : H1 \qquad H1 \vdash I2 : H2}{H \vdash I1; I2 : H2}$$

Fig. 4-a. Propagation through a sequence of statements

The rule of figure 4-b expresses the simplification of such a sequence. Given an environment H, to simplify a sequence of statements I1;I2 the first statement I1 is simplified in I'1, and the environment H is propagated through I1. In this new environment H1, the second statement I2 is then simplified.

$$\frac{H \vdash I1 \longrightarrow I'1 \qquad H \vdash I1{:}H1 \qquad H1 \vdash I2 \longrightarrow I'2}{H \vdash I1;\, I2 \longrightarrow I'1;\, I'2}$$

Fig. 4- b. Simplification of a sequence

Figure 5 presents some simplification and propagation rules for alternatives. If the condition C of an alternative evaluates to true, then:

- the environment H' resulting from the propagation of H through the alternative is obtained by propagating H through the statements of the then-branch (first rule: propagation),
- the simplification of the alternative is the simplification of its then-branch (second rule: simplification).

The last rule of figure 5 is a propagation rule. It shows that information can sometimes be derived from the equality tests that control alternatives. If the condition of an alternative is expressed as an equality such as X=E, where X is a variable that does not belong to the domain of the environment H and E evaluates to a constant N, then this equality is added to the environment related to the statements of the then-branch (but this equality is not inserted as assignment in the code).

There is a corresponding rule for a condition of an alternative expressed as an inequality such as X≠E: this condition is transformed in an equality such as X=E which is added to the environment related to the statements of the else-branch. Such rules have been generalized to conditions of alternatives expressed as conjunctions of equalities and disjunctions of inequalities. The last rule expresses that only equalities between variables and constants can be added to the environment. Thus, if other information are expressed in the condition, they are not taken into account by the partial evaluator.

$$\frac{H \vdash C \longrightarrow \text{true} \qquad H \vdash I1{:}H'}{H \vdash \text{if C then I1 else I2 fi: } H'}$$

$$\frac{H \vdash C \longrightarrow \text{true} \qquad H \vdash I1 \longrightarrow I'1}{H \vdash \text{if C then I1 else I2 fi} \longrightarrow I'1}$$

$$\frac{\begin{array}{ll} H \vdash E \longrightarrow \text{number}(N) & X \notin \text{dom}(H) \\ H \cup \{X \rightarrow N\} \vdash I1{:}H1 & H \vdash I2{:}H2 \end{array}}{H \vdash \text{if } (X = E) \text{ then I1 else I2 fi: } H1 \cap H2}$$

Fig. 5. Some rules for alternatives

Note that the dynamic semantics is a special case of our system: if the initial environment associates values to all input variables, the final environment will give us (among others) the values of all output variables, with the executable part of the program sometimes simplified to skip. Thus, a very special use of the implementation of our system is to see it as a standard interpreter.

To prove eventually the correctness and completeness of our simplification, with respect to the dynamic semantics, we have to prove that two derived inference rules hold in the union of the two systems (simplification and propagation). These two inference rules are presented in figure 6, where *sem* is the system formalizing the dynamic semantics of Fortran. We are currently investigating such proof, and we use the similarity with rules for the proof of translation [8].

$$\frac{S1 \vdash P \rightarrow P1 \qquad S1 \cup S \overset{sem}{\vdash} P : S'}{S \overset{sem}{\vdash} P1 : S'} \quad \textit{(completeness)}$$

$$\frac{S1 \vdash P \rightarrow P1 \qquad S \overset{sem}{\vdash} P1 : S'}{S1 \cup S \vdash P : S'} \quad \textit{(correctness)}$$

Fig. 6. Correctness and completeness of the simplification

Since the simplification is performed in the context of the propagation, and the propagation uses the simplification of expressions, we have chosen to represent rules grouping propagation and simplification. They are very close to the rules we have implemented in the Foresys [10] toolkit, which compile them into Prolog. Foresys has been built upon the Centaur/Fortran environment [5].

4.2. Combined rules

For every Fortran statement, we have written rules that describe the combination of the propagation and simplification systems. This combination $\longrightarrow$ of these two systems is defined by:

$I \longrightarrow I', H'$ iff $H \vdash I{:}H'$ and $H \vdash I \longrightarrow I'$.

From this rule, we may define inductively the $\longrightarrow$ relation. For instance, figure 7 shows the rule for a sequence of statements. A sequence is evaluated from left to right: The partial evaluation of a sequence of two statements I1 and I2 consists in simplifying I1 in I'1 then in updating (adding, deleting or modifying) the data environment H. In this new data environment H1, I2 is simplified in I'2 and H1 is modified in H2.

$$\frac{H \vdash I1 \longrightarrow I'1, H1 \qquad H1 \vdash I2 \longrightarrow I'2, H2}{I1; I2 \longrightarrow I'1; I'2, H2}$$

Fig. 7. Partial evaluation of a sequence of statements

In the sequel of this paper, we call such rules partial evaluation rules.

Figure 8 specifies the rules for partial evaluation of assignments. The *eval* notation refers to the formal system of rules which simplifies the expressions, that we have previously presented.

If the expression E evaluates to a numerical constant N, the data environment H is modified: the value of X is N whether X had already a value in H or not. With the kind of propagation we perform, the assignment X := E can be removed only if all possible uses of that value of X do not use another value of X. For instance, in the sequence:

X := 2; if CODE ≠ 5 then X := X+1 fi; Y := X

the value 2 of X is propagated in the expression X+1 but the assignment Y:=X can not be removed because the assignment Y:=X may use two different values of X. Thus, that sequence is simplified into:

X := 2; if CODE ≠ 5 then X := 3 fi; Y := X.

To detect such situations we use classical «dead code elimination» algorithms.

If E is only partially evaluable in E', the expression E is modified in consequence in the assignment X:= E and the variable X is removed from the environment if it was in it.

$$\frac{H \overset{eval}{\vdash} E \longrightarrow \mathit{number}(N)}{H \vdash X := E \longrightarrow \text{skip}, H\dagger\{X \to N\}}$$

$$\frac{H \overset{eval}{\vdash} E \longrightarrow E' \qquad E' \neq \mathit{number}(N)}{H \vdash X := E \longrightarrow X := E', \{X\} \lhd H}$$

Fig. 8. Partial evaluation of assignments

The following examples illustrate these two cases. In example *Ex.1*, as the value of the variable A is known, the new value of the assigned variable C is introduced in the data environment. We suppose that the assignment C := A+1 can be removed from the simplified program. In example *Ex.2*, after the partial evaluation of the expression A+B, the value of C has become unknown. Such a case only happens when A and B do not have both constant values.

Ex.1 $\{A \to 1, C \to 4\} \vdash C := A+1 \longrightarrow \text{skip}, \{A \to 1, C \to 2\}$
Ex.2 $\{A \to 1, C \to 2\} \vdash C := A+B \longrightarrow C := 1+B, \{A \to 1\}$

The rules for partial evaluation of alternatives are defined in figure 9. If the condition C evaluates to a logical constant, the alternative with condition C can be simplified to the corresponding simplified branch. If C is only partially evaluated in C', the partial evaluation proceeds along both branches of the alternative. It leads to two different environment H'1 and H'2. Their intersection (that is the identifier/value pairs common to both environments) is the final environment.

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{false}) \qquad H \vdash I2 \longrightarrow I'2, H'}{H \vdash \text{if C then I1 else I2 fi} \longrightarrow I'2, H'}$$

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}(\text{true}) \qquad H \vdash I1 \longrightarrow I'1, H'}{H \vdash \text{if C then I1 else I2 fi} \longrightarrow I'1, H'}$$

$$\frac{\begin{array}{ll} H \overset{eval}{\vdash} C \longrightarrow C' & C' \neq \mathit{bool}(B) \\ H \vdash I1 \longrightarrow I'1, H'1 & H \vdash I2 \longrightarrow I'2, H'2 \end{array}}{H \vdash \text{if C then I1 else I2 fi} \longrightarrow \text{if C' then I'1 else I'2 fi}, H'1 \cap H'2}$$

Fig. 9. Partial evaluation of alternatives

As for alternatives, the rules for partial evaluation of loops, presented in figure 10, depend on the ability to evaluate the truth or falsity of the condition C from the current environment H. The first rule specifies that if the loop is not entered, it is removed from the code. There is no specific rule for the case where C evaluates to true, because we do not expand loops, not to alter the structure of the code. Figure 10 shows the rules for while-statements, but similar rules exist for repeat-statements.

In the second rule, if C evaluates in C' (and C' differs from false), the statements I of the loop can be simplified, given H restricted to a loop invariant Inv(I). Inv(I) is a pessimistic estimation of the variables that are not modified in the loop. It is calculated by the partial evaluator and consists in a list of variables whose values are known and that are neither in a left-hand side of an assignment, nor a parameter of a call or a read statement. The sequent that transforms I into I' belongs to the simplification system. Since we have imposed a pessimistic loop invariant, we have not written a sequent referring to the system $\longrightarrow$: performing the propagation through I would not have modified the restricted environment.

$$\frac{H \overset{eval}{\vdash} C \longrightarrow \mathit{bool}\ (\text{false})}{H \vdash \text{while C do I end} \longrightarrow \text{skip, H}}$$

$$\frac{\text{Inv(I)} \lhd H \overset{eval}{\vdash} C \longrightarrow C' \qquad C' \neq \mathit{bool}\ (\text{false}) \qquad \text{Inv(I)} \lhd H \vdash I \longrightarrow I'}{H \vdash \text{while C do I end} \longrightarrow \text{while C' do I' end}, \text{Inv(I)} \lhd H}$$

Fig. 10. Partial evaluation of while-loops

5. Conclusion

We have used partial evaluation for programs which are difficult to maintain because they are too general. Specialized programs for some values of their input variables are obtained by propagating these constant values (through a normalisation of the expressions) and by performing simplifications on the code, for instance assignments are removed and alternatives are reduced to one of their branches. This technique helps the maintainer to understand the program behaviour in a particular context. The residual program is furthermore more efficient because many statements and variables have been removed in it, and no additional statement has been inserted. Another advantage of this technique is that it can also be applied to abstractions at a higher level than the code (e.g. it can be applied to algorithms). Note that the techniques we develop are not new, but we specify (inference rules), implement (Centaur) and use them (for program comprehension) in a novel way.

Our tool may be used in two ways: by visualizing the residual program as a part of the initial program (for documentation or for debugging) or by generating this residual program as an independent (compilable) program.

We are also focusing on the possibility for the user to supply general properties about input variables. These general properties are for instance relational expressions composed of some literal values (e.g. $x<z+4$) instead of only equalities to constant values. We will consequently take into account that kind of information in the conditions of alternatives and loops. We intend to apply linear resolution methods and symbolic manipulation packages for Fortran [7] to propagate such properties.

References

[1] *Fortran* ANSI standard X3.9 1978.

[2] V.Ambriola, F.Giannotti, D.Pederschi, F.Turini *Symbolic semantics and program reduction* IEEE Trans. on Software Engineering, 11(8), 08 /85, 784-794.

[3] L.O.Andersen *C program specialization* Master's thesis, University of Copenhagen, May 1992.

[4] A.Berlin, D.Weise *Compiling scientific code using partial evaluation*, Computer, December 1990, 25-37.

[5] *Centaur 1.1 documentation* INRIA, January 1990.

[6] A.Coen-Porsini, F.De Paoli, C.Ghezzi, D.Mandrioli *Software specialization via symbolic execution* IEEE Trans. on Software Engineering, 17(9), September 1991, 884-899.

[7] P.D.Coward *Symbolic execution systems - a review* Software Engineering Journal, November 1988, 229-239.

[8] J.Despeyroux *Proof of translation in natural semantics* Symp. Logic in Comp. Science, Cambridge USA, June 86.

[9] A.P.Ershov, B.N.Ostrovski *Controlled mixed computation and its application to systematic development of language-oriented parsers* Program Specification and Transformation, IFIP'87, 31-48.

[10] Connexité *Reference manual of the Foresys line of software products* 1993.

[11] K.B.Gallagher, J.R.Lyle *Using program slicing in software maintenance* IEEE Transactions on Software Engineering, 17(8), August 1991, 751-761.

[12] M.Haziza, J.F.Voidrot, E.Minor, L.Pofelski, S.Blazy *Software maintenance: an analysis of industrial needs and constraints* IEEE Conference on Software Maintenance, Orlando, USA, November 1992.

[13] C.B.Jones *Systematic software development using VDM* Prentice-Hall, 2nd eds., 1990.

[14] N.D.Jones, P.Sestoft, H.Sondergaard *MIX: a self-applicable partial evaluator for experiments in compiler generation* Lisp and Symbolic Computation 2, 1989, 9-50.

[15] G.Kahn *Natural semantics* Proceedings of STACS'87, Lecture Notes in Computer Science, vol.247, March 1987.

[16] V.Kasyanov *Transformational approach to program concretization* Th. Computer Science, 90, 1991, 37-46.

[17] R.Kemmerer, S.Eckmann *UNISEX: a UNIX-based Symbolic Executor for Pascal Software Practice and Experience*, 15(5), 1985, 439-457.

[18] U.Meyer *Techniques for evaluation of imperative languages* ACM SIGSOFT, March 1991, 94-105.

[19] G.Nicolas & al. *A finite volume approach for 3D two phase flows in tube bundles: the THYC code* Kernforschungscentrum, Karlsruhe, Vol.2, 1989, 1247-1253.

[20] D.Sahlin *An automatic partial evaluator for full Prolog* Ph.D. thesis, SICS, Copenhaguen, March 1991.

[21] T.H.Sneed *The myth of 'top-down' software development and its consequences* IEEE Conference on Software Maintenance, Miami USA, October 1989, 22-29.

[22] H.J.Van Zuylen *Understanding in reverse engineering The REDO handbook* Wiley eds., September 1992.

[23] M.N.Wegman, K.Zadeck *Constant propagation with conditional branches* ACM Trans. on Programming Languages and Systems 13(2), April 1991, 181-210.

Facilitating the Comprehension of C Programs : An Experimental Study

Panagiotis Linos
Tennessee Technological University
Computer Science Department
Cookeville, TN 38505, USA
email : PKL3678@tntech.edu

Philippe Aubet, Laurent Dumas,
Yan Helleboid, Patricia Lejeune
and Philippe Tulula
Institut Superieur D'Electronique Du Nord
41 Boulevard Vauban
59046 Lille Cedex, France

Abstract

A software environment called **C.A.R.E.** (Computer-Aided **Re**-engineering) that facilitates the comprehension of existing C programs is described in this paper. Program comprehension in CARE is accomplished by visualizing program dependencies (i.e. entities and their relations). A repository of such dependencies is maintained and displayed using a graphical model which combines control and data-flow information. Moreover, CARE entails transformation tools and abstraction mechanisms that support *monolithic* and *multiple-view* organization of program dependencies. Results from an experimental study with the CARE environment has shown that the productivity of its users was increased and the quality of the changes made during a software maintenance task was improving. Finally, the lessons learned from an empirical evaluation of the CARE environment indicated that its graphical model, transformation tools and abstraction mechanisms constitute a promising platform for the comprehension of C programs.

I. Introduction

Since program understanding is a key issue in software maintenance, the study of various approaches to program comprehension becomes a compelling issue [11]. Understanding programs is a time consuming and tedious task because very often the only alternative for the software engineers is reading a poorly documented source code. Visualization of program dependencies (i.e. entities and their relations) is a step toward a better understanding of the internal structure of programs [12]. Currently, there are several software maintenance environments which utilize graphical models in order to display structural and functional dependencies of programs. Some examples of such environments are the CIA System [10], the Act and BattleMap [8], VIFOR [9] and the Dependency Analysis Tool Set [8]. However, the resulting complex drawings and the display of only one relation between entities are some of the distinct limitations of these environments. In addition, the lack of abstraction mechanisms and transformation tools that facilitate program understanding constitute another weakness of existing software maintenance tools. Therefore, software environments with effective display models, abstraction mechanisms and transformation tools are needed to facilitate the comprehension of existing programs [1].

In this paper, we present an environment called C.A.R.E. (Computer Aided Re-engineering) which facilitates the comprehension of existing C programs [3]. The implementation of CARE is an effort aimed at providing practical solutions to the limitations of existing software tools and it is based on our past experience with VIFOR (Visual Interactive Fortran), a software environment for maintaining Fortran 77 programs [9]. In addition, the results from an experimental study with the CARE environment are discussed which support our hypothesis with respect to the performance of its users during software maintenance. Finally, the lessons learned from an empirical evaluation of CARE and how these lessons were used in order to improve its functionality and user interface are also included in this study.

The rest of this paper is organized as follows: the next section includes our motivation and hypothesis. A description of the CARE environment follows in the third section. The fourth section, includes the experimental study of CARE and the next section discusses the lessons learned from that study. Finally, the history of the CARE environment is entailed in the last section of this paper.

II. Motivation and Hypothesis

Some of the weaknesses of existing software maintenance environments is the lack of efficient display models for program dependencies as well as the limited abstraction mechanisms and transformation tools that facilitate program comprehension. The CARE environment provides generalized graphical support by capturing several program dependencies within the same graphical representation. In particular, CARE facilitates program comprehension by introducing a display model which is a combination of a graphical representation of the data-flow (i.e. use of variables, constants, data types and parameter passing) called *colonnade* and the traditional hierarchical display of the control-flow (i.e. call-graph). In addition, CARE entails efficient abstraction mechanisms and transformation

0-8186-4042-1/93 $03.00 © 1993 IEEE

tools that facilitate the comprehension of existing C programs.

For the purpose of this research, we hypothesize that the combination of the colonnade and the hierarchical display of program dependences constitute a promising platform for the comprehension of the overall architecture of C programs. Moreover, code visualization facilities, transformation tools and graphical abstraction mechanisms entailed in the CARE environment enable software engineers to improve the quality of changes made on existing C programs and increase their productivity.

III. The CARE Environment

C.A.R.E. (Computer-Aided Re-engineering) is a software environment for facilitating the comprehension of existing C programs. The CARE environment enables the comprehension of existing C programs by maintaining a repository of program dependencies (i.e. entities and their relations). *Monolithic* and *multiple-view* visualization of program dependencies, transformation tools between different representations (i.e. graphical and textual) and abstraction mechanisms are some of the important features of CARE.

The CARE environment supports a compact data model for C programs which consists of five entity and four relationship sets as shown in figure 1. The *function, constant, variable, type* and *parameter* are the entity sets. The relations among these entities are the *calls* ,*uses* , *has* and *defined-as* relations. A function can *call* another function, *use* constants and variables and *have* formal parameters. Also, constants, variables and parameters can be *defined* in terms of a certain type. This information about C programs is maintained and graphically presented in the CARE environment.

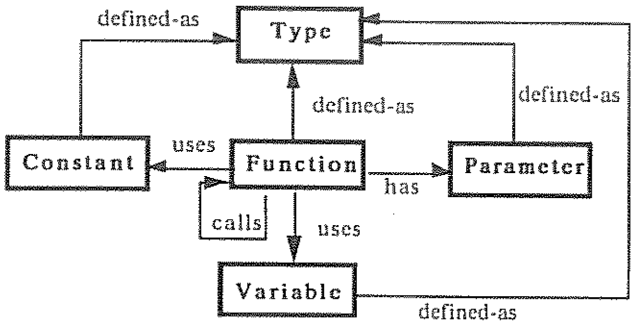


Figure 1 : The C language data model used by CARE.

The architecture of CARE comprises of two main modules; the *code analyzer* and the *display manager* as shown in figure 2. The *code analyzer* parses *classic* or *ansi* C source code and populates a repository with program entities and relationships. The *display manager* transforms the information found in the repository into various representations. It entails two graphical editors; the *colonnade* and the *hierarchical* editor. The colonnade editor displays the entity sets of the data model in different columns and the relationship sets as connecting lines between columns. The call relationships among functions (i.e. call-graph) are displayed by the hierarchical graph editor. The repository of program dependences maintained by CARE can be queried and displayed on specially designed user interface windows. Figure 3 displays the main window of CARE and two graphical windows. The main window consists of six icons; the first one represents the *Project Manager* where the user can set or select a working directory. The second icon is used for default settings. The third icon is for exiting CARE. The *Parser* icon is used to invoke the *Code Analyzer* for parsing C files and populating a repository of program dependencies, the next icon invokes the hierarchical editor for displaying call-graphs and the last icon represents the colonnade graphical editor. In the same figure but separate windows, two graphical representations of program dependencies are displayed. The graphical model for displaying program dependencies supported by the CARE environment is described next.

A.Visualization of Program Dependencies

Understanding programs in CARE is accomplished by maintaining and visualizing program dependencies using a novel graphical model. This model combines the hierarchical display of the control-flow (i.e. call-graph) and a graphical representation of the data-flow (i.e. variables, constants, types and parameters) of the program called *colonnade*.

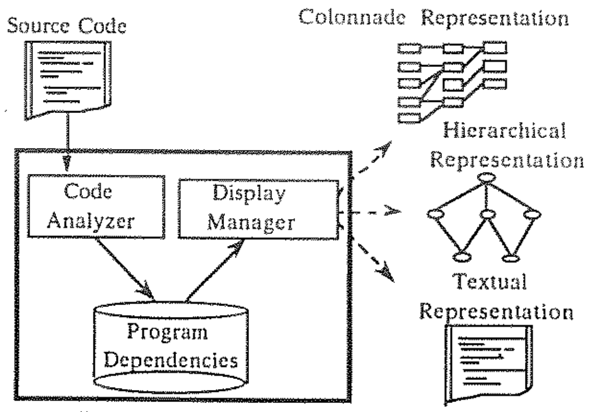


Figure 2 : The Architecture of CARE.

A colonnade is an extension of the *two-column* display used by VIFOR [9] and it has been formally defined as a *m-column* graph [2]. The novelty of the colonnade comes from the fact that it provides generalized graphical support by capturing several program dependencies within the same graphical representation. Figure 4 contains a colonnade representation for part of a C program. It consists of five columns. The first column contains all local and global variables of the program, the second entails types, the third column includes parameters, the next the functions and the

Figure 3: The main window of CARE and two graphical displays of program dependencies.

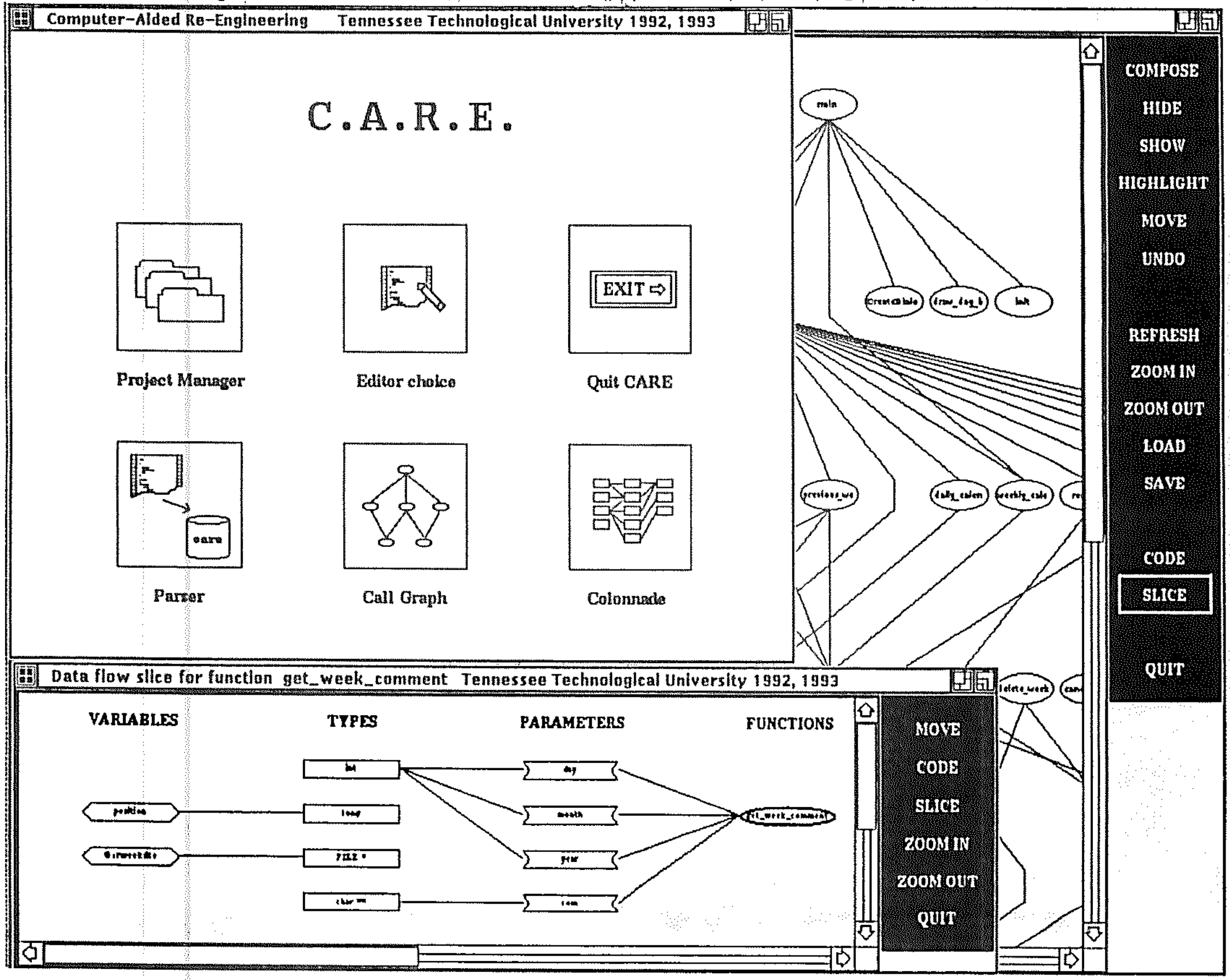
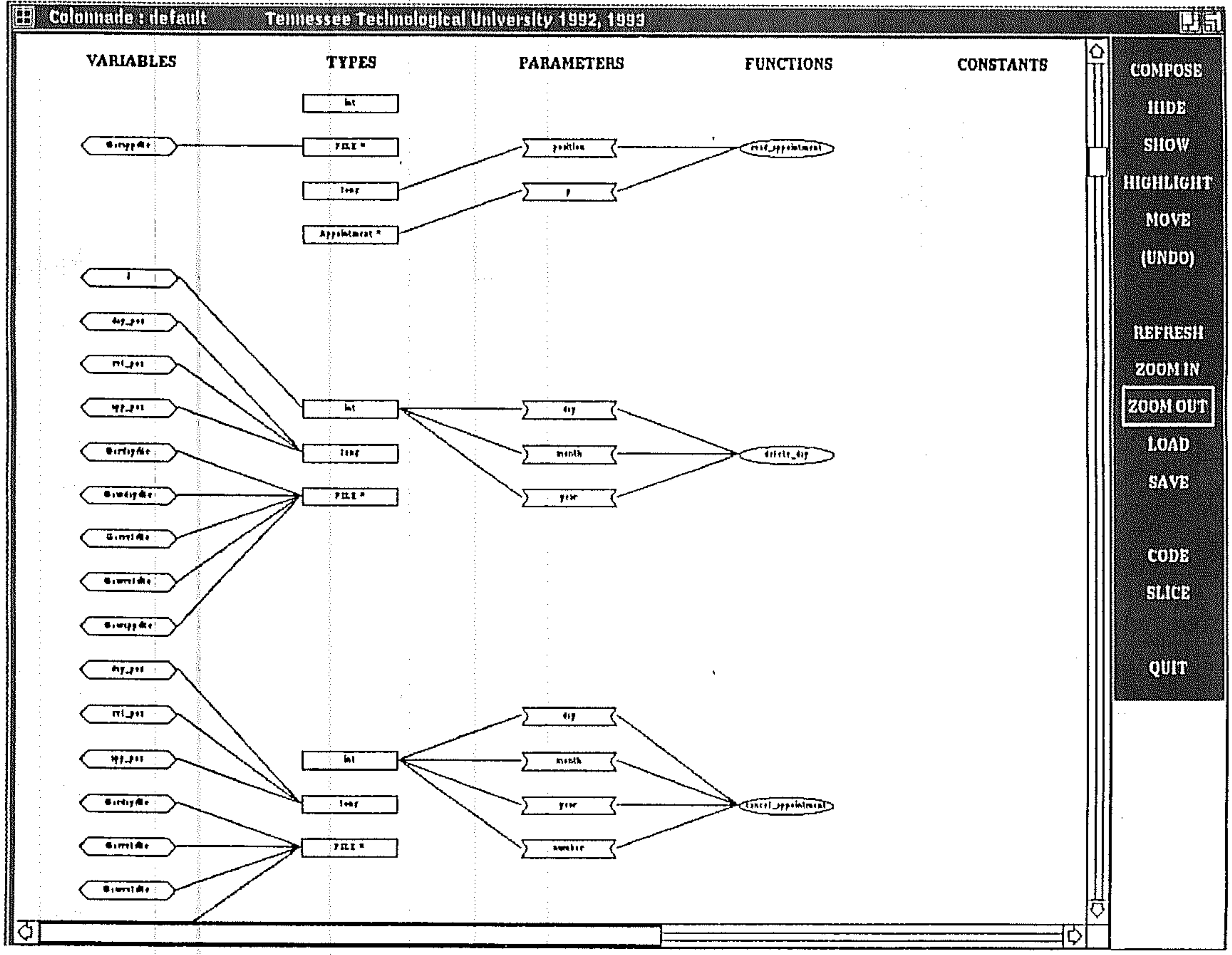


Figure 4: A colonnade display of data-flow program dependencies.



last column displays all the constants used in the program. The connecting lines represent the relations between these entities. For instance, the function *delete_day* has three parameters *day, month* and *year* all of type *int*. The same function uses also several local and global variables of various types and it has no constants. One limitation of the colonnade is that it cannot display relationships between non-consecutive columns. For instance in figure 4, the fact that the function *read_appointment* returns a value of type *int* cannot be displayed until the columns FUNCTIONS and TYPES become adjacent (the *move* operation overcomes this limitation and it is discussed in a later section). The colonnade produces *crossing-free* and aesthetically pleasing layouts and complements the call-graph. As a result, software engineers can have multiple graphical views of the program architecture (i.e. control and data-flow).

B. Abstraction Mechanisms

The CARE environment encompasses abstraction mechanisms in order to manage the complexity of large program displays and to allow for the organization of multiple graphical and textual *views*. In CARE, program dependencies can be displayed by *monolithic* or *multiple-view* representations. A *monolithic* view entails the complete code or global graphical representations (i.e. complete call-graph or colonnade) of existing programs. The *multiple-view* representations consist of graphical or textual *slices* which are fragments of the code, colonnade or the call-graph and allow for multiple organization of program dependencies. Moreover, graphical slices provide a platform for flexible transformation mechanisms and reduce the complexity of large graphs. Figure 5 contains a C function and its data-flow graphical slice. The *zoom, move, compose, hide , refresh* and *highlight* are graphical operations available by the CARE environment. The *zoom* operation is used to enlarge or reduce the size of the graphical display. The *hide* operation removes a selection from the display, as well as all the relationships associated with it. The *compose* operation allows the user to abstract parts of the display in order to manage its complexity. The *move* operation permits the user to relocate graphical entities in different positions on the display. The *highlight* operation enables the user to prominently display parts of the display. Moreover, two editing operations are available in CARE; the *code* and *delete* operations which allow the user to modify the contents of the source code and the database respectively. The *code* operation enables the user to have direct access to the code using a text editor (e.g. emacs, vi), whereas the *delete* operation erases a selected entity from the database. Additional operations allow the user to *save* or *load* specific hierarchical or colonnade layouts. Finally, in order to improve the readability of the layouts produced by CARE modifications of the algorithms found in [5] and [6] are used.

C. Transformation Tools

The transformation and slicing mechanisms available by CARE are shown in figure 6. The textual representation of the code can be transformed into either the hierarchical representation of the control-flow or the colonnade representation of the data-flow. In addition, colonnade graphs can be transformed into call-graphs and vice versa. Graphical or textual slices can also be created from these representations. Control-flow slices can be derived from the colonnade and data-flow slices can be generated from the call-graph. For instance, in CARE, the user can point and click on a specific function on the call-graph and get a pop-up slice of the data-flow (i.e. global and local variables, constants and parameters used and their types) for that function. Similarly, a control-flow slice can be created for a selected function in the colonnade. The control-flow slice contains the *call* and *called-by* relationships of the selected function.

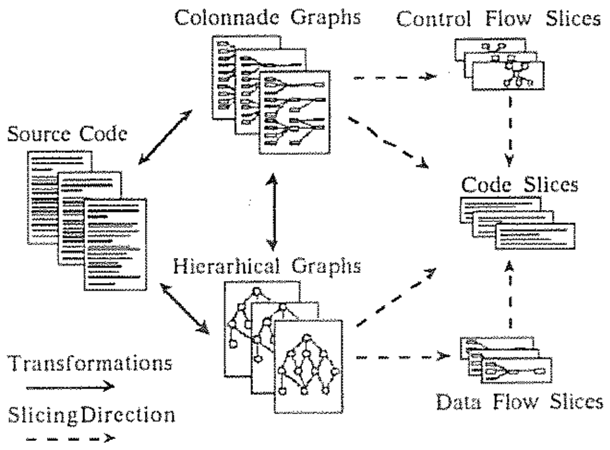


Figure 6 : The transformation tools and slicing mechanisms available by CARE

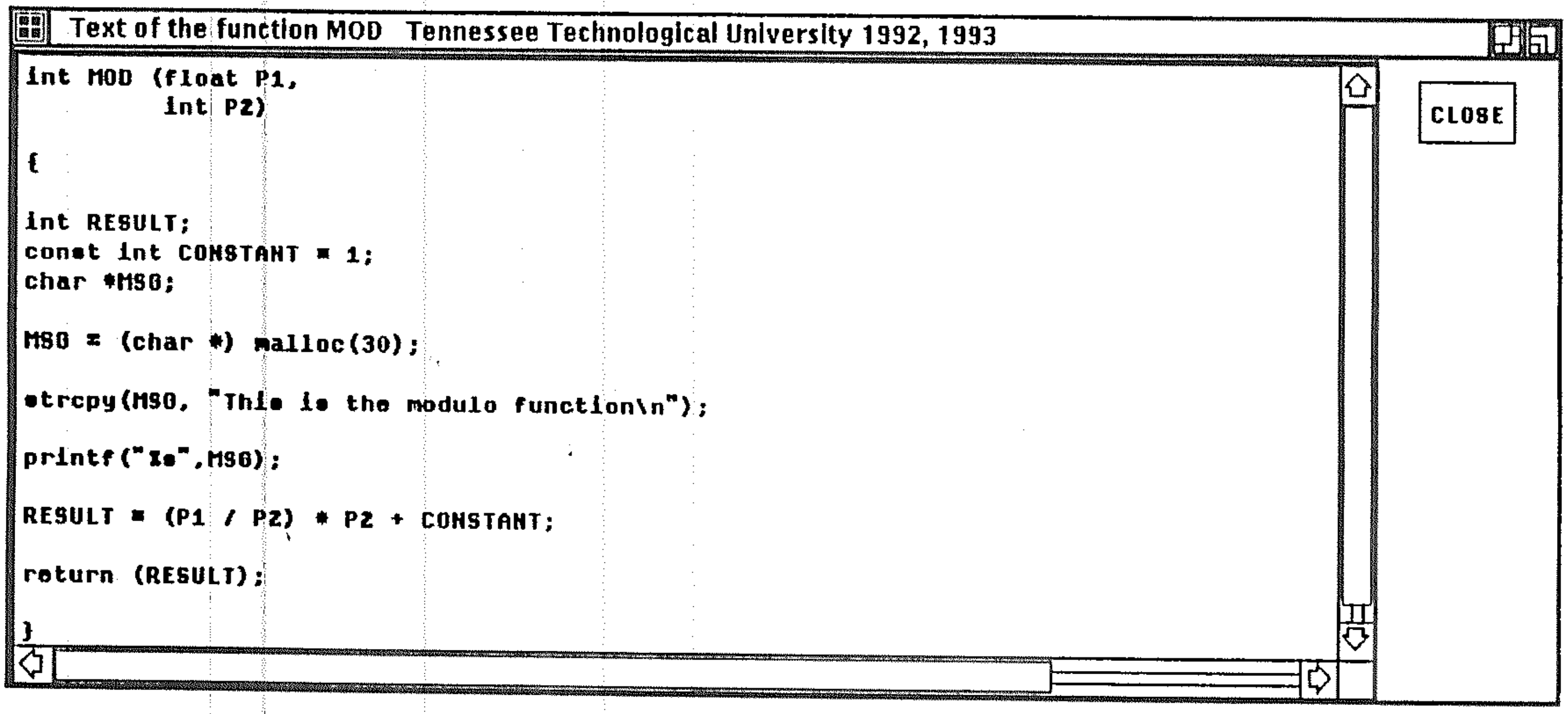
IV. An Experimental Study

In this section, an experimental study with the CARE environment is described. In this study, we performed an experiment in order to verify our hypothesis with respect to improving the productivity and quality of changes made during the maintenance of C programs. Moreover, results from an empirical evaluation of the user interface of CARE are included in this section.

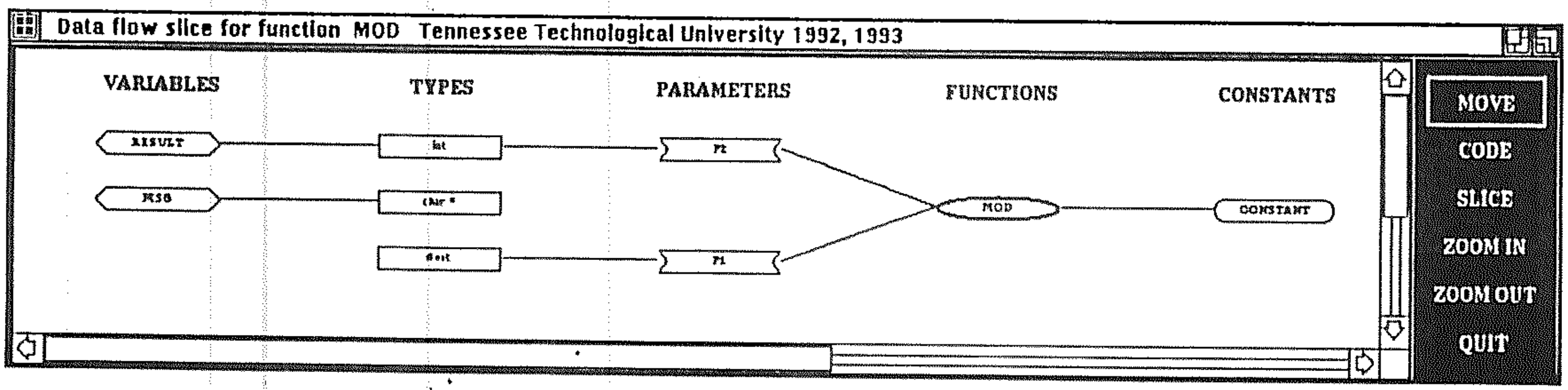
A. Experimental Framework

An experiment with forty computer science graduating seniors was performed during this study. The graduates were divided into groups A and B, each having 20 members and they were asked to complete a maintenance task. The experiment was performed in two phases as shown in figure 7. During the first phase, both groups were given an

Figure 5: A C function and its data-flow slice.


Text of the function MOD   Tennessee Technological University 1992, 1993
CLOSE

```
int MOD (float P1,
         int P2)

{

int RESULT;
const int CONSTANT = 1;
char *MSG;

MSG = (char *) malloc(30);

strcpy(MSG, "This is the modulo function\n");

printf("%s",MSG);

RESULT = (P1 / P2) * P2 + CONSTANT;

return (RESULT);

}
```



existing program which automates the process of maintaining a database of appointments, notes and reminders on a weekly calendar (program X). This program was written using the C programming language and utilized the X window manager and graphics routines in order to display the days of the week on the screen. The graduates were asked to modify the weekly display from seven days to six (i.e. Sunday was to be removed from the display).

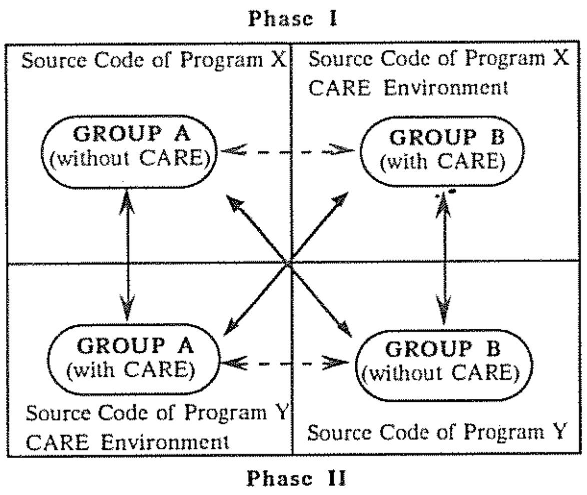


Figure 7 : Experimental Framework

During the first phase, group A was given only the source code of the program to be modified while group B was given the source code plus the CARE environment. During the second phase of the experiment, both groups were given a stand-alone part of the source code of CARE itself (program Y). Then, they were asked to increase the scaling factor of the *zoom* operation by one unit. During the second phase, group A was given the CARE environment in addition to the source code of program Y whereas, group B was given only the source code of program Y. There were no comments included within any of these programs and they both used Xwindow routines. Also, during the experiment the graduates didn't know anything about the internal structure of the programs to be modified. Both programs X and Y were of similar size and complexity. The results from the experimental study supported this assumption. The performance metric for this experiment was the time needed by the graduates to complete the maintenance task. The independent variables in our experiment were the grouping of the graduates (i.e. group A and B), the two phases of the experiment (i.e. phase I and II) and finally, the availability of the CARE environment (i.e. with or without CARE). In addition, during the experiment we followed an empirical evaluation technique known as *thinking aloud* method [7] in which the graduates were asked to verbalize their thoughts while working and their discussions were recorded. That method gave us a direct understanding of their pattern of thinking with respect to understanding the architecture (i.e. control and data-flow) of the program. Moreover, this approach helped us to identify essential misconceptions related to the user interface as well as the users' interpretation of the graphical displays and operations available in CARE. A separate questionnaire was also completed by the graduates at the end of the maintenance task.

From this study, we have gathered qualitative and quantitative information. The qualitative data contains results from the empirical evaluation of the user interface and functionality of the CARE environment. The quantitative results reflect the measurements of the time needed to perform the maintenance tasks. Next, representative qualitative and quantitative results are discussed in separate sections. Finally, a summary of the overall results of this study is presented in the following section.

B. Analysis of the Quantitative Results

In this section, the quantitative results from the experiment are discussed. The overall time needed to perform the maintenance task was measured during the experiment. During the analysis of variances, we correlated the time needed for the modifications (i.e response variable) versus three independent variables namely the phase (I or II), the availability of the CARE environment (with or without) and the group (A or B). Two statistically significant results were found during the analysis of variances. The average time needed to complete the maintenance task was found to be less when the CARE environment was available than the average time needed when CARE was not available to the users. In particular, the mean time when CARE was used was found 101.87 minutes and the mean time without CARE was 162.91 minutes. The standard deviation was found to be 17.48 for both cases (p value was less than 0.0175). The second statistically significant result was found to be the difference between groups A and B. The mean time of group A was found to be 164.58 minutes and the mean for group B was 100.20 minutes. The standard deviation was found to be 16.57 for both cases (p value was less than 0.0125). In other words, a significant difference was found with respect to the *direction* of the experiment. Namely, less average time was needed when the tool was given during phase I and then removed in phase II rather than the opposite scenario. When the users were given the CARE environment from the beginning it guided their thoughts with respect to program understanding and therefore helped them to locate the points to be modified in the program. In addition, during the second phase (after CARE was removed) they were still able to find the points to be modified faster due to the fact that they already had gained some experience using CARE. Finally, there was no statistically significant results found with respect to phases I and II. Namely, there was no difference of the performance found between using programs X and Y for the maintenance task. This supports our assumption that both programs were of similar complexity. In addition, during the experiment the number of statements modified (i.e. added, removed or changed) in the source code were counted. The average number of statements modified was 19.9 when CARE was not available and 12.5 statements when CARE was used by the graduates. It appears that CARE helped the users make better quality changes during the maintenance task.

C. Analysis of the Qualitative Results

In this section, the results related to the user interface and functionality of the CARE environment are discussed. Figure 8 depicts ratings of the frequency of usage for each operation available by the call-graph editor. The most frequently used operations available by the call-graph editor were the *code* (96% of the students used it often) followed by *zoom (88%)* and *highlight* (49%). The *zoom* operation allowed them to focus on a part of the call-graph and therefore helped in locating the points to be modified. The *highlight* feature was found to be useful in understanding the data and control-flow of the program. The *code* option enabled the users to have direct access to the body of any function from the call-graph. The least frequently operations were the *compose* , *undo* , *refresh* and *delete* operations. The *compose* option would be more useful when the users have to work with larger and more complex graphs. However, during the experiment, the call-graphs were fairly small and therefore composing any function was not a demanding task. The *delete* operation removes information from the graph and directly affects the database. Since the main objective of the CARE environment is to facilitate program understanding by graphically displaying program dependencies, any permanent deletion of the graphical information on the screen didn't seem to be of a great interest. Figure 9 shows how often each operation available by the colonnade editor was used during the experiment. The most frequently used operations were *code* and *zoom* and the least used ones were the *delete, undo* and *refresh* operations. For both the colonnade and the call-graph editors, the *save* and *load* operations were not used very frequently. During the experiment, it appears that there was no need to save a particular call-graph or colonnade. Again, one possible reason for that can be the simplicity and small size of the graphs. In addition, the ratings of the operations available by the slice editors (control-flow and data-flow) agree with those of the call-graph and colonnade editors. Despite the fact that graphical or textual slices were new concepts to the graduates, they were found easy to understand and use by the majority of them. An important observation from the overall ratings of the user interface of CARE is that almost all (98%) of the users indicated that CARE was not difficult to learn and the majority (82%) spent little time learning it. Moreover, all users found the main window and the hierarchical call-graph easy to understand and use. Despite the fact that the colonnade was a novel concept to the users, 82% found it easy to understand and 65% easy to use. In addition, the majority of the users found the graphical shapes used to represent information appropriate (95%) and the fonts easy to read (80%). Finally, many users (82%) suggested that colors would be a valuable addition to the user interface of CARE.

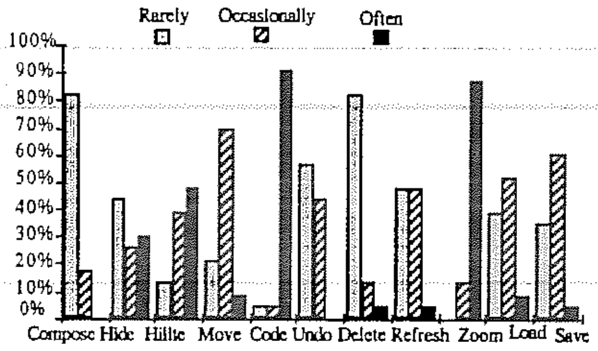


Figure 8 : Frequency of usage for each operation of the call-graph editor.

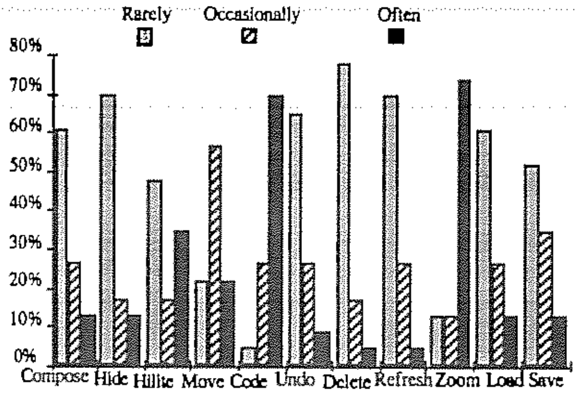


Figure 9 : Frequency of usage for each operation of the colonnade editor.

D. Summary of the Results

In summary, it was found that the users were more productive (by 61.2%) and they produced better quality changes (by 14.7%) during the maintenance tasks when using the CARE environment. In addition, it was observed that the graduates were able to understand the control-flow (i.e. call-graph) of the program using the hierarchical display almost with no problem. Also, using the colonnade editor, they could trace the data-flow (i.e. parameter-passing, variables, types etc) of the given program. Most ratings (82%) of the user interface indicated that the graduates needed little time to become familiar with CARE as a software engineering tool and the majority (98%) acknowledged the simplicity of its user interface. Additionally, the complexity of the colonnade and the hierarchical displays were conveniently managed using the graphical operations and abstraction mechanisms provided by CARE (e.g. zoom, compose, hide etc). In particular, the *zoom, code* and *highlight* operations were frequently used. The *move* operation allowed the graduates to see the relationships among different entities (e.g. type of parameters and functions).

V. Lessons Learned

There are several important lessons we learned during this study which helped us improve the functionality and user interface of the CARE environment. In addition, these lessons assisted us in making some points with respect to program understanding issues.

As mentioned in earlier sections, the need to access the code from the CARE environment was compelling. This is the justification of our *code* operation. On the contrary, the low ratings of the *delete* operation with respect to its usefulness as well as being the least frequently used operation lead us to the decision to remove this operation and disallow any permanent modifications to the database of program dependencies. This observation helped to narrow down the functionality of our tool to be an environment for program understanding. In addition, the need of an *open* architecture for CARE was indicated namely the desire to use a favorite editor or a tool that the users were familiar with was strongly stated. The slicing mechanisms and browsing tools in the database were justified by the users' responses. The most frequent comments about the CARE environment were related to the simplicity of its user interface. Moreover, the graduates had positive comments about the colonnade editor and found it a promising way to represent program dependencies. They also acknowledged the usefulness of the call-graph editor. Although graduates preferred to see the colonnade graphical representation to be separate from the call-graph, they also appreciated the combination of the two models as a promising vehicle towards program understanding. It was also observed that all graduates (100%) believed that the code visualization tools available by CARE enabled them to understand the existing programs easier. During the experiment, the maintenance tasks given to the graduates were considered moderately difficult (70%) or even easy (30%) to do when CARE was available.

A unanimous desire for software engineering tools for program understanding was distinct during and after the experiment. The tendency to understand the whole program before making any changes was rare (almost nonexistent). An incremental, *as-needed* approach was taken during the maintenance task almost in all cases. In addition, using the CARE environment the quality of changes performed on existing C programs appeared to be improving and the time needed to accomplish such changes was reduced. Finally, it was observed that when the graduates were given the CARE environment which facilitated program understanding, they were guided through the process of understanding the control-flow and data-flow of the existing programs and thus improved their performance.

VI. History of CARE

CARE has been an on going research project at the computer science department of Tennessee Technological University since summer 1990 and includes five research students and a faculty member. It is partially supported by the Faculty Research grants #9111 and #9206 from Tennessee Technological University. CARE runs on VaxStations with the Ultrix operating system and it has been implemented in ansi C. It utilizes the X window manager and graphics utilities for displaying program dependencies. The code analyzer has been implemented using *lex* , a lexical analyzer available in the Ultrix environment. In December 1992, a prototype of the CARE environment was demonstrated at the 5th ACM SIGSOFT Symposium on Software Development Environments in Virginia, USA [4].

References

[1] Lerner, M., *A Standard Approach to the Process of Re-engineering Long-lived Systems*, CASE trends, Software Productivity Group, Inc., July/August 1991.
[2] Linos, P., *Graphical Support for Visual Environments*, IAMM, April 23-26, 1990, Detroit, MI, pp. 642-647.
[3] Linos, P., Aubet, P., Dumas, L., *Understanding the Structure of C Programs,* Research Report, Computer Science Department, Tennessee Technological University, December, 1991.
[4] Linos, P., Helleboid, Y., Lejeune P., Tulula, P., *A Software Tool for Understanding and Re-engineering C Programs,* 5th ACM SIGSOFT Symposium on Software Development Environments, Tool Demonstration, Virginia, Dec. 1992.
[5] Linos, P., Rajlich, V., Korel, B., *Layout Heuristics for Graphical Representations of Programs,* 1991 IEEE Conference on Systems, Mans, and Cybernetics, University of Virginia, Charlottesville, Virginia, October 13-16, 1991, pp. 1127-1132.
[6] Messinger, E., Rowe, L., Henry, R., *A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs*, IEEE Transactions on SMC, Vol SMC-21, No. 1, Jan/Feb 1991.
[7] Nielsen J., *The Usability Engineering Life-Cycle,* IEEE Computer, March 1992, pp. 12-22.
[8] Oman, P., *Maintenance Tools,* IEEE Software, May 1990, pp. 59-65.
[9] Rajlich, V., Damaskinos, N., Linos, P., Khorshid, W., *VIFOR : A Tool for Software Maintenance*, Software-Practice and Experience, January 1990, pp. 67-77.
[10] Ramamoorthy V., Chen, F., Nishimoto M., *The C Information Abstraction System,* IEEE Trans on Software Engineering, vol. 16(3), March 1990, pp. 325-334.
[11] Robson, D., Bennett, K., Cornelius, B., Murno, M., *Approaches to Program Comprehension,* J. Systems Software, 1991, vol. 14, pp. 79-84.
[12] Wilde, N., *Understanding Program Dependencies*, Software Engineering Institute, CM-26, August 1990.

Session C:
Experience Report Session

Chair: Norman Wilde

Use of a Program Understanding Taxonomy at Hewlett-Packard

Alan Padula
padula@hpcea.ce.hp.com

Hewlett-Packard Company, Corporate Engineering
Palo Alto, Ca. 94303

Abstract

This report summarizes the use of a Program Understanding taxonomy that was developed at Hewlett-Packard. The primary use of the taxonomy has been in the creation of a company internal document called the Software Tools Report. The Software Tools Report is a selection and evaluation guide to software tools that addresses key company software engineering areas which include Program Understanding. A description of the Report, how it was created, and how it is used is the subject of this paper.

1 Introduction

Hewlett-Packard (HP) Engineers spend an estimated $200 million a year just reading code! Although that is an unofficial, broad estimate, it does capture the magnitude of investment HP makes in understanding (not changing) the software it must maintain. Industry data substantiates that half of the overall effort of making changes is spent just understanding the code [1]. When HP Divisions are presented with this data, most engineers and managers nod in acknowledgment while the few skeptics are generally questioning whether 50% is too low!

As with the rest of industry, Corrective, Perfective, and Adaptive Maintenance are all large parts of the company's software budget [2]. Fewer and fewer software systems are developed from scratch. Source code is constantly picked up and leveraged from other pre-existing systems to quickly make new competitive products. Source code is bought from third parties for modification. Responsibility for maintaining software systems is transferred from one Division to another. Customers report sporadic problems with software that has been around for years and years requiring bug fixes. New hardware platforms are constantly developed that existing software must be ported to. In all of these cases, it is not unusual for the original programmers to be long gone, for the code to be unstructured due to a long history of fixes and enhancements, for the code not to match the design or documentation, and so on.

For all of these reasons and more, Program Understanding at HP has been identified as a key internal software engineering area to focus on. A group called the Software Initiative (SWI) was formed at Corporate Engineering (CE) to help develop and deploy best software engineering processes and technologies to the Company's Divisions. Program Understanding technology is a part of that effort.

2 Why a Program Understanding taxonomy?

The need for a good Program Understanding definition and taxonomy quickly became apparent. It was needed for performing simple phone consulting, establishing long term consulting, conducting needs analysis, and creating most other information deliverables regarding Program Understanding including:

- **The Software Tools Report** - a selection & tools evaluation guide (see below),
- **Tool Information Packets** - bundled vendor & internal data on specific tools,
- **Tutorials** - educational materials directed at different audience levels,
- **Usage Guidelines** - a "how to" process guide for Program Understanding tasks.

3 Program Understanding taxonomy development

The taxonomy was developed and customized to be an *appropriate fit for HP's software development environment*. This last phrase is emphasized since the frame-of-reference was CE's customer (i.e. HP's software Divisions) and their need for the taxonomy to a large extent dictated what was included. The purpose was <u>not</u> to produce the most comprehensive list and description of all

0-8186-4042-1/93 $03.00 © 1993 IEEE

Program Understanding functionality ever made. A general research project was not the point; practical usability of the feature set list inside the HP industrial environment to help improve productivity and quality was! Whereas there are some features listed in the taxonomy that are not available in the HP environment, the focus is on what is commercially available instead of what would be "nice-to-have". Purchasing tools is generally preferable as it is cheaper and lower risk than developing custom in-house tools that may have a high maintenance cost associated with them. More attention was paid to Program Understanding functionality available for C, FORTRAN, and PASCAL-based software than to special Object-Oriented language ones due to the corresponding language mix in HP's maintenance backlog. With the emphasis being on system's software, little regard was given to functionality that helps in understanding Management Information Systems or database type applications.

The taxonomy was developed from many sources including books, research articles, monthly magazine review & ratings, conference & workshop proceedings, vendor brochures, and meetings with vendors and researchers [3-5]. Interviews were conducted with HP engineers to find out what commercial and internal tools they used on a daily basis, what enhancements they wanted to see added to them, and what their list of information needs were. As the purpose of this experience report is to describe how the taxonomy and the subsequent Software Tools Report is being used, the lengthy text of the taxonomy has not been included. The detailed definition of the taxonomy is the topic of another paper. Inquiries regarding the taxonomy may be addressed to the author.

Of all the deliverables that the taxonomy influenced, by far, the most important is the Software Tools Report...

4 Software Tools Report

The Software Tools Report is an internal Hewlett-Packard evaluation and selection guide to software engineering tools. The Report provides a list and description of the software tools that address key internal HP software engineering areas such as Program Understanding, Software Configuration Management, Process Definition, et cetera. It does this in part by mapping the features that aid in different software engineering areas to selected software tools. It provides both an objective, low-level analysis of features as well as a more subjective rating of how well (from Better to Worse) those features are actually provided.

5 Evaluation of tools for the Software Tools Report

Evaluation of the tools was initially done through analysis of commercial brochures and documentation. This created a first pass. This was followed by hands-on-work, which sometimes revealed that actual operation of a tool was a little different from what was actually advertised. This hands-on-work provided direct verification of the functionality and at the same time, gave a feel for the usability, reliability, and other attributes of the product.

Direct Division feedback was heartily solicited and required from engineers who used the product or who were in the process of evaluating it themselves. This is an absolutely essential ingredient in order to maximize credibility of the Software Tools Report with the HP engineering community. A success story or simple endorsement by an engineering cohort is magic when introducing change into an organization. (Our experience is that introducing change into an organization is by the far the most difficult part of aiding Divisions in the adoption of new software engineering tools and processes.)

Every effort was made to validate with the vendors that the Software Tools Report accurately depicted their tool. This usually amounted to sitting down with them and doing a line-by-line feature review of the existing taxonomy relative to their tool. If their product provided functionality that they felt was Program Understanding technology, we considered it for inclusion in the Report. Again, the taxonomy of Program Understanding is not intended to be exhaustive, just practical. Vendors are left responsible for notifying HP when new features are added to their tool which require an update to the Report. This works well as the vendors have a vested interest in keeping their product up to date and saves CE the trouble of having to actively monitor the vendors for new features.

When a change is made to the Report, an electronic notification is sent to HP subscribers advising them. They then may request the latest update of the Report. One of CE's primary goals is to produce a deliverable that does not become dated "Shelfware"!

6 Software Tools Report layout

Figure 1 shows the basic outline of the Software Tools Report. It combines several pages of the Software Tools Report onto one to help give the reader a better overview of how one navigates through the Report. (For legal reasons, the names of the actual tools could not be used in this paper.)

There are four levels (0-3) in the Software Tools Report used to describe tools:

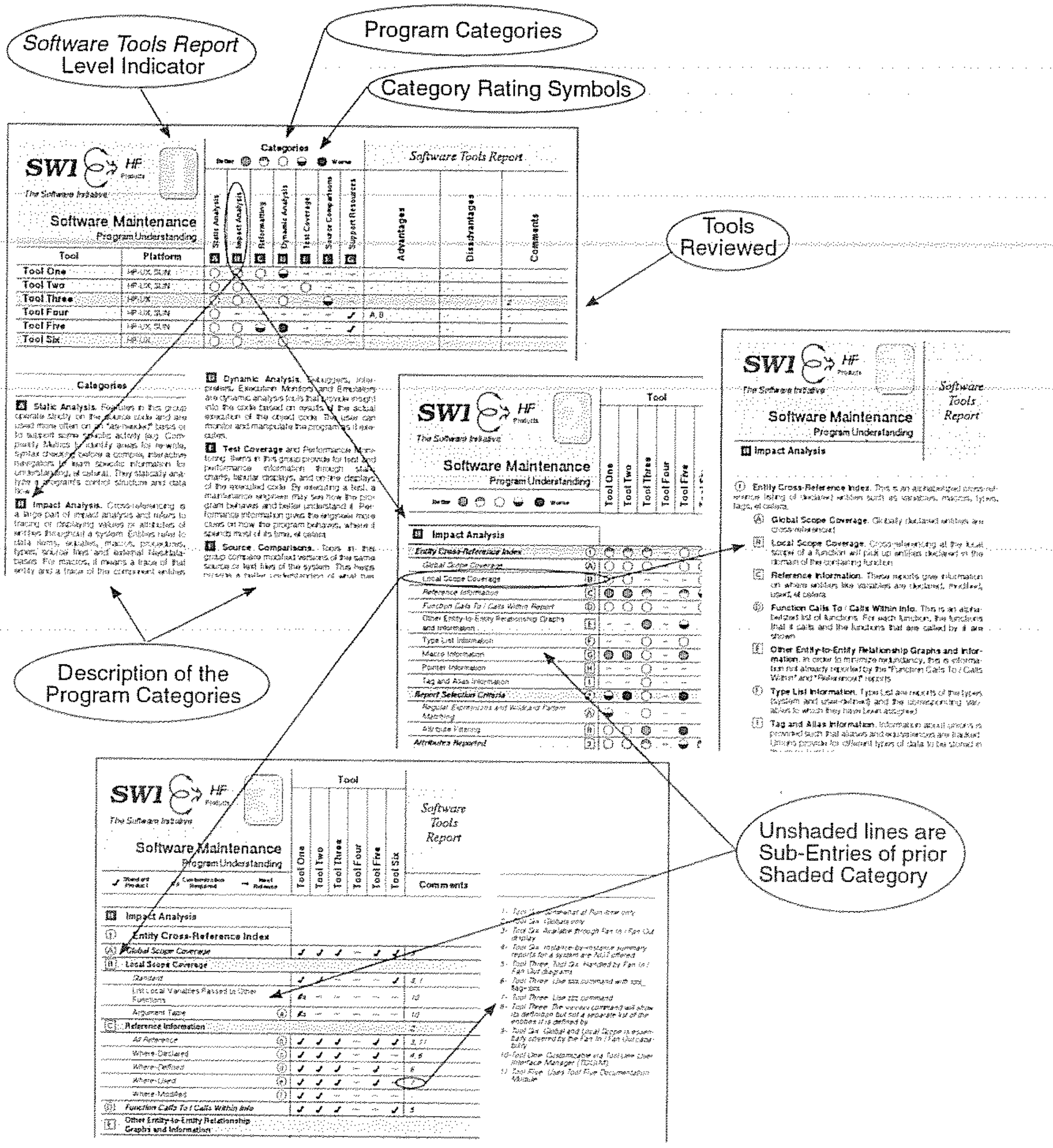


Figure 1.

6.1 Level 0

Level 0 (not shown in Figure 1) is used for identifying basic product environment information about the tools. This includes the required software environment, hardware platform and configuration, the user interface, the available documentation, and the price. It also identifies tools that may be used across different software engineering areas. Tools that provide for Program Understanding may cut across other software engineering technologies such as Software Configuration Management and Reuse.

6.2 Level 1

Level 1 provides the highest view of the functionality of a tool relative to the software engineering area that it is being evaluated on. For HP's purposes, Program Understanding is broken down into seven areas.

- Impact Analysis
- Static Analysis
- Reformatting
- Dynamic Analysis
- Test Coverage
- Source Comparators
- Support Resources

Each tool is rated from Better to Worse in those categories (see Rating below.) HP engineers needing help in any one of those major categories quickly see which tools are the top candidates to consider. They then drop down to the second and third levels to do a more detailed analysis. The use of the Report is intended to be used only as a guide. The needs of individual engineering projects are much too variable to come to any singular recommendation about which tool is actually "best" for everyone.

6.3 Level 2

Level 2 provides the next tier of functionality detail. It too is rated from Better to Worse. An example is Level 1 Impact Analysis which expands at Level 2 as:

- Impact Analysis
  - Entity Cross-Reference Index
    - Global Scope Coverage
    - Local Scope Coverage
    - Reference Information
    - Function Calls To/Calls Within Report

Note that the definition of these terms is provided in the Software Tools Report close to where the terms are introduced (see Figure 1.)

6.4 Level 3

Level 3 contains the most detail and is a further expansion of Level 2. For example, under Impact Analysis & Entity Cross-Reference Index, Local Scope Coverage expands to:

- Impact Analysis
  - Entity Cross-Reference Index
    - Global Scope Coverage
    - Local Scope Coverage
      - Standard
      - List Local Vars Passed to Other Functions
      - Argument Table

At this level, there is no rating from Better to Worse which is subject to interpretation. The objective is to provide a Yes or No answer i.e. the tool either provides the functionality or it does not. A lot of use is made of Level 3 by HP Divisions who are independently doing their own needs assessment and tool evaluation. Only they know what is truly important to them and this gives them the objective data to fairly rate the tools for their own environment. In conjunction with CE consulting, it is used primarily as a reference document. (Note: The initial development for any tool evaluation to be included in the Software Tools Report begins with an analysis that results in Level 3.)

7 Rating

Evaluation criteria was developed to enable assignment of a rating from Better to Worse for Levels 1 and 2. Basically, the Better to Worse symbols each map into a percentage range that has the population of the tools evaluated falling into the shape of a modified bell curve. That is, the majority of the tools do not fall into the Better or Worse extremes, but rather in the middle. Each tool feature is assigned (weighted) a value depending on how important it is thought to be to that particular Program Understanding category. High to low point ranges are then calculated for each symbol in each feature category based on the maximum number of total possible weighted points and the percentage ranges that the symbols represent. Point values are then calculated for the individual tools in each feature category based on their feature set. From this, a symbol indicating the tool's rating is assigned.

The fact that the reviewers may (and probably do) feel certain features are more important in their environment is perfectly all right, expected, and desired. The most important features with the highest multiplier factors are printed in *italics* so the reviewer clearly knows what is thought to be key to that category. As they are provided with all of the objective data and the evaluation criteria and algorithm, they can calculate their own ratings. Meanwhile, reviewers who just want a first blush rating, can get an educated guess of what might be most useful. The Software Tools Report serves only as a guide and is absolutely NOT intended to be a final word for everybody.

8 Legend

A variety of symbols are used in the document. Some of the more interesting ones can be seen in Figure 2.

It may seem logical to think that either a tool has a feature or it does not, but unfortunately even technical evaluations are not always that simple. During our evaluations, we found that we had different assessments as to whether a tool had a feature or not. Oftentimes, the same functionality can be derived from a tool with one or more extra user steps. In the cases where the data is in a suitable form for accessibility and the functionality to easily manipulate the data into useful information is provided, the customization symbol (see Figure 2) is used. Whenever a feature has one of these instead of a check mark, an associated comment is usually attached with further explanation. The Divisions and vendors validated that this is a fair way to go. It properly represents vendors who have one-step push button access to the information (they get a check mark) and gives appropriate credit to the vendors who basically have the functionality through a couple of extra user steps.

italic text	Indicates the entry is part of the minimal criteria.
—	Entry / function does not apply or is missing from the tool.
✓	Level 3 indicator: entry / function is available as part of the standard off-the-shelf offering.
✍	Level 3 indicator: entry / function is available only after customization performed by the user or vendor.
→	Level 3 indicator: entry / function will be available as part of the standard product at the next release.
◍	
◓	
○	Level 1 and Level 2 Category rating symbols from Better to Worse (top to bottom) with the 'o' symbol indicating the minimal criteria has been met.
◒	
●	

Figure 2.

Another hurdle overcome was related to "futures". Many times the vendor would say they have the feature in their next release. It seemed unfair to them to ignore their input but at the same time, we did not want to tell the Divisions that the functionality was there when it was not. We finally settled on a symbol that indicates it is planned to come out next release. Again, this is fair to the vendor and better for our Divisions as they can then plan their purchase based on whatever the latest information is. (Oftentimes, the purchase of a tool is not immediate and occurs a few months later anyway.)

9 Conclusion

Program Understanding is recognized to be a key technology in the internal software engineering processes at Hewlett-Packard. One of the first tasks undertaken to promote Program Understanding in the company was to produce a taxonomy of Program Understanding features. This is used for a variety of purposes including simple phone consulting, long term consulting, needs analysis, tutorial development, Tool Information Packets, Usage Guidelines, the Software Tools Report, and most of the other information deliverables regarding Program Understanding. The most important of these is the Software Tools Report. It provides feature set comparison of a variety of program understanding tools. It greatly aids in the evaluation, selection, spreading and adoption of Program Understanding technology in the company.

References

[1] T. Corbi, *IBM Systems Journal*, Vol. 28, No. 2, p 296, (1989)

[2] E. Swanson & B. Lientz, *Software Maintenance Management*, Addison-Wesley Publishing Co., Reading, Ma. (1980)

[3] C. Sittenauer, D. Dyer, G. Daich, *Software Re-engineering Report*, Software Technical Support Center, Hill AFB, Utah (1991)

[4] H. Holbrook & S. Thebaut, A Survey of Software Maintenance Tools that Enhance Program Understanding, Software Engineering Research Center, Computer & Information Systems Department, University of Florida, Gainesville

[5] N. Zvegintzov (Ed.), *Software Maintenance News*, Vol. 9, #8, p 23-27, Los Altos, Ca (Aug 1991)

Acknowledgments

Special thanks to Ron Becker for the actual construction and a very significant contribution to the format that makes up the Software Tools Report as well as for the creation of the figures used in this report.

Recovering User Interface Specifications for Porting Transaction Processing Applications

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

Abstract

The Reverse Engineering group at EDS Research has developed software tools to mechanically assist in reengineering transaction processing applications. We are applying the software tools to assist in converting a very large minicomputer application written in COBOL to run under CICS on an IBM mainframe. The two platforms provide very different user interfaces and computational 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 demand that any successful mechanical conversion strategy employ sophisticated feature extraction and restructuring techniques. We describe the problem of recovering the user interface specification and using the recovered specification to create the appropriate user interface in the target environment. Techniques such as data flow analysis and other formal analysis techniques appear to be too weak to guide the conversion, and that a priori programming knowledge must be encoded and applied to obtain a successful conversion.

1. Introduction

We first describe the problem of converting large minicomputer application to run under CICS on an IBM mainframe. We discuss our approaches to translation and to the problem of creating new screens for the new environment.

We next describe a specific task within the larger project, the problem of recognizing and restructuring block of code that read in and validate values entered by the user. Our initial approach to this recognition task used a hierarchical collection of recognition rules. The recognition task also tried to find the largest structure in the code that fit a certain pattern using a fixed-point computation. This approach worked on some programs, but ultimately failed because of the complexity and variability of the COBOL code. Instead of extending this approach, we instead encoded a priori knowledge of programming for transaction processing applications as heuristic rules. This later approach has proved very successful in producing usable translations for a wide variety of programs.

2. Background

EDS Reverse Engineering Group

The Reverse Engineering group at EDS Research is developing tools with the goal of extracting "high level" descriptions of existing transaction processing applications from the source code. Examples of high level descriptions include data models, data integrity constraints, user interface specifications, and standard data processing paradigms such as "update a file" or "sum a column". These high level descriptions are used in a variety of tasks, including re-engineering applications for new computing environments.

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. Other Prolog programs compute a control flow graph, extensive cross referencing and data structure information, and 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. We have done preliminary work on a simple symbolic evaluator and a weakest precondition generator.

Project Description

EDS has a contract with a federal agency to consolidate many applications onto an IBM mainframe. The consolidation will save a large amount of money and provide improved service to the applications' users. Several large applications comprising over two million lines of COBOL must be moved to the mainframe under CICS. The contract requires that :

- the converted code be functionally equivalent, but
- the converted code not emulate functionality not inherently available in the target environment, and
- the user interface shall not have additional complexity.

Within these requirements the conversion process should minimize conversion costs and maximize the efficiency of the converted code. For best performance it is important to use "pseudo-conversational" programming on the mainframe. The goal of the project described here is to provide automated tools to help the engineers who are doing the conversion. In the following sections we describe the main differences between the source and target environments, the difficulties these differences impose on the conversion, and the approaches we have used. The translator as it exists today automates a great deal of the conversion process.

3. Source and Target Environments

Minicomputer 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. A typical program will display a formatted screen, such as the one depicted in Figure 1, with a single display operation. The definition of the screen format includes fields in which the user can type values.

The program can then place the cursor at any field on the screen and accept values typed by the user. The user can type in a value followed by a RETURN, or can simply type return. Instead of typing a value in a field or pressing RETURN, the user can press the ESCAPE key. Most minicomputer programs define the ESCAPE key to allow the user to return to a previous field.

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.

Mainframe User Interface

On the mainframe using CICS, in contrast, the program interacts with the user a full screen at a time. The program can display a formatted screen that contains character strings, variable data, and control characters that define fields in which the user can type values. Such a formatted screen is called a *map*. The operation of displaying a map is called a *send*. The operation of reading the data typed by the user is called a *receive*.

When the program sends a map, the user sees the formatted screen. The user then types in all the values in all the fields. The user can use the TAB key to move the cursor from one field to another. When the user is satisfied with all the values on the screen, the user then transmits the entire screen contents to the program. The program receives the map, 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.

Minicomputer 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 2 shows a fragment of minicomputer COBOL code that accepts user input. The Appendix has a brief explanation of the semantics of the minicomputer COBOL verbs.

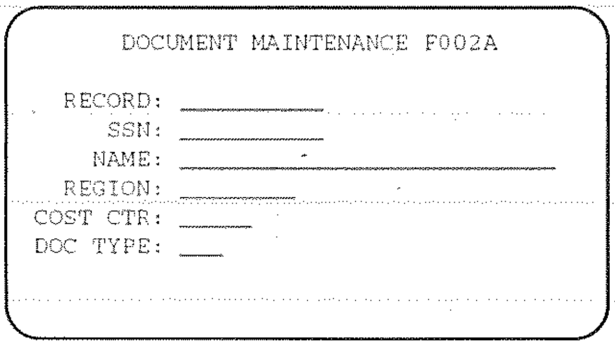

```
          DOCUMENT MAINTENANCE F002A

   RECORD: ________
      SSN: ________
     NAME: ________________________
   REGION: ______
 COST CTR: ____
 DOC TYPE: __
```

Figure 1 Formatted screen

Mainframe Control Structure

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 all program 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 3 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

```
040-RECID.
    ACCEPT SC-RECID ON ESCAPE
        GO TO 130-FINISH.
050-READ-FILE.
    READ PCS-REC INVALID KEY
      GO TO 110-RECORD-CHECK.
    DISPLAY SCREEN-DATA.
    MOVE "C" TO RECORD-FLAG.
060-GET-SSN.
    DISPLAY SC-SSN.
    ACCEPT SC-SSN ON ESCAPE
      GO TO 020-CLEAR-SCREEN.
061-GET-NAME.
    DISPLAY SC-NAME.
    ACCEPT SC-NAME ON ESCAPE
      GO TO 060-GET-SSN.
062-GET-REGION.
    DISPLAY SC-REGION.
    ACCEPT SC-REGION ON ESCAPE
      GO TO 061-GET-NAME.
063-GET-COST-CTR.
    DISPLAY SC-COST-CTR.
    ACCEPT SC-COST-CTR ON ESCAPE
      GO TO 062-GET-REGION.
064-GET-DOC.
    DISPLAY SC-DOC.
    ACCEPT SC-DOC ON ESCAPE
      GO TO 063-GET-COST-CTR.
    IF NOT (PC-DOC = 12 OR 14)
      MOVE "INVALID DOC TYPE" TO
        ERR-MSG
      DISPLAY SC-ERR-MSG
.     GO TO 064-GET-DOC.
```

Figure 2 Fragment of minicomputer COBOL program

```
 IF FIRST-TIME
   PERFORM INITIALIZATION
 ELSE
   RECEIVE SCREEN FROM USER
   RESTORE SAVED VARIABLES
   SWITCH (SAVED-STATE)
     CASE 1:
       PERFORM CASE-1-PROCESSING
       MOVE 2 TO SAVED-STATE
       GO TO SEND-SCREEN
     CASE n:
       PERFORM CASE-n-PROCESSING
       MOVE m TO SAVED-STATE
       GO TO SEND-SCREEN
   END-SWITCH
 END-IF.
SEND-SCREEN.
 SAVE SELECTED VARIABLES
 SEND SCREEN TO USER
 GOBACK.
```

Figure 3 Pseudo-code generic mainframe program

recognized piece of the minicomputer program must be split into separate pieces and this requires a control restructuring of the pieces.

4. Creating New Screens

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 4. 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 separate maps or else the original fields must be rearranged to form one map

Our conversion tools must determine what maps 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 maps can actually be generated by the program. Each user input block uses one map, but a single map may be used by more than one user input block.

Analyzing control paths through a typical program with several thousand statements is combinatorially explosive. We do the analysis in several stages. First we analyze all COBOL DISPLAY statement in the program and determine which characters on the screen they affect. Then we analyze all pairs of DISPLAY to determine whether the pair conflicts, that is, writes the same characters on the screen. If two DISPLAY statements do conflict, we attempt to resolve the conflict. We end up with a list of conflicting DISPLAYs that is used in finding user input blocks described in section 5.

For example, in the example in Figure 4 the DISPLAYs on lines 5 and 8 could be resolved because both are displaying constant strings. The area on the screen that is used by both DISPLAYs can be made a variable and the appropriate constant strings can be moved to the variable before the map is presented to the user. Some conflicts cannot be resolved. These are typically conflicts where a DISPLAY writes a constant string or variable data to an area that is also used as an input field.

```
 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 4 Fragment of minicomputer COBOL program

5. General Translation Rules

That part of the conversion task that does not deal with user input is quite large in its own right. To date, we have implemented about 150 restructuring rules which currently perform about 95% of the translation work. Software engineers then examine the result of the translation, test the new program, and make any changes necessary. The rules typically look for a particular type of COBOL statement or a group of related statements, and substitute a new set of statements. The statements being replaced may be nested within other statements, so the replacement statements must merged into the abstract syntax tree at the correct place. Each rule has a unique ID and performs a specific task. In order to provide an "explanation" of why a restructuring takes place during translation, the change is marked by the ID of the rule applied, the original code that are supposed to be deleted are marked as comments. Any new statements inserted are marked with special labels.

The software engineers provide feedback in terms of how the translation may be improved so that we can add or revise the translation rules.

The general translation rules have proved to be very helpful to the software engineers and have significantly reduced the manual effort required for conversion.

6. Converting the User Interface

Definition

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 map in the mainframe program. We call such a block of code a *user input block* or *uib* .

We define a *user input block* as a block of code such that:

- The block contains a sequence of ACCEPTs.
- The block may contain code to check validity of values typed by the user, or *edits*.
- The block may contain DISPLAYs of values of fields and messages.
- In the absence of input errors and ESCAPE keys control flows from one ACCEPT in the block to another ACCEPT in the block.
- If the user presses the ESCAPE key then control goes to an ACCEPT in the block.
- If the user types invalid values then control goes to an ACCEPT in the block.
- There is a control path from every ACCEPT in the block to every other ACCEPT in the block.

A user input block defined this way corresponds roughly to a single mainframe receive. On the minicomputer within the user input block the user can type in values for fields, and can move to any other field using some combination of ESCAPE, typing values, and RETURN. On the mainframe within a map the user can type in values for fields, and can move to any other field using some combination of TAB and typing values.

Recognizing User Interface Blocks

Recognition of uib's is done by 'growing' the uib from an initial ACCEPT. The recognition algorithm is a fixed point computation - statements are added to the uib if they have a certain relation to statements already in the block. The added statements may in turn cause other statements to be added, and so on until no more statement can be added.

The relation that must be satisfied for a statement to be added to the uib is quite complex. To be added to the uib, a statement must satisfy constraints on control flow, data flow, and screen conflicts. The control constraints are those in the definition of a uib above. The data flow constraints are that DISPLAY's must not be data dependent on an ACCEPT in the uib. The screen conflict constraint is that DISPLAY's must not have an unresolvable conflict with other DISPLAY's in the block.

Translating UIB's

Once a uib is recognized, it must be translated to a send and receive and associated code. To do a correct translation, various parts of the uib must be separately identified. The general form of the translated code is as shown in Figure 5.

To do this translation we must identify within the uib:

- statements that display values of fields before ACCEPT's.
- edits, those statements that check values for validity.
- statements that display error messages.
- statements within the ON ESCAPE clauses of ACCEPT's.

The DISPLAY's of fields, which are separated within the uib, are grouped together and translated into a set of MOVE's. The edits are retained almost intact and inserted at the appropriate point in the translated code. The DISPLAY's of error messages must be analyzed and associated with a particular field. The statements within the ON ESCAPE clauses are discarded, except for the first one in the block, which usually takes control out of the uib.

Ultimate failure of UIB recognition

The approach to recognizing uib's described above ultimately failed to work on many programs. Many programs have been maintained by several programmers over many years. Many of the programs were created by heavily modifying existing programs, but forcing the new program into the structure of the original program. As a result of these practices, the COBOL programs contain extreme variations of coding styles and structures and many examples of poor coding practice.

We show in Figure 6 a simplified example of one control structure that we encountered in several programs. The code is just reading values typed by the user in a series of fields, but the excessively complicated control structure makes this difficult to determine manually, let alone automatically.

The uib recognition techniques described above worked well on the initial set of programs that we translated. However, when we attempted to apply it to programs with drastically different control structures, such as the program shown in Figure 6, it was clear that the recognition methods would have to become much more complicated and specialized to deal with all the variations in the set of programs. We decided that doing automatic recognition of uib's in all the programs would not be possible.

```
  IF FIRST-TIME
    PERFORM INITIALIZATION
  ELSE
    RECEIVE SCREEN FROM USER
    RESTORE SAVED VARIABLES
    SWITCH (SAVED-STATE) .
      CASE 1:
        Format values for DISPLAYs
        Move values to map buffer
        MOVE 2 TO SAVED-STATE
        GO TO SEND-SCREEN
      CASE 2:
        Execute edit tests from UIB
        IF there are errors
          Mark all errors
          Display message associated
             with first error
          MOVE 2 TO SAVED-STATE
          GO TO SEND-SCREEN
        ELSE
          MOVE 3 TO SAVED-STATE
        END-IF
      CASE 3:
        Code following the UIB
        . . .
    END-SWITCH
  END-IF.
SEND-SCREEN.
  SAVE SELECTED VARIABLES
  SEND SCREEN TO USER
  GOBACK.
```

Figure 5 General form of the translated code

Heuristics for user interface translation

We have taken a new approach of applying a priori knowledge in the form of heuristic rules.

We first distinguished between fields that are associated with records in permanent files, and fields that are associated only with data in the program's memory. We also distinguished between fields that are keys of records and all other fields in a record. We assume that fields associated only with data in the program's memory are for control and modification of the program's state. The heuristics we developed are:

- fields that are keys of records should be read in separately,
- non-key fields in records should be read together as one block,
- fields associated only with data in the program's memory should be read in separately.

```
100-GOTO.
     GO TO 106-DEPENDING-LOGIC.
102-RETURN.
     ACCEPT ESCAPE-CODE
       FROM ESCAPE KEY..
     IF ESCAPE-CODE = HELP-KEY
       PERFORM GET-HELP
       GO TO 100-GOTO.
     IF CHANGE-NO NOT = ZERO
       SUBTRACT 1 FROM CHANGE-NBR
       GO TO 100-GOTO.
     GO TO 190-EXIT.
106-DEPENDING-LOGIC.
     GO TO 110-GET-SSN
           120-GET-NAME
           130-GET-REGION
     DEPENDING ON CHANGE-NBR.
110-GET-SSN.
     DISPLAY SC-SSN.
     ACCEPT SC-SSN ON ESCAPE
       DISPLAY SC-SSN.
     GO TO 102-RETURN.
120-GET-NAME.
     DISPLAY SC-NAME.
     ACCEPT SC-NAME ON ESCAPE
       DISPLAY SC-NAME.
     GO TO 102-RETURN.
130-GET-REGION.
     DISPLAY SC-REGION.
     ACCEPT SC-REGION ON ESCAPE
       DISPLAY SC-REGION.
     GO TO 102-RETURN.
190-EXIT.
     EXIT.
      . . .
PERFORM 100-GOTO THRU 190-EXIT
  VARYING CHANGE-NBR FROM 1 BY 1
  UNTIL CHANGE-NBR > MAX-CHANGE-NBR
       OR ESCAPE-CODE NOT = CR-KEY.
```

Figure 6 Fragment of excessively complex code

A second set of heuristic rules deals with the order of reading fields. In the program in Figure 6 it would almost impossible to automatically determine what order the fields are read in. However, from the analysis of the screen layout the program can determine the physical order of fields on the screen. We assume that the physical order is the order in which fields should be read and processed. This heuristic lets us ignore all the problems of analyzing control and data flow and simply examine the screen layout to determine processing order.

These few simple heuristics have proved very effective in guiding the conversion of the programs and in producing translated code that is quite close to the final product.

7. Analyzing Control Paths

The recognition of uib's and the creation of maps both require analysis of possible control paths. Automating this analysis has been a very interesting, but difficult, problem.

The minicomputer source code has a wide variation of coding styles and control structures. Many of the programs have been modified and maintained for years. In many cases new programs have been created by modifying existing programs. The modifications often leave blocks of code that can never be executed.

A central problem has been reducing the number of possible control paths we need to examine by using knowledge of COBOL semantics to eliminate potential control paths. Consider this block of code:

```
1    IF A = 1
2      MOVE 2 TO B
3    END-IF.
4    IF A = 2
5      MOVE 7 TO B
6      COMPUTE C = A * B + 5
7    END-IF.
```

A superficial analysis of this block of code would construct a possible control path of (1, 2, 4, 5) when in fact this path could never be executed. We have developed a preliminary representation of COBOL semantics that we can use to reason about control paths and reduce the number of possible control paths our tools must examine. Given a control path, we can compute a form of weakest precondition on that path. If the formula we compute is not satisfiable, then the path can not be executed. In the above example the formula for the proposed path (1, 2, 4, 5) would be $A = 2 \& A = 1$, which is clearly not satisfiable.

8. 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 and involve a dozen or more CICS maps. 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.

Appendix

Semantics of minicomputer COBOL

• `ACCEPT` *`field`* `ON ESCAPE` *`statement`*
Move the cursor to *field*, which has a specified location, and allow the user to type an input value. If the user presses the ESCAPE key, then execute *statement*, otherwise assign the input value to the variable associated with *field*.

• `ACCEPT` *`field`* `FROM ESCAPE KEY`
Assign to *field* a value which reflects the last key typed by the user.

• `DISPLAY` *`field`*
Display the value of the variable associated with *field* on the screen at the specified location of *field*.

• `READ` *`record`* `INVALID KEY` *`statement`*
Read a record from the file associated with *record*. If the requested record is not found, then execute *statement*.

• `MOVE` *`source`* `TO` *`destination`*
Assign the value of *source* to *destination*.

• `GO TO` *`label`*
Transfer control to the statement immediately following label.

Session E: Models and Proposals for Program Comprehension II

Chair: Vaclav Rajlich

From Program Comprehension to Tool Requirements for an Industrial Environment

A. von Mayrhauser
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

A. M. Vans
Dept. of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

A major portion of the maintenance effort is spent understanding existing software. We present an integrated code comprehension model and our experiences with it in an industrial setting. We use audio-taped, think-aloud reports to investigate how well our integrated code comprehension model works during industrial maintenance activities ranging from code fixes to enhancements, code leverage, and reuse. We analyze the tapes for information needs during maintenance activities and derive tool capabilities accordingly.

1 Introduction

A significant portion of the maintenance effort involves the code comprehension process. Typical tasks that require understanding include troubleshooting, code leverage (reuse with modification), and program enhancement.

The first step in satisfying a maintenance engineer's *information needs* is to define a model of how programmers understand code. The literature [1, 2, 8, 7, 13, 15, 16, 17], provides two approaches to comprehension: cognitive models that emphasize cognition by *what* the program does (a functional approach) and a control-flow approach which emphasizes *how* the program works.

Chapin's Software Maintenance Life Cycle [3] divides maintenance into sub-tasks depending on whether the type of maintenance is *adaptive, perfective,* or *corrective,* or whether we are *leveraging or reusing* code. Each activity has it's own objective and presumably, it's own most effective method of understanding code to complete the tasks. Existing cognition literature states that cognitive results are best when code is systematically understood [9] at the same level of thorough detail. For large scale software this does not seem feasible nor desirable. We need a better model to understand code cognition in an industrial setting.

We investigated two existing comprehension models: Soloway & Ehrlich [16, 17], a top-down comprehension model, and Pennington's [12, 13] control-flow and functional program understanding models (a bottom-up comprehension model). Each model contains a comprehension process, the information or knowledge which are input to these processes, and the mental representation of the program. A significant limitation of these models is that all validation experiments were done using small programs. The Soloway and Ehrlich experiments used programs of about 20 lines of Pascal code. In the Pennington experiments programmers studied 200-line programs. Further, time requirements imposed artificial restrictions on the comprehension process during the experiments.

In industry, large-scale programs are the more prevalent focus of maintenance activities. We have observed that maintainers frequently switch between top-down and bottom-up comprehension. This requires a model that includes both types of understanding processes. Further, we have reason to believe that cognition during maintenance is a reverse engineering activity that re-builds an existing design (the higher level model) from the code, a design task. Guindon, et.al [6] have shown that designers frequently switch between levels of detail for their design as design progresses. Thus we would expect something similar to happen during code cognition.

This report describes an integrated code comprehension model combining both approaches. Rist [14] found that when programs are complex, construction of a mental representation of the program needs top-down (functional) and bottom-up (control-flow) understanding. Additional factors under consideration for this model are a high level of programmer expertise, a focus on maintenance tasks for industrial software, and model classification by maintenance task. Section 2 describes our integrated model in more detail. Section 3 explains the experimental method. Section 4 describes our observations and protocol analysis results. It also describes the need for program understanding tools based on the integrated model. Section 5 lists the tool capabilities derived from the observed cognition activities. In conclusion we argue that our industrial experience with an integrated code comprehension model points to the need for tools that support the cognitive process, rather than impose some process on programmers that is not justified by a validated cognition model.

2 Integrated Model

2.1 Building Blocks

First we to define a comprehension model detailed enough to identify specific maintenance tasks and task

0-8186-4042-1/93 $03.00 © 1993 IEEE

sequences. The second step identifies information requirements and tool capabilities for each task.

Existing program understanding models agree that comprehension proceeds either top-down, bottom-up, or using some combination of the two. Our observations indicate that comprehension involves both top-down and bottom-up activities. Soloway and Ehrlich's [17] model forms the basis for the top-down component (the domain model) while Pennington's [12, 13] model inspired the program and situation models. Our integrated code comprehension model consists of the following major components: (1) *Program Model* , (2)*Situation Model* , (3) *Top-Down Model* (or domain model), and (4) *Knowledge Base*. The first three are comprehension processes. The fourth is necessary for successfully building the three models. Program, situation, and top-down (or domain) model building are the three processes that together construct an understanding of code. Any of these comprehension processes may be activated from any of the other processes. Beacons, goals, hypotheses, and strategies determine the dynamics of the cognitive tasks and the switches between the models. Each process component contains the internal representation (mental model) of the program being understood. This representation differs in level of abstraction for each model. We also find the strategy to build this internal representation. The knowledge base furnishes the process with information related to the comprehension task. It also stores any new and inferred knowledge.

The *Top Down* model of program understanding is typically invoked during the comprehension process if the code or type of code is familiar. The top-down model or domain model represents knowledge schemas about the application domain. For example, a domain model of an Operating System would contain knowledge about the components an OS has (memory management, process management, OS structure, etc.) and how they interact with each other. This knowledge often takes the form of specialized schemas including design rationalization (e.g. the pros and cons of First-Come-First-Serve versus Round Robin scheduling). Obviously, a new OS will be easier to understand for a maintenance engineer with such knowledge than without it. Domain knowledge provides a "motherboard" into which specific product knowledge can be integrated more easily. It can also lead to effective strategies and guide understanding (e.g. to understand high paging rates, I need to understand how process scheduling and paging algorithms are implemented and whether a given system limits the number of pages allocated to processes.)

When code to be understood is completely new to the programmer, Pennington [12, 13], found that the first mental representation programmers build is a control flow abstraction of the program called the *program model*. For example, operating system code may be understood by determining the control flow between modules. Then we may select one module for content analysis, e. g. a scheduling model. This may use an implementation of a doubly linked list. The code representation is part of the program model. The abstraction of the scheduling queue as a doubly linked list is part of the situation model representation.

Once the program model representation exists, 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 integrated model also assumes that maintenance engineers unfamiliar with the domain first start building a program model. However, to assume that a full program model is built before abstracting to the situation or domain level would create cognitive overload for software products such as the ones we saw professionals work on (40,000+ lines of code). Rather, what we expect is abstraction of program model information at the situation and domain level as it helps the programmer remember how the program works and what is does.

Any of the three sub-models may be evoked at any time during the comprehension process. A programmer may recognize clues (called *beacons*) in the code indicating a common task such as sorting. A beacon is an index into knowledge and can be text or a component of other knowledge.[1] If, e.g., a beacon leads to the hypothesis that a sort is performed, we switch to the top-down model. The programmer then generates sub-goals to support the hypothesis and searches the code for clues to support these sub-goals. If, during the search, a section of unrecognized code is found, the programmer jumps back to building the program model. Figure 1 illustrates the relationships between the three sub-models and the related knowledge.

The definition of the integrated model allows a refinement in terms of tasks and tasks sequences for each of the three comprehension processes. We also find task strategies: the systematic bottom-up, the opportunistic top-down, and the situation model (systematic/opportunistic) task strategies.

A systematic approach applies a systematic order to understanding, e.g. code comprehension, one line at a time. An opportunistic approach studies code in a more haphazard fashion, i.e. there is no logical order to understanding. Littman et al [9] found that programmers who used a systematic approach to comprehension were more successful at modifying code (once they understood it) than programmers who took the opportunistic approach. Although the systematic strategy seems *better or safer*, it is unrealistic for reasonably large programs. We just cannot expect a fully systematic approach when understanding large-scale code. A disadvantage to the opportunistic approach is that understanding is incomplete and code modifications based on this understanding may be error prone [9]. Writing code designed for opportunistic understanding is one solution to this problem. Building tools that help make opportunistic understanding less error prone is another.

2.2 The Program Model

When code is completely new to the programmer Pennington [12, 13], found that the first mental rep-

[1] E.g, a beacon may be the characteristic pattern of value switches. Or it may be the name of the function (such as QSORT).

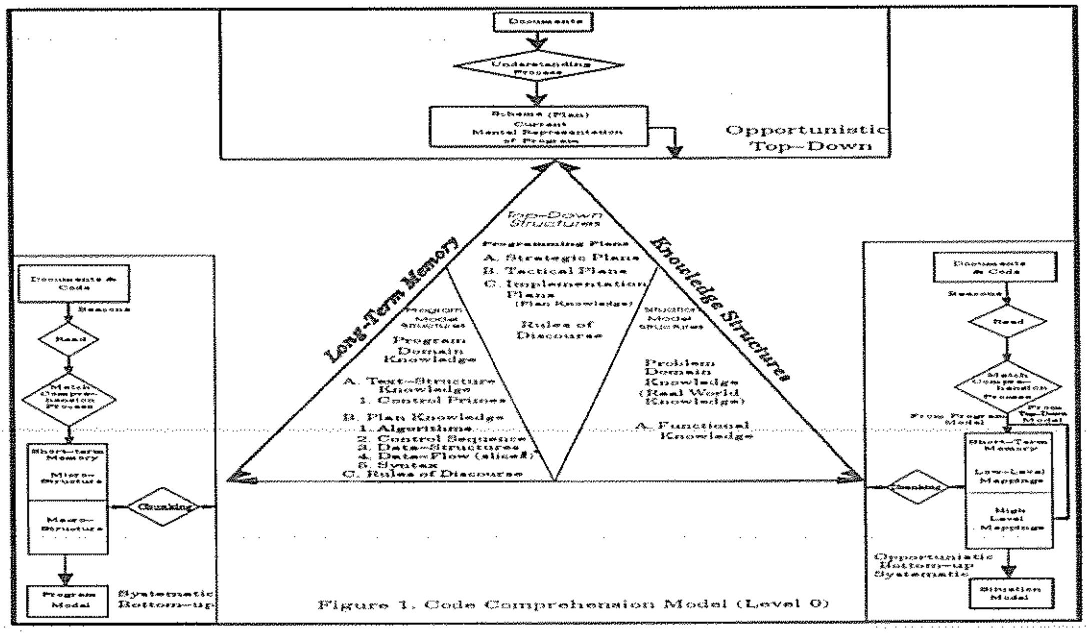


Figure 1. Code Comprehension Model (Level 0)

resentation built is a control flow abstraction of the program (the *program model*).

Vessey, [18], developed a model of debugging that identified tasks that can be mapped into the program model. These tasks include:
1. Reading comments or other related documents
2. If the *micro-structure* is the current focus, then the comprehender examines the next module in text sequence. This is logical since micro-structure development is defined as the understanding of statements and their interrelationships.
3. If the *macro-structure* is the current focus, then the comprehender examines the next module in a control-flow sequence. Macro-structure consists of abstracted chunks of micro-structure components which are identified by a label and structured to correspond to the control-flow of the program text. Analyzing the next module in a control-flow sequence is reasonable here because code examined when building a control-flow abstraction as part of the program model will not typically reside physically in sequence. Further, examining control-flow implies that some micro-structure has been constructed because something must be known about the surrounding code before a control-flow thread can be meaningfully followed.
4. Data-structures are examined.
5. Data slicing. This involves tracking changes to variable contents during program execution.
These first five tasks constitute the matching portion of the program model. The knowledge base can be used through out the matching process. Specific data-structure knowledge is necessary to examine the data-structures in the code.
6. The last step involves chunking the learned information and simultaneously storing it in the current mental representation and long-term memory (knowledge base). A jump to the situation model is also possible.

Let us illustrate program model building using operating system process scheduling as an example: sections of code that take a job from the ready queue, put it into the running state, monitor elapsed time, and remove the job after the time quantum has expired are recognized and then abstracted with the label round-robin scheduler. This information then becomes part of the situation model.

2.3 Top-Down Model

Koenemann and Robertson [7] show that program comprehension as a goal-oriented, hypothesis-driven, problem-solving process. The top-down model of program understanding is normally invoked during understanding if the application area is already familiar. For instance, suppose an expert programmer, whose specialty is operating systems, is asked to maintain an operating system she has never before seen. As an expert, she can immediately decompose the new system into elements she knows must be implemented in the code: a process manager, a file manager, an I/O manager, and a memory manager. Each of these can be decomposed: e.g. process management includes interprocess communication and process scheduling. Process scheduling could be implemented through: round robin, shortest job first, or priority scheduling, etc. The expert operating systems programmer continues top down until she recognizes a block of code, in this case, the precise process scheduling algorithm. It is not necessary to re-learn this algorithm line by line. The engineer must only recognize that the appropriate code exists.

Tasks identified in this model map into the top-down element of the integrated model. The top-down understanding process consists of:
1. Gaining a high-level overview.
2. Determining the next program segment to examine.
3. Determining relevance of the code segment to current mental representation or hypothesis.

4. Generating/revising hypothesis. The programmer generates hypotheses about the functionality of the program segments under consideration while scanning code. A jump to situation model building occurs if an hypothesis is confirmed.

2.4 Situation Model

The situation model can be constructed by mapping functional knowledge acquired through opportunistic understanding to high-level plans. Alternatively, it can be built bottom-up as mappings from the program model in the form of lower level abstractions of actual program code.

The knowledge structures in this model reflect real world knowledge that can be grouped into different domains or Pennington's plan knowledge for the situation model. The knowledge domains are Brooks' [2] knowledge domains.

For example, at the domain level we may have a picture of a process control block (PCB) with the types of information it might contain. At the situation model the PCB may be represented as a table. At the program model level we see a C-structure and how it is used/updated in the code. Actual OS modules at the program level are recognized, then abstracted, and used as information at the situation model level.

2.5 Knowledge Structures

The *knowledge base*, also known as long-term memory, is usually organized as schemas. The large triangle in figure 1 represents this knowledge base. Schemas are grouped into partitions specifically related to the comprehension processes. Situation model structures are associated with the situation model, program model structures are used in program model building, and top-down structures are applied during top-down construction. For example, knowledge of distinctive algorithms (situation model information) or data-structures are used by the *program model building* process, while knowledge about objects in the real-world (domain information) are typically used by the situation model. Specifically, Pennington's [12, 13] text-structure and plan knowledge is used when constructing the program model. Soloway and Ehrlich's [16, 17] strategic, tactical and implementation plans with rules of programming discourse are used when developing a top-down view of a program. The knowledge base also acts as a repository for any new knowledge gained. We associate newly acquired knowledge with the corresponding model.

3 Method

After model definition, the next step is model validation and analysis of experimental results. We audio taped work sessions which were later transcribed.

3.1 Participants

Our participants were professional maintenance programmers working with large size code, from 40K up to 500K lines of code. We classified them into domain and language experts, domain only experts and language only experts. We also distinguished by levels of expertise, types of maintenance tasks, and accumulated knowledge about the code when the session started [21]. These maintenance engineers were considered experienced by their peers and willing to participate in the study. We gathered a wide variety of cognitive behaviors. For purpose of explanation of results, we will explain the cognition behavior of two of our participants.

Prior to the programming session, the engineers were asked to find a maintenance task that included a code comprehension component. Two of the sessions involved debugging activities, the first a simple bug fix, the second debugging after an enhancement. From an industrial perspective most were relatively small, about 40,000 lines of code. Compared to existing cognition experiments this is large. We recorded a two hour session and later transcribed it for analysis. The engineers' statements were studied for a high-level verification of the integrated model and for identification of information needs.

The first debugging session involved fixing a bug found by a customer. The engineer had just recently assumed responsibility for the system and was not familiar with the domain or program itself. He had studied bug reports and talked with the engineer who had responsibility for the system previously. The engineer's lack of domain expertise characterized this session.

The other session was part of the testing phase of an enhancement. The engineer was an expert in the operating system with which he was working. He was also a language expert. He previously implemented an enhancement to the operating system and was in the process of testing the enhancement when he found several bugs.

3.2 Procedure

The participants were asked to think aloud while working. We audio taped them. Sessions were typically two hours long. This is not enough to understand a large scale software product. This is why we decided to find participants with varying degrees of prior experience with the code as this would give us the widest degree of coverage over the code comprehension process.

The audio tapes were then transcribed. Protocol analysis was divided into the following:

1. Enumeration of cognitive activities as they relate to the integrated cognition model of section 2. We also analyzed for possible new activities or interactions between model components.
2. Segmentation of the protocols as to which part of the integrated model is involved in the code understanding (domain, situation, or program model).
3. Identification of information and knowledge items and their aggregates. This included identification of beacons, domain schemas, hypotheses, strategies, and switches between domain, situation, and program model.

Verbal protocol analysis is an exploratory observational technique for research in a new domain. The study of cognitive processes during software maintenance and evolution is such a new domain.

Similar to [6], we faced three issues in determining the validity of generalizing results from our data to other maintenance situations:

1. Task. The tasks (and the code the participants tried to understand) were all representative of maintenance tasks commonly encountered in industry. While not all participants were doing the same task, they were all trying to understand industrial strength C code to maintain it. As we begin to understand whether and how cognition differs, we can move to more specialized tasks to explore each situation better.
2. Sampling of participants. How representative are our participants of the larger population of software engineers who work with existing code and need to understand it? There is no reliable answer given the current maturity of the field. We attempted to get a broad sampling of maintenance tasks, prior work with the code, and programmer experience in domain and language. At this point we cannot claim that these protocols represent the full range of cognition behavior of the population of software maintenance engineers. It is more likely that the description of this population will need to be assembled from many (future) studies similar to ours.
3. External validity. This concerns the degree to which the conditions under which our data were collected are representative of those under which actual maintenance occurs. Code cognition for maintenance purposes takes more than two hours. This we considered by including different amounts of prior preparation in our study. Further all tasks studied represent actual work assignments. Both of these strengthen the generalizability of our findings.

4 Observations

While we give examples of two of our subjects, we can report similar observations from other programming sessions.

Our subjects worked at all three model levels: the domain, situation, and programming model level. Table 1 shows the number of references to each of these model components. It also identifies some of the intermediate cognition goals to support model building.

At the end of two hours the engineer performing the bug fix did not yet have a great understanding of the bug and, as a result, was unable to implement a correction. In fact, he had determined that the information he received characterizing the bug was incorrect and that the engineer who previously had responsibility had not understood the bug. The engineer spent almost half of the time (48%) developing a situation model because he lacked the necessary real-world knowledge. Most of the remaining time (45%) was spent in building the program model with components at the program model level used as drivers for situation model construction.

In contrast, the engineer working on the enhancement not only fixed several bugs during the session, he was also able to identify additional bugs he did not know about. He spent a majority of the time (70%) developing a program model, and substantially less time (20%) in the situation model. This makes sense because he wrote the faulty code and needed to revise the program model. It did not function according to his original program model developed during the enhancement implementation. Very few references to the situation model were necessary because the engineer was an expert in the domain algorithms and did not question his understanding of the operating system.

Both engineers referred to the top-down model infrequently. This is expected since the nature of the debugging activity is one of focusing in on a specific problem rather than developing a high-level view. In other experiments, especially general understanding by engineers who are taking responsibility for a piece of code, the top-down model was referred to at least one third of the time, [21].

Hypotheses are important drivers of cognition. They drive further investigation. Thus generating hypotheses about code and investigating whether they hold or not is an important facet of code understanding. Hypotheses may cause a switch to another model level. In our example analysis we found 33 hypotheses of which 3 failed during the debugging session while 11 out of 31 hypotheses failed during the enhancement session.

The integrated model assumes that these switches can occur between any of the model components at any time. We specifically do not assume that understanding is built from top down (i. e. full domain model before going to situation and program model) or from bottom up (full program model before going up to situation model).

Table 2 shows that the integrated model is correct. We find indeed a significant number of switches between levels. Let us illustrate these switches between levels with an example.

One of our subjects was developing an elaborate theory about the function of a particular type of data structure called "Debug control strings". He developed several hypotheses at several levels of abstraction while switching between situation, program, and top-down model mental representations. "They apparently have some things here that control other routines..." Reading the comments he gets confused. "I was building a theory that debug control strings were used to print debugging output and the reason they [debug control strings] have state is because when they.." <He then explains how they behave using control-flow language>. "Only now this sentence about appearing on the shell command line and passed into other application programs has confused me a bit. So I guess I really still don't understand what they are for."

5 Tool Capabilities

We found that maintenance engineers need to build a mental program model, a situation model and a domain model. They switch frequently between these three levels. Understanding is facilitated when programmers have domain and language experience, as they provide specialized schemas. This guides the understanding process. Specialized schemas act like a "motherboard" into which the acquired knowledge can be integrated. Specialized schemas also support strategies and building and refining of hypotheses. They facilitate retention and retrieval of knowledge about the code.

Model	*Code*	*Process Task*	*Bug Fix*	*Enhance-ment*
Opportunistic Top-Down	OP1	Gain High-Level Overview of Program	12	1
	OP2	Determine next program segment to examine	4	9
	OP3	Generate or revise hypothesis about functionality	2	1
	OP4	Determine relevance of program segment	1	0
	OP5	Determine if this program segment needs detailed understanding	0	0
	OP6	Determine understanding strategy	0	0
	OP7	Investigate Oversight	0	2
	OP8	Failed Hypothesis	1	0
	OP9	Mental Simulation	0	0
		Top-Down Knowledge	3	19
Systematic Program Model	Sys1	Read introductory code comments any related documents	2	2
	Sys2	Determine next program segment to examine	19	6
	Sys3	Examine next module in sequence	19	14
	Sys4	Examine next module in control-flow	1	12
	Sys5	Examine Data structures and definitions	0	2
	Sys6	Slice on Data	0	0
	Sys7	Chunk & store knowledge	27	17
	Sys8	Generate Hypothesis	33	31
	Sys9	Construct Call Tree	2	0
	Sys10	Determine understanding strategy	14	9
	Sys11	Generate new task	0	15
	Sys12	Generate Question	0	8
	Sys13	Determine if looking at correct code	0	2
	Sys14	Change direction	0	2
	Sys15	Generate/consider alternative code changes	0	3
	Sys16	Answer question	0	1
	Sys17	Add/Alter Code	0	15
	Sys18	Determine location to set breakpoint	0	10
	Sys19	Failed Hypothesis	3	11
	Sys20	Determine error/omitted code to be added	0	7
	Sys21	Mental Simulation	0	7
		Program Model Knowledge	16	29
Situation Model	Sit1	Gain Situation Model knowledge	36	3
	Sit2	Develop Questions	13	2
	Sit3	Determine answers to questions	6	1
	Sit4	Chunk & Store	32	6
	Sit5	Determine relevance of situation knowledge	4	0
	Sit6	Determine next info to be gained	5	0
	Sit7	Generate Hypothesis	15	4
	Sit8	Determine understanding strategy	7	1
	Sit9	Determine if error exists (missing functionality)	0	2
	Sit10	Failed Hypothesis	3	2
	Sit11	Mental Simulation	2	2
		Situation Model Knowledge	25	33

Table 1: Task Counts for Debugging and Enhancement

As our maintenance engineers worked with code of significant size (40,000 plus lines of code), cognitive limitations play a big role in the understanding process. We need better tools to alleviate capacity limitations of short-term memory. We also need tools that make retrieval of information from long term memory more reliable, for example by representing domain model schemas and situation model schemas in a user understandable form.

Lack of relevant specialized knowledge leads to slower code cognition, partly because understanding has to proceed mostly bottom up with small chunks [10]. It also is slower, because strategies are inefficient: the maintenance engineer does not have the higher level functional knowledge about what the code is supposed to do and how one commonly implements it. This precludes identifying strategies for opportunistic understanding. It also takes longer to build hypotheses. Further, chunking in small pieces leads to many layers of program model until the situation model level can be reached. This is cognitively taxing.

Lack of specialized knowledge is also associated with a lack of cognitive knowledge structures supporting memory during cognitive activities. For example, a maintenance engineer needs to remember postponed hypotheses and the tests for them, as well as the results of hypothesis tests.

Most of the tools our maintenance engineers have today emphasize support for building a program model. A few reverse engineering tools support aspects of domain and situation models. Yet they all can be considered single level tools. In our experience, maintenance engineers frequently switch between these three levels as they build and test hypotheses and find beacons that trigger retrieval of knowledge at a different level of abstraction.

While a comprehension model allows us to understand how programmers go about comprehending

Number of References Debugging	*Number of References Enhancement*	*Model*	*Model Switches – Debugging*			*Model*	*Model Switches – Enhancement*		
			Top-Down Model	*Situation Model*	*Program Model*		*Top-Down Model*	*Situation Model*	*Program Model*
22	32	Top-Down	N/A	14	3	Top-Down	N/A	6	16
143	58	Situation	14	N/A	28	Situation	9	N/A	17
133	204	Program	3	24	N/A	Program	12	20	N/A

Table 2: References and Switches Between Models

code, it does nothing to make this process more productive. Tools are needed to support and speed up the comprehension process. Such tools must quickly and succinctly answer programmer questions, extract information without extraneous clutter, and represent the information at the level at which the programmer currently thinks [20]. Thus tool information should be available at the program model, situation model, and top-down model levels. Tool information should accommodate the relevant knowledge structures, and should aid in switching between components of the code comprehension model.

Unfortunately, most 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 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 identifying a strategy on how to recognize code). Nor is the information represented succinctly as an answer to a specific question (e.g. a full data-flow table as compared to showing data-flow of a specific variable through code highlighting and/or code elision). Connections between the three levels of code comprehension are not commonly tool-supported. What is worse, we still see attitudes reflected in tool builders' minds that if we just teach programmers to understand code the way they ought to (i.e. the way their tools work), the understanding problem will be solved [4].

Maintenance programmers work at all levels of abstraction. Indeed, we can argue that cognition is close to reverse engineering, as in both cases we must develop a higher level abstraction of existing code. Thus our current best bet to supporting code understanding for maintenance engineers at the situation and domain level is the current state of the art of reverse engineering tools.

Table 3 shows a partial tool capabilities matrix we extracted from analyzing cognition behavior and information needs. For a more detailed tools capability table see [19]. This table is based on the observations reported in this paper and contain domain-specific items. Some information items will need to be replaced by other domain-specific items. E.g., internal support routines and utility functions will change based on the application domain under consideration.

While some tools with these capabilities exist, tool availability thins when we go to the situation and domain model level. (A $\surd$ mark next to a tool capability in table 3 indicates that such tools exist.) We also find too few that are able to support switches between levels. No tools are capable of formulating and keeping track of hypotheses, representation of domain knowledge and specialized domain schemas, cognition strategies, and analysis of hypothesis failure.

6 Conclusion

We must support the natural process of understanding code with our tools, not hinder it. With the information gained from our integrated model and from our experiments we developed a tool capability matrix which relates comprehension tasks to individual tool capabilities.

Many maintenance tools are designed to enforce a prescribed maintenance process. We suspect that different work styles of programmers (e.g. systematic study of all related information, focused study only on information perceived to be related to the task at hand, or dynamic execution of program) affect the actual task sequence during maintenance activities. When tools support maintenance processes that are defined in terms of task sequences these tools may actually hinder the understanding process.

Some of the tool capabilities we found are non-trivial, such as the need for a function classification scheme that is not textual and crosses several levels of abstraction, or the availability of a domain expert. Further progress in tool support for maintenance engineers will require facilities for representing hypotheses and switches between domain, situation, and program models. We also must represent strategies and help the maintenance engineers to remember postponed hypotheses and the tests for them as well as the results of testing hypotheses. In summary, we need better tools for representing the intermediate results and dynamically defined steps of code understanding.

Acknowledgements

This research was supported by the Hewlett-Packard Co.,Inc.

References

[1] Ruven Brooks, **Towards a theory of the cognitive processes in computer programming**, In: International Journal of Man-Machine Studies, 9(1977), pp. 737-751.

[2] Ruven Brooks, **Towards a theory of the comprehension of computer programs**, In: International Journal of Man-Machine Studies, 18(1983), pp. 543-554.

Task	*Sub-task*	*Information Needs*	Tool Capability
Gain High-Level View (Top-Down Model)	Study function call structure	Display of call graph	Full System Display √
	Determine file structure	Directory layout/organization include files, main file, support files, library files	Source Text Reformatter √
	General Understanding at high-level	Maintenance manual, info from project documents	On-line documents with keyword search across docs
	Determine most frequently used functions and routines	Count of Function calls	Function Counts √
		Identify utility functions	Utility function list
Determine Next Module to Examine (Top-Down) Model)	Narrow down possibilities	Organized functions into categories in which functions are related.	List of functions categorized according to specific classification scheme
		In call graph, get rid of extra info not relevant, e.g. error routines, I/O, functions not in specified category.	Pruning of call tree based on specified categories
	Want to see overall structure without irrelevant information	Structure of relevant functions only at different levels of abstraction	Pruning of call tree based on specified categories
	Look at one function in a group of functions to get an idea of what and how all functions in group work	A general classification of routines and functions so that if one is understood the rest in the group will be understood	List of functions categorized according to specific classification scheme
Generate Hypothesis (Top-Down Model)	Determine if the routine performs vital function	Routines that do most of the application domain type work.	Display domain applicable functions.
	Determine if function is internal support function	List of internal support functions	Display internal support routines.
Gain Situation Model Knowledge (Situation Model)	Recall previously learned information	Situation knowledge	Scratch files √ annotatable code
	Determine nature of bug and characterized.	Bug behavior isolated	On-line bug reports √
	Duplicate bug	How to duplicate bug	Simulate behavior, steps to reproduce include data
Develop Questions (Situation Model)	Determine what a specific term means in the Situation Model	Definitions of terms/concepts in the application domain	On-line docs with key-word search √
Determine next Info to be Gained (Situation Model)	Understand a domain concept	Sources on domain knowledge knowledge	On-line access to ref manuals explaining domain concepts
			Domain expert
Generate Hypothesis (Situation Model)	Determine meaning of acronym of acronym	Acronym Definitions	Acronym definitions using functional definitions
Determine Next Program Segment to Examine (Program Model)	Search for calling procedure to see how object was used.	Use of object in a specific procedure.	On-line Cross √ reference
	Return to previous code segment	Last place in code before preceding branch	History of browsed locations √
Chunk & Store (Program Model)	Determine how a particular procedure fits together with surrounding & adjacent procedures	The nesting level of a particular procedure or function	Fan-in/Fan-out √ Function Level
Generate Hypothesis (Program Model)	Determine if a word is a reserved reserved word	List of reserved words	On-line language √ docs with hypertext
	Look at one object use in a group of object uses to get idea of what & how the object is used in all references	A general description of how the object is used including any deviations from expected use	On-line documentation with key- √ word search
Construct Call Tree (Program Model)	Determine call-tree structure for a specific object.	Call-tree graph for a specific object	Graphical display call-graph with pruning for specific object. √
Determine Understanding Strategy (Program Model)	Determine if a program language construct is different from constructs in similar languages	List & explanation of constructs that deviate from standard program language stmts and constructs	On-line language documents with key-word search & hypertext √
	Determine if all calls to objects will be investigated	Frequency count of references to a particular object	Source Code Metrics √ Number of entities
	Determine best code segment to examine next when confused	Good direction to follow given what is already known, possible program segments to examine	Intelligent agents give advise based on what has been done.

Table 3: Tool Capabilities

[3] Ned Chapin, **Software Maintenance Life Cycle**, In: Conference on Software Maintenance, 1988, pp. 6-13.

[4] **Program Comprehension Workshop – CSM-92**, Workshop notes, IEEE Computer Society, Conference on Software Maintenance, November 9, 1992, Orlando, Florida.

[5] Edward M. Gellenbeck and Curtis R. Cook, **An Investigation of Procedure and Variable Names as Beacons during Program Comprehension**, Tech Report 91-60-2, Oregon State University, 1991.

[6] Raymonde Guindon, Herb Krasner, and Bill Curtis, **Breakdowns and Processes During the Early Activites of Software Design by Professionals**, In: Empirical Studies of Programmers:Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 65 - 82.

[7] Jurgen Koenemann and Scott P. Robertson, **Expert Problem Solving Strategies for Program Comprehension**, In: ACM? March 1991, pp. 125-130.

[8] Stanley Letovsky, **Cognitive Processes in Program Comprehension**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 58 - 79.

[9] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway, **Mental Models and Software Maintenance**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 80 - 98.

[10] Katherine B. McKeithen and Judith S. Reitman, **Knowledge Organization and Skill Differences in Computer Programmers**, In: Cognitive Psychology, 13(1981), pp.307-325.

[11] Paul W. Oman and Curtis R. Cook, **The Book Paradigm for Improved Maintenance**, In: IEEE Software, January 1990, pp. 39-45.

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

[13] Nancy Pennington, **Comprehension Strategies in Programming**, In: Empirical Studies of Programmers:Second Workshop, Eds. Olson, Sheppard, and Soloway, ©1987, Ablex Publishing Corporation, pp. 100 - 112.

[14] Robert S. Rist, **Plans in Programming: Definition, Demonstration, and Development**, In: Empirical Studies of Programmers: 1st Workshop, 1986, Washington, D.C., pp. 28-47.

[15] Ben Shneiderman, **Software Psychology, Human Factors in Computer and Information Systems**, In: Chapter 3, ©1980, Winthrop Publishers, Inc., pp. 39-62.

[16] Elliot Soloway and Kate Ehrlich, **Empirical Studies of Programming Knowledge**, In: IEEE Transactions on Software Engineering, September 1984, Vol. SE-10, No. 5, pp. 595-609.

[17] 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, Lawrence Erlbaum Associates, Publishers, pp. 129-152.

[18] Iris Vessey, **Expertise in debugging computer programs:A process analysis**, In: International Journal of Man-Machine Studies, (1985)23, pp.459-494.

[19] A. von Mayrhauser and A. Vans, **Code Comprehension Model**, Technical Report # CS-92-145, 1992, Colorado State University, Fort Collins, CO.

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

[21] A. von Mayrhauser and A. Vans, **An Industrial Experience with an Integrated Code Comprehension Model**, Technical Report # CS-92-205, 1992, Colorado State University, Fort Collins, CO.

[22] Susan Wiedenbeck, **Processes in Computer Program Comprehension**, In: Empirical Studies of Programmers, Eds. Soloway and Iyengar, ©1986, Ablex Publishing Corporation, pp. 48 - 57.

Model-Based Tools to Record Program Understanding

E.J. Younger & K.H.Bennett

School of Engineering and Computer Science, University of Durham, South Road, Durham DH1 3LE, UK

Abstract

Tools to record the knowledge gained by maintainers engaged in understanding an existing program are described. These tools are based on a model of the comprehension process and of reverse engineering as a whole. They form a part of an integrated reverse engineering toolset based around a central repository. Using these tools new documentation for an existing software system may be built up incrementally by successive maintainers who work on the system.

Keywords: Program comprehension, documentation, reverse engineering, incremental redocumentation.

1 Introduction

The twin problems of understanding existing software systems and the frequently poor quality of their supporting documentation are generally accepted to be among the greatest facing software maintainers today. The two are related – good quality documentation can greatly aid the process of understanding an application, and its absence forces the maintainer to spend a great deal of time in studying the application code in order to rediscover elements of the design of the system which could have been better recorded by its developers and subsequent maintainers.

The process of understanding an unfamiliar system has been shown to be the most expensive part of a typical maintenance task [1], and software maintainers frequently cite poor documentation as the greatest hindrance to them in their work [2]. Despite this fact, the understanding gained by maintainers working on a system is rarely recorded for the benefit of their successors. This compounds the problem of concentration of knowledge about an application in a small number of "experts", who have worked on the same system for years and have now become indispensible, having an intimate knowledge of the system which is inaccessible to others.

This paper describes research carried out into the process of understanding software and the recording of this understanding, as part of the Esprit II project REDO [3], and the tools developed to record the understanding gained by successive maintainers who work on a particular application. This knowledge becomes part of an information base stored with and linked to the application itself.

2 Problems with documentation

Many applications in use today were developed 15 or 20 years ago, before the advent of well-defined development methods, and certainly before there was much concern about the problem of maintenance. Such systems very often have little or no documentation of use to a maintainer.

Major software development projects are frequently behind schedule and over budget. Towards the end of such a project, the pressure is on the development team to meet deadlines for the release of the software. The result is that anything considered non-essential to the release is postponed, often indefinitely. Generally, documentation (other than that which accompanies the released system) is considered to fall into this category. Development documentation therefore becomes out of date as refinements and amendments are made to the system and never documented — this also applies to the post-release maintenance phase of the product's life. In the worst case some documents are never written at all.

The advent of IPSE's and software engineering databases allows documentation to be stored in a common repository with the rest of the application. This in itself goes some way towards overcoming the above problems: the status of the documentation is implicitly elevated to that of an integral part of the application, rather than a optional extra. It is also possible to include the documentation in the same quality assurance procedure which is, or ought to be, applied to the application as a whole. Similarly, where a computerised configuration management system is used to manage a system the documentation can be brought under its control also.

This technology is however very recent. Existing applications were with very few exceptions developed without the aid of such systems, and so the documentation is typically stored separately from the rest of the application, either online in a separate document preparation system, or much more probably on paper only.

Even good quality documentation may not be useful to the maintainer if it does not convey the information required to carry out the task at hand. System documentation is generally restricted to two types: development documentation and user documentation. Whilst both provide information which is useful to a maintainer, neither are produced with the needs of maintainers in mind, and therefore do not necessarily present information in the best form for use in maintenance. Even the best development documentation certainly does not contain all the information needed to understand an unfamiliar system. Much background information, concerning for example the application domain, rationale for design decisions, etc. are typically omitted. This is especially true when the application has been developed by specialists in a particular domain, who will have substantial undocumented knowledge both of the application domain itself and of the optimal design of systems to meet its requirements.

Thus the maintainer approaching an unfamiliar system is typically confronted by documentation which is incomplete, out of date, inconsistent, difficult to access, difficult to understand, or in the worst cases inaccurate. In many cases the documentation is so poor, or the maintainer has so little confidence in its accuracy, that it is not used at all, and the maintainer relies solely on the source code to understand the system. It is this situation which redocumentation seeks to rectify.

3 Program comprehension theories

The process of program understanding has been studied almost exclusively by observation of programmers at work on real or "artificial" problems which require them to develop an understanding of an unfamiliar system. These experiments have varied in their approach; some have encouraged the participant to "think aloud" while studying a program, and have recorded and analysed the results [4]. Others have allowed a period to study a program or set some task such as a simple modification to force the participants to develop an understanding of the program, and subsequently set tests to to determine the level of understanding developed, the nature of the information used, and the concepts in whose terms knowledge of the program is expressed (e.g. [5]). As a result, numerous theories have been expounded covering various aspects of the comprehension process.

Brooks has suggested that comprehension is based on a system of mapping between the problem domain and the programming domain [6]. The developer constructs these mappings and the maintainer has to reconstruct them, using whatever information is available (which may be only the source code). It is argued that this process is a bottom-up activity, rather than the top-down approach taken in the development process. The bottom-up approach to comprehension is also described by Basili and Mills [7]. Letovsky argues that the comprehension process typically is a mixture of both bottom-up and top-down procedures, the programmer switching between the two and exploiting one or the other on the basis of cues in the available information [4].

It is widely accepted that an understanding of programs is developed in terms of **plans** (also variously known as *cliches* or *schemata*). A plan is a typical action sequence in a program which implements some common function [8] [9] [10]. Plans are lower-level concepts than algorithms, which are typically implemented by combining a set of plans [9]. In understanding a program, an experienced programmer attempts to identify plans in the code. This process may be complicated by the fact that the statements which make up a plan can be distributed in the code [11].

3.1 Knowledge types and representations

Knowledge stored in human long-term memory can be subdivided into the two categories of *factual* and *semantic* knowledge. (See for example [12]). Factual knowledge is concerned with such information as the spelling and pronunciation of words, telephone numbers, syntactic constructs in languages, etc. Semantic knowledge is concerned with the meaning and interpretation of factual knowledge, and with its organisation into structures and models of reality.

Factual knowledge appears to be stored in a flat, unstructured form. It is specific, i.e. it is not transferable to a domain other than that of its origin. The nature of the access paths to factual knowledge is not clear, but it has been observed that the less frequently factual knowledge is accessed, the "weaker" the access paths become. There is no evidence that knowledge ever disappears from the brain, however the access paths may become too weak to be found.

Semantic knowledge conversely is structured, the structure itself being a part of the knowledge. It appears to be represented as a network of links representing semantic relationships between facts. It is also layered — networks of related information at one level may be treated at a higher level as a single item.

Whereas factual knowledge is common between individuals (there being no alternative representations for the correct spelling of a word, for example), semantic knowledge is highly individual. The difference between novices and experts in a particular field lies largely in their semantic knowledge. Novices may begin with much the same body of factual knowledge, but the lack of structure in their knowledge means that they are unable to make the logical connections between facts that allow an expert to reason about problems or to transfer knowledge by analogy between related domains. As expertise is gained it is manifested in an increasingly rich network of semantic relationships built over the unstructured factual knowledge.

3.2 Knowledge types in comprehension

Schneiderman and Mayer [13] identify three type of knowledge used in program comprehension:

1. **Syntactic knowledge of programming.** This includes all programming language–, operating system–, and hardware–specific facts and rules, such as the grammar of the programming language.

2. **Semantic knowledge of software engineering.** This includes knowledge of programming algorithms and plans, data structures, etc. This knowledge is language independant. It also includes strategic knowledge about the appropriate application of this knowledge.

3. **Semantic knowledge of the application domain** This is knowledge about the real-world domain in which the program operates. For example, it might be knowledge of accountancy practice, physics, air-traffic control procedures etc. Any knowledge of the business-rules of the users of the program fits into this category. It also includes knowledge of typical use of software engineering semantic knowledge in a particular application domain.

It appears from the experiments described above that the major differences between novice programmers and experts lie in the types of knowledge which they possess. Novices begin with syntactic knowledge and a small amount of software engineering semantic knowledge. They then spend their careers in gaining more and more semantic knowledge of software engineering and application domains. Novices have insufficient semantic knowledge to enable a top-down comprehension strategy to be used, and are forced therefore to rely predominantly on syntactic knowledge and hence a bottom-up approach. They also typically lack domain knowledge and are thus unable to relate program functionality to the real world. Pennington [5] has demonstrated that programmers whose mental models of a system incorporate concepts from both the application domain and the program/software engineering domain develop a better understanding of the system than those who rely heavily on one or other type of knowledge.

3.3 Comprehension strategies

Littman et al. [14] have described two strategies which programmers were observed to use when studying small programs with a view to making changes to them. In the first strategy, the *systematic* approach, the programmer studies and attempts to understand the entire program before beginning to modify it. In the second, *as-needed* strategy, the programmer studies and understands only those parts of the program which are deemed necessary to carry out the particular task. The writers conclude that the systematic approach leads to superior performance. However the study was restricted to small programs. In the case of a large application, it is impossible or at least impractical for a maintainer to gain a complete understanding of the entire system, and as a result an as-needed approach must necessarily be adopted.

In the as-needed approach, it is necessary to identify the parts of the application which are relevant to the maintenance task. It is possible to do this in either a top-down or bottom-up fashion. The bottom up approach begins by analysing the syntax of program statements in order to derive semantic information. More abstract constructs are then developed by grouping statements which together implement plans. The top-down method involves searching the program for those plans which are *expected* to make up the functionality of the program. This process is an iterative one of hypothesis refinement and validation. Brooks [6] introduces this process, and it is further expounded by Gilmore [15]. The programmer begins with an initial hypothesis about the functionality of a program, routine or module based for example on its name or on application domain knowledge. Software engineering knowledge suggests ways in which this functionality may be implemented, i.e. which plans are to be expected. The programmers then tests this initial hypothesis by examining the code, and refines the hypothesis iteratively until it matches the actual code (see fig 1). Formation of an accurate initial hypothesis is greatly assisted if the specification and design of the program are well documented.

The important factors for our model to emerge from this research are

1. the process of comprehnsion is iterative and not linear, even for a small program for which a complete understanding can be developed

2. for larger programs, the maintainer develops an understanding of the program only to the extent necessary to allow the maintenance task to be carried out. Large parts of the program typically will be omitted from this process, while a detailed knowledge of other parts will be acquired.

The design of appropriate and useable tools both for program comprehension and to record the understanding gained must take account of these factors.

4 Redocumentation

Redocumentation is the process in which new documentation is generated for an existing system either to replace or augment any documentation which already exists. Recording understanding gained about a program is a form of redocumentation – it is concerned with making available to other maintainers the information required to understand the system.

The approach to redocumentation adopted will vary depending on the aim of the activity. Redocumentation may take place in response to a variety of triggers, for example:

- In response to a maintenance request, parts of the system may be (wholly or more probably partially) redocumented as a side effect of the maintainer's efforts to understand the system.

- As part of a full-scale reconstruction of the application to improve its maintainability, complete redocumentation of the application may be carried out. The reconstruction process may itself be triggered by various stimuli. [16]

These two scenarios correspond conceptually to the two system comprehension strategies discussed in section 3, namely the *as-needed* strategy and the *systematic* approach. The latter is a very expensive undertaking, and in practice occurs very infrequently. In by far the majority of cases, redocumentation occurs as a side effect of some other maintenance activity. The process of understanding a large application is as we discussed earlier both iterative and incremental. Iterative in the sense that it is based on a cyclic process of refining and validating hypotheses, and incremental in

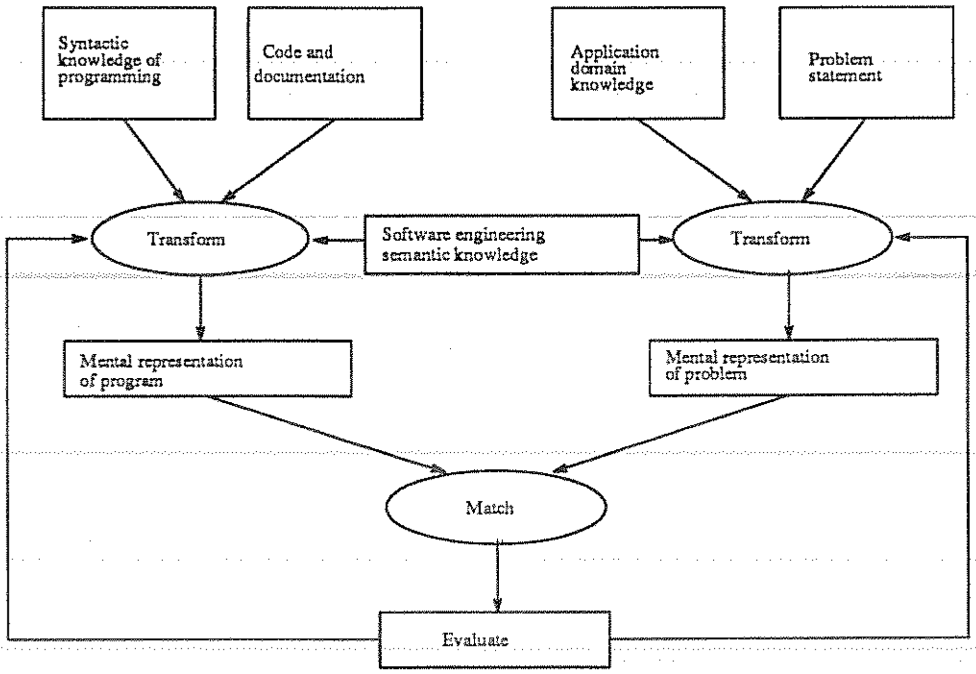

Syntactic knowledge of programming
Code and documentation
Application domain knowledge
Problem statement
Transform
Software engineering semantic knowledge
Transform
Mental representation of program
Mental representation of problem
Match
Evaluate

Figure 1: A model for program comprehension

the sense that only parts of the application are studied during a single maintenance task. Tools to support the recording of understanding must take account of this in their design.

Maintainers engaged in satisfying particular maintenance requests must gain an understanding of parts of the application in order to modify it. The understanding required is quite wide–ranging and at various levels of abstraction. A general understanding of the high–level design may be needed to identify those modules which need to be modified. Partial knowledge of the control– and data–flows may be required to identify potential side–effects elsewhere in the application. A detailed knowledge of the implementation in the vicinity of the change(s) is of course essential (assuming that is that the application is maintained at the implementation level, rather than (partially) re–implemented from a modified design).

This knowledge may be gained using a variety of methods, e.g. code–reading, analysis of code using software tools, etc. The knowledge gained is very often not recorded for the benefit of future maintainers, resulting in duplication of effort - research has shown that large parts of a typical application remain stable over time while particular "hot spots" are modified repeatedly. Failure to record the understanding gained represents a waste of resources, which could be avoided if appropriate tools were available.

We define *incremental redocumentation* to be the recording of knowledge about the application as it is acquired. It will generally be a side effect of some other maintenance activity rather than a task in its own right. We are not aware of much research in this area though tools have been reported which allow the user to enter informal text and link this to program code [17] [18]. If this approach is adopted then an information base of knowledge about the application will be built up as a natural part of the maintenance process, concentrated on exactly those "hot spots" which are most likely to be the subject of future maintenance activities.

An information base which is subject to maintenance, in the sense of casual updating, is likely to degrade with time just as applications do. Good practice in the maintenance of both can combat this trend, but is unlikely to eliminate it completely. A tool which supports the casual incremental addition of information is likely to speed up the degradation of quality unless steps are taken to prevent this. Factors which could contribute to deterioration of quality include

- maintainers entering incorrect, inconsistent, duplicate or conflicting information.
- modification of the application without corresponding changes to the documentation.
- proliferation of information, resulting in duplicate or redundant information, and a poorly structured database with "spaghetti" links.

A degree of consistency checking between the application and its documentation may be performed automatically if appropriate links are set up in the database. This might be at the level of flagging code which has been changed without a change to the corresponding documentation. Correctness and consistency of the documentation itself can be maintained by a quality control process for the application as a whole, and by version and configuration management procedures. It is essential that the maintainer can have confidence in the correctness of any documentation which is used as a basis for understanding the application, otherwise it is likely that the documentation will simply not be used. There is a clear need to mark documentation so that information which has been quality-checked and found to be reliable can be identified as such. This applies equally to existing documentation which has been imported into the maintenance environment, and to new documentation generated by users of the system. Proliferation of information is perhaps a consequence of ease-of-use in this case. The aim of the tools is to encourage users to record the knowledge which they gain about the system for the benefit of their successors. Restricting the update facilities offered by the tools according to the identity of the user can go some way to controlling this, but a periodic restructuring/reconstruction of the database might still prove necessary.

The desirable features of a (re)documentation system for software maintenance can now be summarised (see also [17]):

1. **Integrated source code**
   Traceability from documentation to the related source code is essential in maintenance. The direct embedding of documentation in the code itself is possible, using either the commenting facilities of the language or systems such as WEB [19] but is not desirable for the reasons discussed above. The system should therefore use some form of linking between separately stored code and documentation.

2. **Integrated technical documentation**
   Technical documentation should be linked both to the code and to other documentation in the system, to allow it to be used efficiently.

3. **Integration of existing documentation**
   It should be possible to incorporate any existing documentation for the application, whether on-line or on paper, into the system. This should not require the re-typing of all existing documentation, as this will prove expensive, but it should be possible to incorporate references to external material unavailable on-line.

4. **Incremental documentation**
   It should be possible to build up the documentation for the application gradually over time, as described above, and the system must be useable despite the documentation being incomplete.

5. **Informal update**
   It must be easy for the maintainer to add information to the documentation base while engaged in studying the application. The intention is to provide a system which encourages maintainers to record their understanding of the application as it emerges. A system which is cumbersome to use or which causes the user's train of thought to be broken is not likely to be used in the manner intended.

6. **Quality assurance**
   In order to prevent degradation of the documentation over time, quality assurance procedures are essential. These can be aided by stamping new information with the identity of its author and the date and time its entry. Existing information can be consolidated and summarised periodically to prevent the documentation base from growing uncontrolledly.

7. **Configuration management**
   Where an application is evolving or where different versions exist, configuration management must allow the documentation appropriate to a particular software version to be recovered.

8. **Information hiding**
   The documentation base for a well documented application of even moderate size will be very large. It must be possible to view this information at various levels of abstraction, and to screen out unrequired parts of the documentation.

9. **Team use**
   The team involved in maintaining or reverse-engineering a typical application will have many members. Therefore the documentation system must support concurrent access and update by multiple users.

5 Requirements for tools

We can now begin to formulate some requirements for documentation tools, based on the analysis in the the preceding sections of this paper. It is important that we base the design of our tools on a sound model of the task which they are to support. In our case there are two related tasks: system understanding and redocumentation. In the former process tools are required which the maintainer will use to access existing documentation as an aid to system comprehension, and in the latter tools which support the addition of information to the database of documentation.

We concluded in our earlier analysis that the process of system understanding both iterative and incremental, using a mixture of top down and bottom up approaches and is based, for large systems, on an "as-needed" strategy wherein the understanding developed is confined to those areas of the application to which the maintenance task is related. As a result of this process knowledge about the application is gained, and this knowledge needs to be recorded for the benefit of future maintainers. Since the comprehension process is an iterative one, the tools for recording this knowledge must support incremental recording or redocumentation.

5.1 The REDO maintenance environment

The maintenance tools produced by the REDO project are integrated around a shared database known as the System Description Database or SDDB. This database is implemented using the Eclipse toolbuilder's kit from Ipsys Software plc. [20], which provides repository facilities based on the PCTE Object Management System data model [21], extended to provide fine-grained data modelling below the level of PCTE objects. The data model is essentialy a network model with object-oriented extensions.

A populated SDDB contains a representation of the application code stored as an abstract syntax tree decorated with semantic information [16]. In addition it contains documentation for the application. This includes "conventional" documentation in the form of lifecycle documents (specifications, design documents, user documentation etc.), documentation extracted from the application (control and data flow, etc.), and also some more novel forms of descriptive material made possible by the SDDB itself. A populated SDDB can be regarded as self-documenting, in the sense that an SDDB instantiated to reverse-engineer a particular application will also contain information to describe the application. This will include pre-existing documentation (if any) which has been loaded into the SDDB, and information generated incrementally during the iterative process of reverse engineering and added to the SDDB.

5.2 Documenting relationships

Reverse engineering is defined by Chikofsky and Cross [22] as the identification of the components of a system and their inter-relationships. This decomposition of a system into components may in principle be done in a variety of ways according to various paradigms. Such decompositions constitute an abstract view of the system which can be regarded as a design. Since the redocumentation tools form part of a reverse engineering toolkit, the users of the tools will usually be performing reverse engineering tasks. Therefore the tools should be capable of documenting the results of these tasks, which will include the relationships between components of the application. This requirement supplements the requirement to record and represent relationships between components of documents, and between documentation and the application. Since the application and documentation are stored in a repository/database, the possibility exists to represent relationships by *links* in the database. Some of these can be set up permanently when the database is loaded; others may be built up incrementally as part of the comprehension and redocumentation process.

The nature of these relationships are many and varied. Relationships will exist between documentation representing the development process, of refining specifications through designs to implementation, and the nature and semantics of these are well defined. Other relationships will exist which are less generic in nature, and in many cases are peculiar to the individual application. Their semantics cannot be specified *a priori*. In designing a tool to represent relationships we are therefore faced with a choice. We may choose to define a set of semantic relationships between components of a system, each of which is represented in the database by a unique link type. This approach has been adopted by the EPSOM project in the ESF (Eureka Software Factory) programme. [23] The disadvantage of this approach is that it restricts the documentation tools to represent only relationships having the semantics represented by one of the predefined link types. In order to provide a sufficiently rich choice of semantics the set of link types will necessarily become large.

Alternatively, we may provide a single generic link type, and leave it to the user to define the semantics of the each link, i.e. the nature of the relationship which it represents, when it is created. This has the advantage that only a single link type is required, and relationships of any kind may be represented. The disadvantage is that, since the user defines the meaning of the link, it is possible for different users to use different terms to describe essentially the same semantic relationship , with the effect that one user's description may be misinterpreted or not understood by a subsequent user. There is therefore a need, if this alternative is adopted, for an agreed standard use of terms among the users.

This latter approach has been adopted in the REDO toolkit, since it conveys the advantages described above. It does however require a means for the user to define the semantics of links on an individual basis. The most general method possible has been adopted: the user enters a textual description of the relationship which the link represents. This is attached as an attribute to the link itself.

The links themselves are completely generic, in that they are in principle able to link any two types of object in the SDDB. These might be nodes in the abstract syntax tree of the original source code, components of documents, or abstractions created by the reverse engineering tools. The resulting network of objects connected by links representing relationships between them corresponds well with the above definition of the aim of reverse engineering.

5.3 Incremental documentation

Incremental documentation tools provide the maintainer with the means to record information about the application as it is acquired, during the process of understanding the application. The tool for documenting relationships discussed above falls into this category; a tool is also required to enable the maintainer to add new textual documentation to record knowledge gained about the application. This text is attached, as an autonomous object, to the component of the application which it describes. In this respect the tool represents an advance on the DOCMAN system [18] in that text may be attached to any node in the syntax tree of a program, or in principle to any object in the SDDB. As a distinct object, this text or *note* may in principle have relationships with other components of the application, its documentation, etc. which may be represented using the link creation tool discussed above.

These notes may be used for several different purposes, for example:

1. as temporary annotations for the maintainer's own personal use only

2. as permanent additions to the documentation of the application

In addition, when a large number of notes is attached to particular object, it may be appropriate to create a note which summarises the existing notes, without replacing them.

The type of a note (permanent, temporary or summary) should be made visible to the user, as should the author, date and time of creation, and Q.A. status (see below), as these may influence the interpretation of the note text. As large numbers of notes may exist, the user may wish to see only notes of a particular type, only quality-checked notes, only notes by a particular author, etc. The tool which displays the notes ought to provide such filtering facilities.

5.4 Quality assurance

The need for quality asssurance of documentation was identified in section 4. Notes and links each have a set of attibutes which are used for Q.A. purposes. Each note or link is marked automatically with the date and time of its creation and the identity of the user who created it. A third attribute is set if the note or link has passed a quality inspection. This attribute can only be set by a member of a set of *priveleged users*: this is enforced by the tools.

The text of a note or the annotation of a link may be edited after creation, but in the interests of maintaining quality this may only be done by the creator of the note or link or by a priveleged user. Similarly, notes and links may be deleted from the database only by their creator or a priveleged user.

6 The REDO documentation tools

In addition to the tools described in detail below, the REDO toolkit includes a browser/editor for conventional structured documentation, which is stored online in the central repository together with other components of the application. The approach taken in this tool is similar to that used in structured documentation tools such as SODOS [24]. The tools described in this paper represent a different, complementatry approach to information recording and presentation.

6.1 Note tool

The Note Tool supports incremental redocumentation of an application using notes, as described above. The note tool is always invoked from another tool, which displays the "target" object, i.e. the object to which the note is to be attached. Typically this would be done by selecting the target object with the mouse and selecting from a pop-up menu. The documentation browser provides such an interface, as does the related source code browser.

Each note comprises of a set of structured information, including the date and time of creation, author's identity, quality status etc., and an unstructured body in which the text of the note is stored. The Note Tool has two functions: to browse through existing notes and to add new ones. The tool has two subwindows for these functions, each with its related "control panel" containing buttons and information fields (see figure 2).

The two subwindows are respectively for the display of the text of existing notes (note display window), and for the entry and editing of the text of new notes (text edit window). The message window displays information or warning messages from the tool. The existing notes are displayed one-at-a-time, starting with the most recent summary note if one exists or the most recent note otherwise. The "Next" and "Previous" buttons are used to step through the set of notes, in chronological order. The "Make Link" button invokes the Link Tool (see below) to create additional links to or from the displayed note.

The "Set Filters" button accesses a subwindow which allows the user to restrict the set of notes which are displayed on the basis of their Status, Quality and Author attributes.Only those notes whose attributes have the selected values will be displayed, for example only summary notes which have been vetted for quality. With the exception of the Author attribute, these values are selected from pop-up menus. Where there are a large number of notes, this facility is useful to restrict the tool to only those types of notes which the user wishes to see.

Two further options are available only to the author of the current note or to a priveleged user. The "Delete" button removes the current note, and the "Edit Note" button allows the text of the current note to be edited in the text edit window. If the current user does not have the necessary priveleges then these buttons are removed from the control panel. The values of the other attributes of the note, such as its status or quality attributes, may be altered via the control panel fields. The quality and status fields each have an associated set of possible values, which may be selected via a pop-up menu.

The text edit window provides a mouse-driven editing environment for the entry of new note text or the editing of existing notes. The type of the note can be set via a menu associated with the "Note Type" field. Only a priveleged user may create a note of type "Summary". The default note type is "Temporary", and all new notes are created with quality attribute set to "Not_Vetted".

6.2 Link tool

The Link Tool supports the documentation of relationships between components of the applications, by creating links between these components and storing these in the SDDB. A link is in fact an object in its own right, which is connected to its "source" and "destination" components using the linkage mechanism of the underlying database. Again, the Link Tool is in practice always invoked by another tool, which displays the object to which the link is to be connected. In fact, the two objects which are to be linked must be displayed, though not necessarily in the same tool or even simultaneously on the screen. As is the case with the note tool, the object which is to be linked

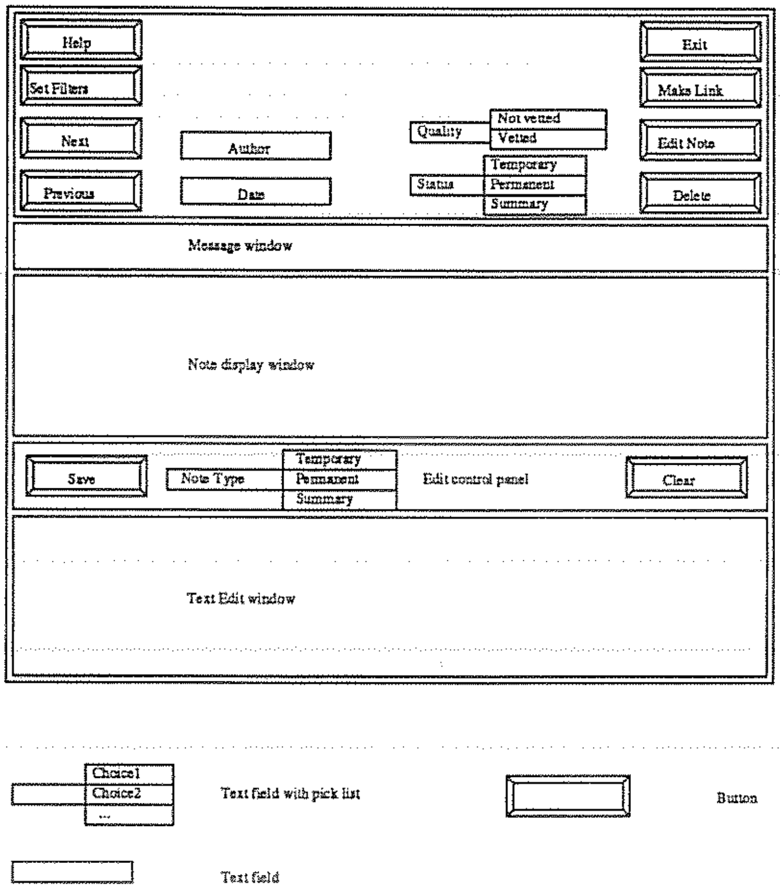


Figure 2: The Note Tool window

from is selected, and "Create Link" is selected (typically from a pop-up menu, though this may depend on the invoking tool). This starts a copy of the link tool, whose window then appears on the screen. The object to be linked *to* is then selected in its display window and the process is repeated. The information for the "destination" object is added automatically into the Link Tool window (see figure 3. Text to annotate or describe the link can then be typed into the appropriate field in the tool window; clicking on the "Create Link" button completes the link, which is added to the database. A reverse link may then be created; the annotation text for this is created by the tool by prepending the string "Inverse of" to the annotation for the "forward" link, though this can be edited or changed by the user.

In addition to the date and time of creation and identity of the link's creator, a third attribute records the quality status of the link. It is set to FALSE when the link is created, as in the case of a new note, to indicate that the link has not yet undergone a quality inspection.

7 Conclusions

The tools described in this paper support the recording of knowledge about an application which is gained by maintainers during the performance of maintenance tasks. These tools are conceptually based on a model of application understanding as an iterative and incremental process. The goal of this process is, ultimately, to "reverse engineer" the application to produce a more abstract view, expressed as a set of components and their inter-relationships. The tools support the incremental addition of information to a repository of knowledge, linked to the application itself, and also the documenting of relationships, as they are discovered, by the creation of links withint this central repository which holds the application and its documentation. The resulting network of components and relationships represents the emerging abstract view of the application. Existing information in the repository, entered during earlier maintenance, is made available via a simple and intuitive user interface. Quality assurance facilities mean that users of the system can have confidence in the quality of the information on which their own understanding of the application may be based.

The system is in some senses incomplete and in

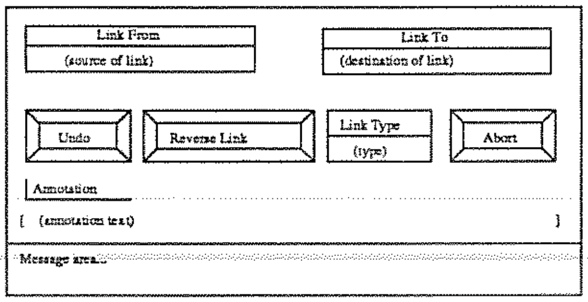


Figure 3: The Link Tool window

need of further development. While the facility exists to document relationships between components of the application, the capability to navigate to related information by following the links in the repository is presently missing. There is no great technical difficulty in adding these facilities, however this was not possible during the original research project due to shortage of resources. Also lacking at this point is an evaluation of the tools in a commercial software maintenance environment. This is to be expected during the exploitation of the REDO toolkit by the commercial partners in the project.

References

[1] T.A. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering* SE-10 (1984), 494–7.

[2] E. Chapin, "Software maintenance: a different view," National Computer Conference, AFIPS Conference Proceedings 54, 1985.

[3] P.S. Katsoulakos, "REDO," *CASE '90. Fourth international workshop on computer aided software engineering, Irvine, California* (December 1990).

[4] S. Letovsky, "Cognitive Processes in Program Comprehension," in *Empirical Studies of Programmers*, E. Soloway & S. Iyengar, eds., Ablex, Norwood, NJ, 1986, 58–79.

[5] N. Pennington, "Comprehension Strategies in Programming," in *Empirical Studies of Programmers: Second Workshop*, G.M. Olsen, S. Sheppard & E. Soloway, eds., Ablex, Norwood, NJ, 1987, 100–113.

[6] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies* 18 (1983), 543–54.

[7] V.R. Basili & H.D. Mills, "Understanding and documenting programs," *IEEE Computer* (October 1982).

[8] E. Soloway, "Learning to program = learning to construct mechanisms and explanations," *Communications of the ACM* 29 (1986), 850–8.

[9] R.C. Waters, "The Programmers Apprentice - knowledge-based program editing," *IEEE Transactions on Software Engineering* SE-8 (1982).

[10] W.L. Johnson & E. Soloway, "PROUST - knowledge based program understanding," *IEEE Transactions on Software Engineering* SE-11 (March 1985), 267–275.

[11] S. Letovsky & E. Soloway, "De-localised plans and program comprehension," *IEEE Software* 3 (1986), 41–9.

[12] W. Kintsch, *Memory and Cognition*, Wiley, 1977.

[13] B. Schneiderman & R. Mayer, "Syntactic/semantic interactions in programming behaviour," *International Journal of Computer and Information Science* 8 (1979), 219–38.

[14] D.C. Littman, J. Pinto, S. Letovsky & E. Soloway, "Mental Models and Software Maintenance," in *Empirical Studies of Programmers*, E. Soloway & S. Iyengar, eds., Ablex, Norwood, NJ, 1986, 80–98.

[15] D.J. Gilmore, "Models of debugging," *Proceedings of Fifth European Conference on Cognitive Ergonomics, Urbino, Italy* (September 1990).

[16] H.J. van Zuylen, ed., *The REDO Compendium of Reverse Engineering for Software Maintenance.*, John Wiley, Chichester, 1993.

[17] N.T. Fletton & M. Munro, "Redocumenting Software Systems using Hypertext Technology," *Proc. Conference on Software Maintenance, Phoenix, Arizona* (1988).

[18] J.R. Foster & M. Munro, "A Documentation System based on Cross-Referencing," *Proc. Conference on Software Maintenance, Austin, Texas* (1987).

[19] D.E. Knuth, "Literate Programming," *The Computer Journal* 27, 97–111.

[20] Ipsys Software plc., Toolbuilder's Kit 2.5, 1991.

[21] ECMA, "Portable Common Tool Environment (PCTE): Abstract Specification," ECMA Standard ECMA-149, 1990.

[22] E.J. Chikofsky & J.H. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Software* 7 (Jan. 1990).

[23] E. Minor, "Results from the ESF EPSOM project," *6th European Software Maintenance Workshop, Durham, UK* (September 1992).

[24] E. Horowitz & R.C. Williamson, "SODOS: A Software Documentation Support Environment - Its Definition," *IEEE Trans. Software Engineering* SE-12 (Aug. 1986), 849–59.

A Hybrid Approach to Recognizing Programming Plans

Alex Quilici
Department of Electrical Engineering
University of Hawaii at Manoa
Honolulu, HI 96822 *

Abstract

Most current models of program understanding are unlikely to scale up successfully. Top-down approaches require advance knowledge of what the program is supposed to do, which is rarely available with aging software systems. Bottom-up approaches require complete matching of the program against a library of programming plans, which is impractical with the large plan libraries needed to understand programs that contain many domain-specific plans. This paper presents a hybrid approach to program understanding that uses an indexed, hierarchical organization of the plan library to limit the number of candidate plans considered during program understanding. This approach is based on observations made from studying student programmers attempt to perform bottom-up understanding on geometrically-oriented C functions.

1 Introduction

Our current research in automated program understanding concentrates on the problem of automatically extracting geometric objects and operations from sizeable C programs. In particular, we're primarily concerned with recognizing:

- The representation and use of objects such as points, lines, circles, squares, and polygons.
- The implementation of operations that involve these objects, such as finding the distance between two points, determining whether two lines intersect, noticing when a point is contained within a circle, and deciding whether a particular triangle is a right triangle.

*An earlier version of this paper appears in the Proceedings of the 1993 IEEE Conference in Reverse Engineering, Baltimore, MD, May 1993.

The ultimate goal of our project is to be able to automatically recognize these objects and operations, and then to replace them with libraries containing human-generated object-oriented code written in C++. We've chosen this particular project as a study for automated program understanding for several reasons. First, even if we can only automatically recognize and replace a small portion of the many objects and operations in a set of real-world programs, it's a win over attempting to revise these programs entirely by hand. And second, doing this task by hand requires constructing replacement C++ classes, so it's not unreasonable to assume that information about C implementations of these classes can be provided while constructing the replacement classes.

Our approach to this task has been to study student programmers attempting to understand short C functions and then to try to build a program that mimics the process they go through. This paper describes the resulting model.

2 Background

Most recent approaches to program understanding [4, 6, 7, 10, 11, 14, 15] feature:

- An internal representation of the actual program instructions and the data and control flow between them.
- A library of programming plans or cliches [12].
- A process for mapping program actions to these plans.

The final result of their understanding process is usually some sort of tree (or lattice) structure, with program instructions at the leaves, programming plans in the nodes, and the goals the program achieves as the root.

0-8186-4042-1/93 $03.00 © 1993 IEEE

The process of understanding can usually be classified as either top-down or bottom-up. The top-down approach starts from knowledge about the goals the program is assumed to achieve, determines what plans can achieve these goals, and attempts to connect these plans to the actual program instructions. Typically, this includes using matching rules to detect how these instructions achieve various subgoals within a plan, and difference rules to recognize how they differ from the instructions expected by a plan. In contrast, the bottom-up approach starts with the instructions themselves, determines which programming plans might have these instructions as components, attempts to infer higher-level goals from these plans, and repeats the process until the programmer's actual goals are recognized.

Both of these program understanding approaches have shown significant promise, but both also have obvious limitations. The top-down approach requires detailed advance knowledge about the goals the program is supposed to achieve, which is often unavailable for many real-world programs. It also is not well-suited for performing partial understanding, since it understands a program fragment only when it's connected to a top-level goal. On the other hand, bottom up approaches tend to suffer from a combinatorial explosion of possibly relevant plans. Each program action may be part of a variety of plans, which themselves can be part of a wide variety of plans, and so on. This explosion greatly limits the length and complexity of the programs that can be understood with this approach, unless a method can be found for limiting the search space, an area that has so far received little attention.

Previous program understanders have avoided worrying about the size of the search space by either performing top-down searches for a limited number of plans or by performing bottom-up searches with a library containing a limited number of largely domain-independent plans. Unfortunately, understanding existing, aging programs in the geometric domain requires a bottom-up search and a large library. That's because these programs are most naturally described in terms of *domain-specific* objects and operations, which necessitates recognizing not only those programming plans that carry out these operations but also the programming plans representing the objects being manipulated. As a result, a program understander can't rely primarily on a library of general programming plans, such as READ-PROCESS-LOOP or COMPUTE-SQUARE, although these are often necessary. Instead, it must also have a library that contains a variety of domain-specific operations, such as DISTANCE-BETWEEN-POINTS, IS-POINT-ON-LINE, IS-POINT-IN-CIRCLE, and so on, as well as information about objects such as POINTS and ARRAY-OF-POINTS.

These domain-specific operations and objects form a domain model that's absolutely critical for understanding the programs built on top of them. Unfortunately, however, as soon as domain-specific plans are necessary, the plan library tends to increase dramatically in size. And as the plan library grows, so does the need to limit the number of plans that are considered during the program understanding process. Unlike current automated program understanding systems, human programmers appear to do this quite sucessfully. That's led us to observe student programmers performing bottom-up program understanding and to create a model of program understanding based on the process we've observed. Our belief is that capturing the process by which humans focus their search for programming plans has the potential to significantly improve automated plan recognition techniques.

3 A Study Of Student Programmers

We've studied a small group of student programmers attempting to understand a set of geometric functions. Figure 1 shows one of the functions given to the students to understand.[1]

The point of this study was to discover which plans from the plan library were considered during program understanding and what indices triggered the consideration of these plans. We deliberately chose the geometric functions we used for this study to be novel compositions of plans the students were likely to have seen before. This allowed us to focus on plan retrieval and recognition techniques without worrying about the additional problems inherent in understanding unplanful code [2]. We also deliberately chose relatively simple functions without global variables rather than more realistic complete programs. This allowed us to avoid worrying about complications such as delocalized plans [8]. Finally, we deliberately chose implementations for our functions that minimized the available syntactic indices to plans [13], such as variable names that accurately describe the use of a variable. This allowed us to focus on conceptual indices: particular combinations of actions that suggested considering a particular plan.

[1]The reader should try to understand what this function does before continuing. That way the reader can compare his or her understanding process to the process that we observed in our student programmers.

```
/*
 * Mystery function
 */
#include <math.h>

#define BIG_DOUBLE 1.0e38

double mystery(int a[], int b[], int n)
{
  double r, d, t1, t2;
  int j, k;

  j = 0;
  r = BIG_DOUBLE;
  while (j < n-1)
  {
    k = j + 1;
    while (k < n)
    {
      t1 = a[j] - a[k];
      t2 = b[j] - b[k];
      k = k + 1;
      d = sqrt(t1 * t1 + t2 * t2);
      if (d < r)
        r = d;
    }
    j++;
  }
  return r;
}
```

Figure 1: An unknown C function.

Our methodology was to take think-aloud protocols of students attempting to understand these functions. Students were told to verbalize any question or hypothesis they generated about the code, as well as any conclusions they drew and what led them to draw that conclusion. They were given no advance knowledge of what the functions were supposed to do, and were told to try provide as succinct a description as possible of what the program actually did. To better understand the process, we frequently asked them to explain why or how they generated a hypothesis or reached a conclusion.

3.1 How Students Understood This Example Function

We now describe the process students went through in understanding this program.

The first plan most students recognized was that the outer while was iterating j from 0 to n − 1 by 1. Protocols and later questioning suggested that the students reached this conclusion by going through the following steps:

1. Skipping over the initial declarations and assignment statements. That is, they didn't try to match the assignment statements against any plans (athough several students puzzled briefly over what BIG_DOUBLE was before continuing).
2. Noticing that the outer while contained a relational test involving a variable. Most hypothesized that this loop was performing an iteration on j (the plan ITERATE-OVER-RANGE), although one suggested that it was instead iterating on n. No students immediately hypothesized an iteration by 1 (the more specific plan FULL-ITERATE-UP-OVER-RANGE)
3. Confirming that a variable in the test (j) was initialized before the loop and updated during the loop. At this point, most concluded that j was the iteration variable, and that this plan was to run j through all values from 0 to n − 1.
4. Verifying that the other variables in the test weren't changed during the loop body, that the iteration variable was only updated once in the loop, and that the loop contained no hidden exit (such as a break). This step was actually done by a only a few of the students.

After recognizing the function of the outer loop, students then took an almost identical approach to recognizing that the inner loop was performing a similar task.

Students next tried to figure out what the combined loops were doing. Most students quickly concluded that the loops were generating pairs of j, k values (the plan GENERATE-PAIRS). Students then proceeded to determine exactly what these pairs of values were. For most, that involved actually writing these pairs until a pattern emerged that they could classify.[2] A few, however, examined the inital values and test few, however, examined the inital values and test cases and concluded that these loops were generating unique pairs (the plan GENERATE-UNIQUE-PAIRS) between 0:1 and n-1:n (*0:1*, *0:2*, ..., *0:n*, *1:2*, *1:3*, *1:n*, and so on). Some then hypothesized that the plan was examining the top half of a matrix, but quickly discarded that hypothesis when they realized the loop didn't contain a two-dimensional array.

[2]This process is known as concrete simulation [9]. It appears to arise frequently when programmers are dealing with unplan-like situations and are trying to evalute the external coherence between plans [2]. Its occurrence suggests that our geometric functions contained plans that some students had never before encountered.

Students then proceeded to try to figure out what happened inside the loop. No one formed a hypothesis or drew any conclusions about the subtractions. When they encountered the `sqrt` expression, however, they all recognized several plans. They began by trying to understand the subexpressions in its argument. They recognized that the multiplications were actually computing squares (the plan COMPUTE-SQUARE) and that the addition was computing the sum of these squares (the plan SUM-OF-SQUARES). At this point, almost all students hypothesized that this expression was computing a distance (the plan COMPUTE-DISTANCE) and went back to verify that the values being squared were the result of subtractions.

After recognizing the distance computation, students jumped rapidly to conclusions. First, that the combination of `a[j]` and `b[j]` formed a point (the plan POINT) and, similarly, that the combination of `a[k]` and `b[k]` formed another point. Second, that `a` and `b` were arrays representing a collection of points (the plan ARRAY-OF-POINTS). Third, that `j` and `k` were indexing POINTs in this array of points. Fourth, that the loop was therefore generating indices which were used to select pairs of points. And finally, that the loop's job was to run through and compute the individual distances between each of these pairs of POINTs.

After recognizing the loop's primary function, most students then examined the `if` and the assignment it contained. Students used the `if`'s relational test of two values and its assignment of the smaller value to hypothesize that a minimum was being saved. Most verified that variable to have been assigned before the iteration and some explicitly noted they now understood the purpose of `BIG_DOUBLE`. Students were then satisfied they understood the purpose of this function. The usual student summary was something like: "The function returns the distance between the closest pair of points in a set of points."

3.2 Some Lessons Learned

From this informal study, we've learned several lessons that have significantly influenced our model of the process of program understanding.

First, student programmers don't match every program action against every programming plan that might contain it. Instead, they attempt to recognize a program plan only when they are *reminded* of it by a combination of program actions or program plans. For example, the combination of a `while` and a relational test involving a variable and any relational operator other than equality consistently caused students to consider the the plan ITERATE-OVER-RANGE. Similarly, the combination of a FULL-ITERATE-UP-OVER-RANGE contained in another FULL-ITERATE-UP-OVER-RANGE caused them to consider the plan GENERATE-PAIRS. And the combination of a SUM-OF-SQUARES contained in a SQUARE-ROOT caused them to consider the plan COMPUTE-DISTANCE. None of the students immediately considered ITERATE-OVER-RANGE when they encountered assignments, nor did they consider COMPUTE-DISTANCE when they noticed subtractions, even though assignments and substractions are steps in these plans.

Principle #1:

Program understanders require explicit indexes from combinations of program actions to entries in the plan library. They should match only indexed plans against a given code fragment, not every possible plan known to contain that code fragment.

Second, student programmers try to specialize the programming plans and actions they encounter in the code before moving on to the next program fragment. For example, after students recognized the plan GENERATE-PAIRS, they then proceeded to try to discover exactly what types of pairs it was generating, and most eventually realized that what they had was the specialization GENERATE-UNIQUE-PAIRS. Similarly, after recognizing the plan ITERATE-OVER-RANGE, all students realized that the iteration variable was going up by one each time, and noted that they had the more specialized plan FULL-ITERATE-UP-OVER-RANGE. And all students specialized the multiplications of t1 and t2 to instances of the plan COMPUTE-SQUARE.

Principle #2:

Program understanders require a specialization hierarchy of programming plans. They should then try to specialize any recognized program action or plan as far as possible before trying to understand the role of the next program action or plan.

Third, student programmers attempted to draw conclusions after recognizing plans. For example, after recognizing the plan COMPUTE-DISTANCE, all students immediately realized that this meant that the values being subtracted represented coordinates within POINTs. And after recognizing that points were represented as elements from a pair of arrays with the same subscript (the plan ARRAY-POINT), all students realized that the arrays themselves were an instance of the plan ARRAY-OF-POINTS. These realizations were the crucial ones in understanding the real purpose of the C function they were given.

Principle #3:

Program understanders require that programming plans contain explicit knowledge about additional plans that can be inferred once they are recognized. They should make these inferences before continuing on to try to recognize other plans.

Essentially, what our study suggests is organizing the plan library as a specialization hierarchy of plans, with explicit indices from combinations of plan components to plans, and with explicit implication links to other plans whose presence they imply.

4 Our Plan Library

Our approach to constructing a plan library has been to extend the representation and organization used in the well-described plan library developed by Andersen Consulting for understanding Cobol programs [6].

Essentially, this work represented programming plans as data structures containing two parts. The first is a plan definition, which lists the attributes of the plan that are filled in when instances of the plan are created. The plan ITERATE-OVER-RANGE, for example, has five attributes:

INDEX: the index variable.

START: the expression that calculates the initial value of the index variable.

END: the test for whether to go through the loop.

UPDATE: the expression that computes a new value for the index variable.

SEQUENCE: the set of statements to be iterated.

The second part of a programming plan is a plan recognition rule, which lists the components of a plan and the constraints on those components. An instance of the plan is recognized when all its components have been recognized without violating the constraints. Consider the plan ITERATE-OVER-RANGE. To recognize this plan, it's necessary to find the following components in the code:

LOOP-STATEMENT: a language looping construct,

INIT-EXPR: an assignment expression,

END-TEST: an expression to determine whether to stay in the loop,

UPDATE-EXPR: an expression that assigns a new value to a variable.

The constraints are that: the INIT-EXPR must precede LOOP-STATEMENT, the LOOP-STATEMENT has a data dependency on INIT-EXPR, the UPDATE-EXPR must be contained in the body of the loop and update the variable assigned to in the INIT-EXPR, and the END-TEST must appear as part of the LOOP-STATEMENT's test.

This description of a programming plan is not sufficient to support the process used by our student programmers. In particular,

1. It lacks indexing information about under what circumstances it is worth checking for the plan's presence. This means that a bottom-up search is stuck trying to match all possible plans that might have a particular program action as a component.

2. It contains no explicit representation of the differences between plans at different levels in the hierarchy. This means that to recognize a specialization of a recognized plan, it's necessary to do a complete match against the components and constraints of that specialization.

3. It contains no knowledge about other related plans that can be inferred once the plan is recognized. This means that additional matching must be done to try to recognize these other plans

To alleviate these difficulties, we provide three additional pieces of information with each programming plan: indices, specialization constraints, and a list of implied plans.

4.1 Plan Indices

Each plan can index other plans in the library. An index consists of a set of tests and a pointer to a partial instantiation of the indexed component. The tests are represented with arbitrary predicates, as are constraints. The partial instantiation is represented as set of bindings to that plan's components.

For example, the WHILE action has an explicit index to the ITERATE-OVER-RANGE plan. The index's test is: *Does the the WHILE's test involves* *a single relational operator other than equality and contain a variable?* The index points to an instance of ITERATE-OVER-RANGE, with its LOOP-STATEMENT bound to the WHILE, its END-TEST bound to the relational test, and its INDEX bound to the variable. (If there are multiple ways to satisfy the index test, multiple instances of ITERATE-OVER-RANGE are generated.)

Each action or plan may index a set of other plans. For example, the WHILE action also indexes the FLAG-CONTROLLED-ITERATE plan. That index's test is: *Does the WHILE's test involve a single variable and an equality or inequality test against a numeric constant?*

The idea behind indexing is to perform index tests when each program action or plan is examined and to match only the indexed plans against the program. This focuses the search and has the potential to greatly reduce the search space. In addition, since the index is to a partially recognized plan, only the components not already recognized need to be searched for in the code, speeding up the matching process.

4.2 Specialization Constraints

Each plan may be linked to a plan it specializes. This link describes the additional constraints on it that cause it to differ from the plan it specializes. For example, the plan FULL-ITERATE-UP-OVER-RANGE is defined as a kind of ITERATE-OVER-RANGE whose UPDATE-EXPR is constrained to be an increment. Similarly, COMPUTE-SQUARE is defined as a kind of MULTIPLY, where both attributes of MULTIPLY are constrained to be equivalent expressions.

Each plan may be specialized by a set of different plans. ITERATE-OVER-RANGE, for example, is specialized by a set of plans, which includes FULL-ITERATE-UP-OVER-RANGE, FULL-ITERATE-DOWN-OVER-RANGE, ITERATE-UP-OVER-POWERS, and ITERATE-DOWN-OVER-POWERS, among others.

The idea behind specialization constraints is to allow plan recognition to proceed by following an index to a general plan and then to gradually specialize that plan. This simplifies indexing, since every plan does not require an index to be recognized, and allows plan recognition to take advantage of commonalities between different plans. But it means that whenever a plan is recognized, it's necessary to check all of the specialization constraints to the more specialized versions of that plan.

4.3 Implied Plans

Each plan may have a list of plans that are implied by its recognition. For example, recognizing COMPUTE-DISTANCE implies having recognized three new plans: two instances of POINTs and one instance of PAIR-OF-POINTS that combines these POINTs. These plans are not components of COMPUTE-DISTANCE, since they do not have to be recognized to recognize an instance of COMPUTE-DISTANCE. But once an instance of COMPUTE-DISTANCE is recognized, these other plans are now known to exist. Similarly, recognizing the specialization ARRAY-POINT of POINT (which is an implementation of point as two array elements with a shared subscript) implies having recognized another new plan: ARRAY-OF-POINTS (which has the two arrays as its components, but doesn't have ARRAY-POINT as a component).

The idea behind the list of implied plans is to avoid having to try to recognize these additional plans bottom up from the code. Essentially, it avoids matching them with the code by taking advantage of the fact that their existence is implied from plans that have already been recognized.

4.4 Summary Of Organization

In summary, each programming plan in the plan library is now indexed by particular combinations of program instructions or program plans. The program understander considers that a particular plan might be relevant only when it encounters the index during understanding—not simply whenever some component of the plan appears in the code. Each programming plan is also linked to specializations of that plan, with explicit tests to determine whether that specialization is present. Whenever a program understander recognizes a particular plan, it attempts to specialize it as far as possible. This avoids the need for explicit indexes to each individual specialization. Finally, each programming plan is also linked to other plans that can be inferred once its recognized. The understander can then recognize these plans without explicitly matching them against the code.

5 Plan Recognition Using This Plan Library Organization

The first step is the translation of the program into an abstract syntax tree with frames used to represent each action in the program and its relationship to other actions. An action is any unit the translator recognizes from language constructs, from low-level multiplication of a pair of values to higher-level constructs such as a loop with a test and a body. The contents of the frames and their slots vary with each type of action. The task of the recognition algorithm is to construct a plan hierarchy that maps entries from the plan library onto this abstract syntax tree.

5.1 The Algorithm

The recognition algorithm is straightforward. It proceeds by removing and examining the first component C in a program-component list L, which is initialized to the list of the entries appearing in the abstract syntax tree. The plan recognizer then does the following to try to infer the underlying plan hierarchy P:

1. Checks C's specialization constraints and attempts to specialize C as far as possible. It adds any specialization to P, with a specialization link to C.
2. Determines whether C or its specialization (if any) index any programming plans. If they do, it tries to match these indexed plans against the code. It places any programming plans that match into P, along with links to their components. In addition, it adds any recognized plans to the head of L, preceded by any plans that were implied by its recognition.

This process repeats until L is empty.

5.2 An Example

We'll briefly illustrate how this process recognizes the first few plans in our example. The recognition process starts by examining the declarations and assignments. None of these can be specialized, nor do they index any plans, so they wind up being quietly removed from L.

The recognition process next examines the outer `while` loop (the WHILE action in the program's abstract syntax tree). It first checks whether this action can be specialized (such as to INFINITE-LOOP, which is a WHILE with the additional constraint that its test is always true). In this case, however, none of its specialization constraints hold. The recognition process then checks whether any of the indexing tests associated with the WHILE succeed. In this case, it successfully indexes ITERATE-OVER-RANGE, so this indexed plan is added to the front of L.

The recognition process now examines ITERATE-OVER-RANGE. It checks whether it can be specialized; it can, to FULL-ITERATE-UP-OVER-RANGE. It then checks whether it or its specialization indexes anything. In this case it doesn't (although it would, for example, if it were contained in another iteration construct) and so it's removed from L. The result, at this point, is that the recognition process has determined that the outer `while`, the assignment to `j` that precedes it, and the increment of `j` inside it form an instance of FULL-ITERATE-UP-OVER-RANGE.

The recognition process then examines the assignment within the loop and finds it can't be specialized and doesn't index any concepts. It then encounters the inner `while` loop, which by the same process as before causes it to recognize another instance of FULL-ITERATE-UP-OVER-RANGE. As before, the recognition process then precedes to determine whether it can specialize this plan and whether it indexes any new plans. This instance indexes a new plan, GENERATE-PAIRS. (The index test is whether the FULL-ITERATE-UP-OVER-RANGE is contained within the SEQUENCE of another FULL-ITERATE-UP-OVER-RANGE.) This new plan is recognized, added to the front of L, and specialized to GENERATE-UNIQUE-PAIRS, which is another dead-end, and is eliminated from L. The result, now, is that the recognition process has realized the planning roles played by the structure of both loops and their loop indices.

The recognition process then examines the subtractions within the `while` loop, but they can't be specialized and don't index any plans, so they're quietly removed from L. It then processes the multiplications in the `sqrt`, specializing them to instances of COMPUTE-SQUARE, and processes the plus, which it specalizes to SUM-OF-SQUAREs. Finally, it processes the call to `sqrt`, which together with SUM-OF-SQUARES, indexes COMPUTE-DISTANCE. After matching COMPUTE-DISTANCE against the code, it then adds the plans it implies to the program-component list (the two POINTs and the PAIR-OF-POINTS) followed by the COMPUTE-DISTANCE plan itself. It then proceeds to specialize POINTs into ARRAY-POINTs, which in turn implies the existence of ARRAY-OF-POINTs, then continues in a similar fashion from there.

6 Conclusions

This paper has presented a modified bottom-up approach to the recognition of programming plans.

The approach relies on a highly organized plan library, where each plan has indexing, specialization, and implication links to other plans. It uses an algorithm that takes advantage of these indices to suggest general candidate plans to match top-down against the code, specializations to refine these general plans once they're recognized, and implications to recognize other, related plans without doing further matching. While others have pointed out the potential usefulness of indices (or "beacons") [1, 3], they have used them within a top-down approach to suggest plans other than the ones found by refining and decomposing a set of initial plans presumed to be in the code.

Our approach to hybrid bottom-up/top-down recognition is based on observations of student programmers understanding code. It has the potential to greatly reduce the number of plans that a program understander must try to match against the program. But by using indexes to make guesses about which plans might actually appear in the code, it is trading completeness (the ability to recognize every instance of every plan in the plan library that appears in the program) for efficiency (the ability to quicky recognize those plans it recognizes).

How well this approach will actually work in practice will depend significantly on whether it will be possible to accurately determine indexes to plans in the plan library and whether the necessary indexing, specialization, and constraint tests can always be implemented efficiently. Our approach to determining indices for our plan library has so far been to infer them from observations of programmers understanding programs, but this clearly is too time-consuming and too ad-hoc to be practical for real-world applications. As a result, we're currently investigating statistical methods for trying to automatically generate indices for plans. In addition, our approach uses a wide variety of different predicates to perform tests, some of which have the potential to be hideously inefficient when implemented in their full generality. Another current focus is in trying to construct a clean library of efficient predicates to use for making these tests.

Finally, we're currently testing our approach on a large library of geometric objects and operations. Our testbed is two sources: a set of sizeable student programs, and a set of smaller geometrically-inclined textbook programs translated into C.

References

[1] R. Brooks. "Toward a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, 18, 1983.

[2] F. Detienne and E. Soloway. "An Empirically-Derived Control Structure for the Process of Program Understanding", *International Journal of Man-Machine Studies*, 33, 1990.

[3] S. Fickas and R. Brooks. "Recognition In A Program Understanding System", in Proceedings of the 6th IJCAI, Tokyo, Japan, 1979.

[4] J. Hartman. "Understanding Natural Programs Using Proper Decomposition", in Proceedings of ICSE-91, Austin, TX, 1991.

[5] W. Lewis Johnson. *Intention Based Diagnosis of Novice Programming Errors*, Morgan Kaufman, Los Altos, CA, 1986.

[6] W. Kozaczynski, J. Ning, and T. Sarver. "Program Concept Recognition", in Proceeedings of the 7th Annual Conference on Knowledge-Based Software Engineering, Washington, DC, 1992.

[7] S. Letovsky. "Plan Analysis of Programs". Ph.D. Thesis, Yale University, New Haven, CO, 1988.

[8] S. Letovsky and E. Soloway. "Delocalized Plans and Program Comprehension", IEEE Software, 3(3), 1986.

[9] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance", in *Empirical Studies of Programmers*, E. Soloway and S. Iyengar (editors), Ablex, Norwood, NJ, 1986.

[10] W.R. Murray. *Automatic Program Debugging For Intelligent Tutoring Systems*, Morgan Kaufman, Los Altos, CA, 1988.

[11] J.Q. Ning. "A Knowledge-Based Approach To Automatic Program Analysis." Ph.d. Thesis. University of Illinois, 1989.

[12] C. Rich and R. Waters. *The Programmer's Apprentice*, Addison Wesley, Reading, MA, 1990.

[13] E. Soloway and K. Erdlich. "Empirical Studies of Programming Knowledge", *IEEE Transactions on Software Engineering*, 10(5), 1984.

[14] L.M. Wills. "Automated Program Recognition by Graph Parsing", Ph.D. Dissertation, MIT, Cambridge, MA, 1992.

[15] L.M. Wills. "Automated Program Recognition: A Feasibility Demonstration." *Artificial Intellgence*, 45(2), 1990.

Session F: What Does Program Comprehension Mean For Industrial People?

Loredanna Mancini
John Foster
Sylvie Cochinal
Alan Padula
Takis Katsoulakos
Roberto Ciampoli

What does industry need in order to understand programs? (Position Statement)

Loredana Mancini

O.GROUP - Olivetti Information Services
Via B. Croce, 19
00142 Rome

Abstract: The panel has the aim to discuss the different views on the meanings of program comprehension, both seen from the research and from the industry point of view. The participants will highlight the different perceptions and needs on the problem of understanding a software system and its maintenance.

Sometimes there seemed to exist something like a gap between industry and research. It is not clear if this is because industry people think that they are too busy in giving concrete results and cannot allow themselves to follow "dreams" like research, or because researchers are always looking for new interesting challenges and do not stop to create, out of their interesting results, something really usable in an industrial context.

Recently this tendency seemed to come to have been resolved, in fact research people seem to be tired of having only new "prototypes" that presented limits in their applicability in the real world, and they are looking forward to realising usable products. On the other hand there is a movement in industry that sees companies, at least the biggest and innovative ones, starting to ask for something that can help them in their everyday work, and that are also willing to experiment something that can be further exploited for their purposes.

This change is particularly important in disciplines, like program comprehension, that are strongly related with aspects connected with human perception and gathering of information. Human minds follow specific recognition patterns in order to comprehend software and the understanding and explicitation of these patterns should be one of the results of the Program comprehension research.

This panel wants to focus on topics like: "what does industry think program comprehension is?" and "Which are the industry needs to understand programs, and in general software systems?"

The participants will present their experience and their point of view on the topic, considering also the activities they are performing in this area. John Foster (British Telecom) will discuss the challenges that industry gives to research, Roberto Ciampoli (Olivetti) will present the industrial point of view on the application of program comprehension tools and techniques, Alan Padula (Hewlett Packard) and Silvy Cochinal (Cap Gemini Innovation) will give some hints on their experience in specific projects related to the theme discussed, and Takis Katsoulakos (Lloyd's Register) will present the extended experience he has acquired as project manager of a reverse engineering E.E.C. funded project: REDO

0-8186-4042-1/93 $03.00 © 1993 IEEE

An Industry View on Program Comprehension

John Foster
BT Laboratories
UK

Abstract

Industry needs program comprehension tools and techniques. It looks to researchers to provide them, but will cheerfully move the goalposts when they do.

1 Comprehend to Compete

If we cannot comprehend programs, we cannot reliably change them and the business that relies on them stagnates. Program comprehension enables *people* to provide a *rapid response* to *unexpected* requests for *change* on what are often very *large* systems.

Taking those keywords in reverse order:

Large: The complete works of William Shakespeare look trivial when laid alongside the source code of some industrial systems, yet we require programmers to understand both broad concept and fine detail of these huge works. No-one can hold so much information in their head. The sheer size also makes it unreasonable to expect "proper" documentation across the system: the future lies in providing automated broad coverage, with added human effort reserved for the detailed investigation of isolated parts.

Change: This is our most common reason for wanting to understand a program. It places great demands on a repository (as commonly used) because of the difficulty of relinking human-provided information after a major source change.

Unexpected: Most changes during the life of a system were unanticipated when the system was being developed. We cannot realistically expect the developers to document how changes might be made, because from their point of view the variety of possible changes is infinite. We also cannot deal with comprehension tools which expect that their users will interact with the tools to provide comprehensive documentation across the system. Without knowing in advance where the changes will come, we would waste an unacceptable amount of effort.

Response: A response to a request for change doesn't count unless it is an accurate response, with no new errors introduced. Comprehension methods and tools need to support desk-checking activities and inspection teams, as well as the maintainers. An accurate response means the program must fit with its application domain: domain knowledge is rightly understood to be a vital part of program comprehension, but we know little about how best to automate it.

Rapid: When a response is needed, it is needed fast. That, after all, is just part of being commercial. So there are more limits on how much we can afford to have the comprehension tool rely on human input for its information.

People: Comprehension is a human activity, and comprehension tools and methods only support it. People have their own ways of thinking and of problem-solving, and these are not well understood. Good comprehension tools are produced now, but far better ones should be available if we understood better how maintenance programmers work. We are wary of the results of studies carried out on novice programmers, and yet unwilling to spare the time of experienced staff for experiments. Either way, program comprehension research is a multi-disciplinary activity.

2 Towards a Solution

The ability to parse source code, to derive information on its structure and to present a variety of different views on the resulting data is exciting and welcome. It can be enhanced by extensions to the related studies of how people set about the problem solving process, and this research will feed on itself if that behaviour is itself modified by the new tools available. Coping with domain knowledge on a large scale remains a major problem, but researchers who manage to solve even that should comfort themselves: we in industry, as soon as we are able to cope with the systems we have, will soon produce even larger ones to tax the ability to understand.

0-8186-4042-1/93 $03.00 © 1993 IEEE

Program Comprehension: The Gap between Research and Industry

Sylvie Cochinal
Cap Sesa Sud-Ouest - Space Skill Center
8, rue Mesplé
31036 Toulouse Cedex, France

Introduction

The need for program comprehension in industry is now clearly recognized and its applications are various :

- maintenance (corrective, evolutive and adaptive) that represents more than 60% of the need,

- test and quality control, that, at its higher and more complex level, expects to check the mapping between the semantics of the program (what it really does) and the original functional specification,

- reuse, a new domain, that needs to know what a module does before classifying it and being able to reuse it in another application.

Various Needs for Program Comprehension

All these activities imply the comprehension of the whole or parts of the program. Depending on the activity and the features of the firm, the degree of comprehension is very different.

For example, high security aspects of some firms will demand a comprehension tool to give results that are sure at 100%. This criteria will then be prefered to a tool for a higher level of program comprehension that would understand more, but with a less high validity probability.

For other needs (mainly in maintenance), the user will expect the comprehension tool to be less precise but to give ideas of what and where to look in the code; this means to do a first filter on the enormous amoount of code in order to speed up the user's comprehension process.

Sometimes, industry will not need comprehension automatisation but "just" an ergonomic interface and storage capacity for stocking, query and retrieval of what users have already understood in the program.

The Gap between Industry and Research

Above, I remarked that industry "just" requires storage capacity and an ergonomic interface because it seems to be simpler than an automatic comprehension process. But in fact, research is not always well aware of the different needs of industry, and works more on some complex automated functional comprehension than on such basic "realizable" solutions that could be directly used in industry.

Moreover, as I have realized when passing from research in Cap Gemini Innovation to industrial applications at Cap Sesa, research uses as basic techniques for program comprehension some techniques that seem to be very advanced and not yet very fiable for industrial point of view. Knowledge bases, capture and modelisation of human expertise, expert systems and so on, that are used for program comprehension, frightens industry which sees these techniques as being too new to be applied for industrial purposes. Industrial application of program comprehension demands a high level quality and fiability that research prototypes don't have. And sometimes, the high complexity of the research result makes the industrialisation phase so difficult, that very often, it is abandoned due to the effort. The prototypes, then, stay on the shelf !

Conclusion

On one hand, there is great need of communication between research and industry to know what the second needs from the first one. Moreover, to be able to make the research results maybe less sophisticated and more reliable and easy to apply to a real industrial application.

On the other hand, research must always keep one step in advance on industrial needs and anticipate what industry will need in 10 years. This is the continual dilemma and the right dosage has to be found between responding to current needs and anticipating the future.

0-8186-4042-1/93 $03.00 © 1993 IEEE

Session G: Program Representations for Program Comprehension

Chair: Gerardo Canfora

A Toolset for Program Understanding

Panos E. Livadas
Computer and Information Sciences
University of Florida
Gainesville, FL 32611

Scott D. Alden
Computer and Information Sciences
University of Florida
Gainesville, FL 32611

Abstract

Program understanding is associated with the hierarchy of abstractions and interpretations that are deduced from the code [16]. Therefore, the understanding process parallels that of the bottom-up programming process in that maintainers begin by associating small groupings of individual instructions with higher-level interpretations. The understanding process is repeated until the desired level of understanding is attained.

Program understanding in this context requires the identification and study of the numerous complex interrelationships that are induced by the *data flow*, *calling*, and *functional* dependencies that exist in the software. Therefore, an environment is needed in order aid the programmer in understanding software. The *internal program representation* (IPR) plays a critical role in the nature of that environment.

In an earlier paper, we discussed both an internal program representation and an environment that conforms to the requirements stated above.[11] The internal program representation, the system dependence graph (SDG), is a directed labeled multigraph that captures all control and data dependences, as well as the calling context of procedures; it is based on the one proposed in [8]. The toolset is referred to as *Ghinsu* and it supports a number of tasks over a program written in a subset of ANSI C such as slicing, dicing, and ripple analysis.

In this paper we will present some background on the problems associated with program understanding and show how the *Ghinsu* toolset can aid the programmer in understanding software.

1 Introduction

Software maintenance is an expensive, demanding, and ongoing process. Boehm [3] has estimated that one US Air Force system cost $30 per instruction to develop and $4,000 per instruction to maintain over its lifetime. This case may be exceptional, but the maintenance costs for a large embedded system seem to be an average of two to four times the development costs. It is generally recognized that the primary reason that software maintenance is so costly is that each modification requires first and foremost that the software be *understood*. A program is said to be understood if an overall interpretation of the program is achieved. Most of the proposed models fall into one of the two categories: code-driven (bottom-up) or problem-driven (top-down) [4, 5, 15].

We hypothesize that both strategies are employed by the programmers engaged in this activity. We also support the notion that different maintenance tasks require different *kinds* of program understanding, and therefore different processes are required. As an example, consider the code segment below.

```
( 1).       sum = 0;
( 2).       read(next);
( 3).       while (next>=0) do
( 4).           begin
( 5).               sum = sum + next;
( 6).               read(next);
( 7).       /*          do_something(next); */
( 8).       end;
( 9).       write(sum);
(10).       write(next);
```

We can identify several different kinds of program understanding such as the following:

1. Understanding that the definitions of sum that can directly affect the use of sum at statement (5) are at statements (1) and (5).
2. Understanding that the definitions that directly affect statement (5) are at statements (1), (2), (5), and (6).

0-8186-4042-1/93 $03.00 © 1993 IEEE

3. Understanding that the value of `sum` at statement (9) depends on the statements (1), (2), (3), (5), and (6).
4. Understanding that the segment adds up a number of input values until it reads a negative value, after which it prints the result and the last value read.
5. Understanding that the author assumes that the source of input values is non-empty.
6. Understanding that the segment determines the sum of all scores in the recent exam of the class CS201.

This example illustrates the two principal domains of program understanding: the *programming* and the *application* domain. The first four kinds of understanding belong to the programming domain, and represent an extrapolation of the code's intent in terms of standard programming interpretations and problem solving techniques. The fifth kind belongs to the application domain, and differs from the rest because it represents an abstraction of the code's intent in terms of a specific application. This domain lies outside of the domain of program interpretations, and requires program documentation for understanding.

Program understanding is associated with the hierarchy of abstractions and interpretations that are deduced from the code [16]. Therefore, the understanding process parallels that of the bottom-up programming process in that maintainers begin by associating small groupings of individual instructions with higher-level interpretations. The understanding process is repeated until the desired level of understanding is attained.

Program understanding in this context requires the identification and study of the numerous complex interrelationships that are induced by the *data flow*, *calling*, and *functional* dependencies that exist in the software. Program segments are not just as simple as the example above may erroneously indicate. The example contains only localized interactions. As Letovsky and Soloway [9] have established, programmers have difficulty understanding code that has non-local interactions. For example, if the call to procedure `do_something` is uncommented, it is not clear which of the definitions of `sum` can reach the use of `sum` at statement number (5). The answer to this question depends on whether or not the variable `sum` is defined (as a global variable) in the body of the procedure `do_something`, or by some other procedure that `do_something` invokes (directly or indirectly) before it returns.

Given the complexity of the task, is not surprising therefore that programmers spend approximately 60% of the maintenance time "looking at" code [19]. Therefore one can conclude that maintenance quality and productivity can be improved by supplying the maintainer with a set of proper tools that he/she may employ for understanding the target software.

On the other hand, a number of organizations have found that simply purchasing new tools does not automatically increase productivity [1]. What is needed is the creation of a process for each type of understanding that uses a set of tools designed within the framework of this process. These tools should allow the maintainer to ask questions about a program and be provided with *precise* answers.

In order to study tool-assisted program understanding, we must provide an effective environment for understanding programs. Such an environment should be integrated with the existing software maintenance tools and should provide additional facilities to support other software engineering activities. We have already developed much of this environment, but some research issues remain. For example, the answers that the understanding tool provides should be presented to the programmer in a way that best improves the maintainer's understanding of the program relationships. This task is not trivial because of the large amount of dependencies in a software. Furthermore, the maintenance tools should have a fast interactive response time. Otherwise, the maintainers will be discouraged from using them.

Realizing the need for such an environment, two and half years ago we embarked upon the task of developing such an environment and tools and turning the theoretical concepts into practical realities, with the support of the Software Engineering Research Center[1]. We have made considerable progress towards these goals.

The key element of our system is its internal program representation (IPR), the *System Dependence Graph*, or SDG. The SDG is a parse tree representation of the program. The nodes represent program constructs, in and out parameters, call-sites, etc. The edges represent various kind of dependencies (such as data flow, control flow, and declaration) among the nodes to which they are adjacent. The main benefit of this structure is that it represents a vast amount of information that could be shared by numerous software engineering tasks. These applications typically use the

[1]Funded by the National Science Foundation, the Center's 15 industrial affiliates, and the Florida High Technology and Industry Council.

same kind of information, but use different representations. Our approach eliminates the redundancies. Since all algorithms use the SDG as the underlying structure, they are source language independent.

The environment, referred to as *Ghinsu*, supports a number of tasks such as program slicing, dicing, ripple analysis, dependency analysis, DU-chain, UD-chain, and reaching definitions calculation as well as a host of browsing activities.

The remainder of this paper is organized as follows. The next section presents background on slicing, dicing, and ripple analysis. We then briefly describe the internal program representation and how it is derived. Finally we present a tour of the *Ghinsu* toolset and show some of its major functions and tools.

2 Background

From our perspective, the most important concept is that of a static slice, since it is used to build the SDG. In addition, a slicing tool can provide useful information for the software maintainer.

Slicing provides a way to decompose a large program into smaller, independent components. Let P be a program, p be a statement in P, and V be a subset of the variables of P. Weiser defines as a *static slice* of P relative to the *slicing criterion* $< p, V >$ to be the set of all statements and predicates of P that *might* affect the values of variables in V at the statement p. Weiser reports experimental results that experienced programmers use slicing when debugging [17]. Weiser found that programmers remembered the slice relevant to the bug as having been used or probably having been used in almost half the cases examined. When debugging, programmers view programs in ways that need not conform to the program's textual or modular structures. In particular, the statements in a slice may be scattered throughout the code of the larger program. Yet, experienced programmers routinely abstract these slices from a program. Weiser concluded that since programmers remembered the relevant slices from the program they had debugged, they were probably mentally constructing and using these slices while debugging. Presumably each programmer had independently developed the slicing method. If novice programmers were taught the concept of slicing, they could avoid this reinvention and learn debugging techniques faster.

Since debugging is a process in which programmers try to better understand code to find and eliminate bugs, and since programmers find slices when debugging, it is logical that a tool that automatically creates program slices would be useful not only in debugging but also in code understanding [17].

Suppose that during testing, we find that the value of a certain variable, v, is incorrectly computed at statement n. By obtaining a slice of v at n, we may extract a significantly smaller piece of code than the entire program in which to locate the bug. If the value of another variable, w, is computed correctly at statement n, then we may employ a method that was suggested in [13] and is referred to as *dicing* (computing the intersection of two slices). The bug is likely to be associated with one or more statements in a set referred to as the *Fault Prone Statement Set (FPSS)*, which is the set of statements associated with the slice on v minus those associated with the slice on w. The FPSS is obtained by generating the complement of the slice on w relative to the slice on v.

If a large program computes the value of a variable x and the code associated with this function is needed in another application, then one could slice on this variable and use the extracted program in the latter application. Therefore, program slicing aids in code reuse.

The number of slices, their spatial arrangement, etc., may hold significant information about the structuring of a program [17]. Hence, an assortment of *program metrics* can be computed and their actual significance investigated. Useful metrics include coverage, overlap, clustering, and tightness.

Ripple analysis identifies the statements that will be affected when a change is to be made at a given statement (i.e, ripple analysis is "forward" slicing). A program maintainer can examine the ripple of a statement to help determine the possible effects of a proposed modification.

Structured walkthroughs and code inspection activities would be easier to perform by calculating inter-procedural reaching definitions (the set of statements s' which reach a statement s) , *DU-chains* (a chain that links a use to all definitions that may reach it), and *UD-chains* (a chain that links a definition to all of its possible uses).

Furthermore, run-time support can be provided through *automatic data generation* (by using the calculated UD and DU-chain information). *Dynamic slicing*, and other pertinent tools that can be built by using the SDG to instrument the generated code. Most of our tools can be run on incomplete programs provided that they are compilable. Hence, these tools can be used even at the development stage.

We have also developed an *object finder* that uses information from the SDG to group together related

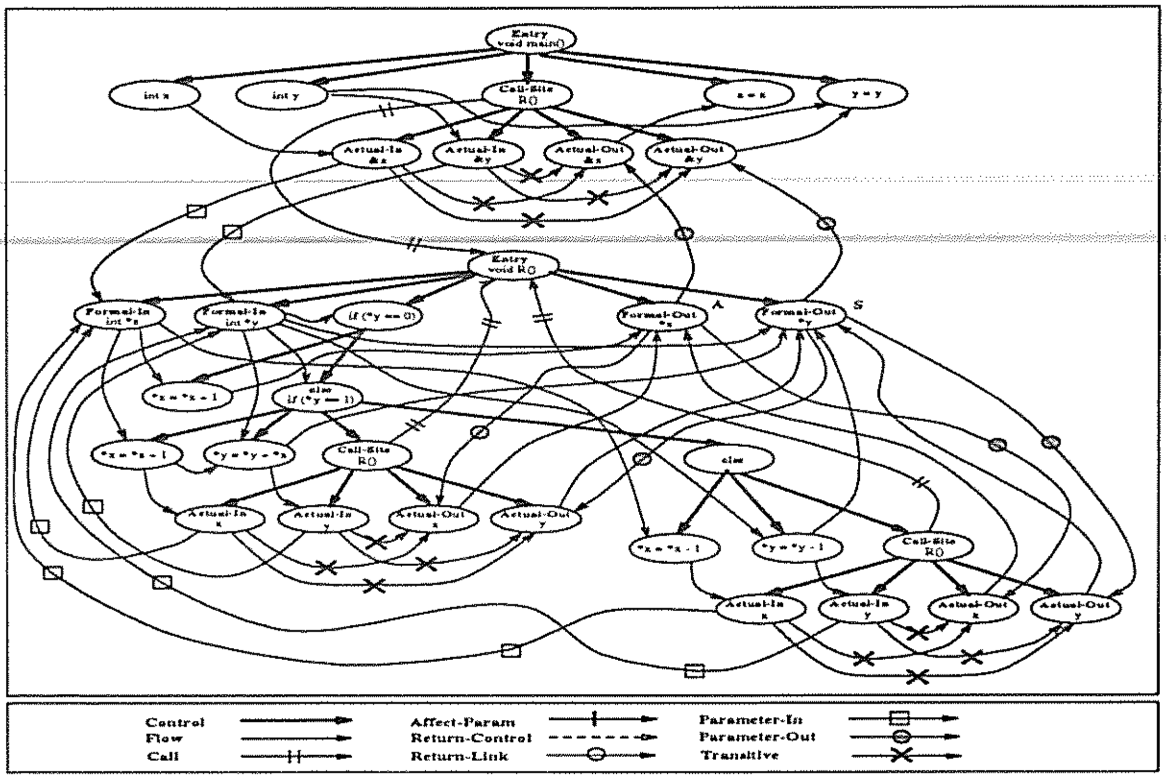


Figure 1: The SDG corresponding to the code in Figure 4.

types, data, and routines [10, 12]. We note that this tool can help objectify code and capture the objects that the original designer had in mind.

3 The Internal Program Representation

Weiser's slicers [18] were based on a flow-graph representation of programs. Ottenstein et al. [14], show that an *intraprocedural* slice could be found in linear time by traversing a suitable graph representation of the program that they referred to as the program dependence graph (PDG). Horwitz et al. [8] introduce algorithms to construct *interprocedural* slices by extending the program dependence graph to a supergraph of the PDG, which is referred to as the system dependence graph (SDG). This extension captures the calling context of the procedures which was lacking in the method proposed by Weiser.

This new approach not only permits more precise slices than [18], it also permits slicing when the program contains calls to *unknown* procedures (procedures whose bodies are not available), provided that the *transitive dependencies* (discussed later) are known. As was pointed out in [14], the internal program representation (IPR) chosen plays a critical role in the software development environment. An example of a SDG is shown in Figure 1

We have developed a prototype that accepts programs written in ANSI C or Pascal and generates a *parse tree based* SDG. We have implemented tools such as a slicer, dicer, ripple analyzer, dependency analyzer, DU-chains, UD-chains, a reaching definitions calculator (even if these definitions or chains span procedure boundaries), and a browser that utilize this SDG.

The grammar proposed in [7] consists of a single (main) program and supports scalar variables, assignment statements, conditional statements, and while loops, but does not support variable declarations. The language consists of a collection of procedures whose parameters are passed by value-result, and which end with `return` statements. These `return` statements can not be arbitrarily located in the procedure, nor can they actually return values to their calling procedure(s).

We extend this grammar and consequently modify the SDG as follows: First, variable declarations are supported. Second, we distinguish between pass-by-value and pass-by-reference parameters. The same

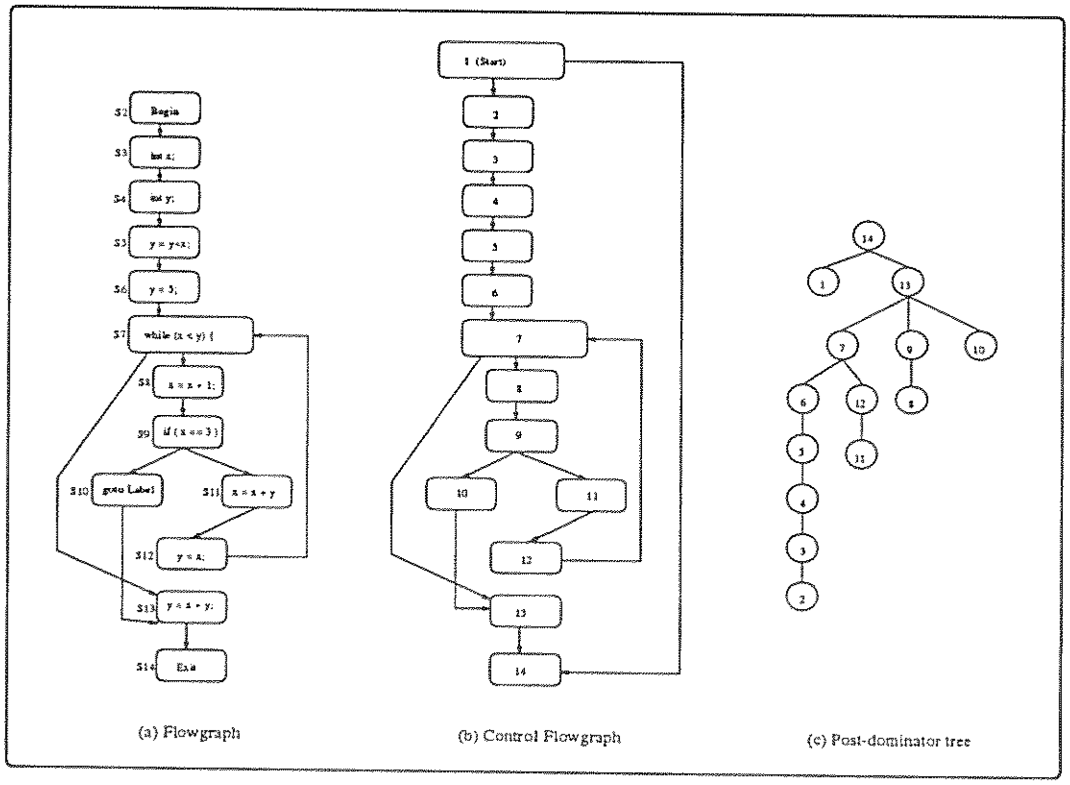


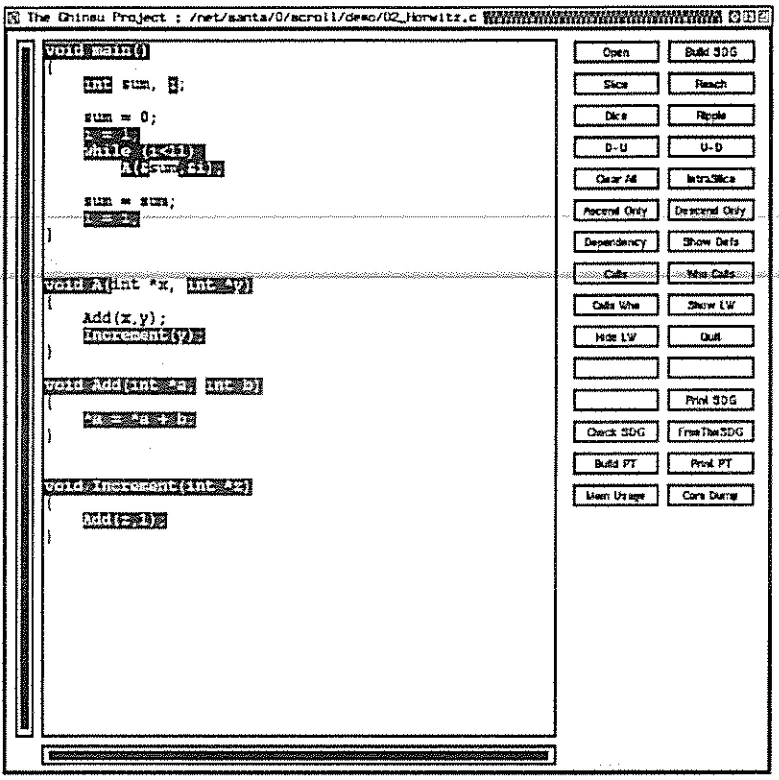
Figure 2:

notation is employed as in C, in order to determine the type of parameter passing. However, pointer operations are restricted to those that constitute pass-by-reference parameters; i.e., if `x` and `y` are pointer variables, we permit assignments of these variables such as `*x = 4` and `*y = *x` (where `*` denotes a de-referencing of the contents of the variables). Third, any number of `return` statements are permitted to appear anywhere in a procedure. These `return` statements can contain expressions that may include variables and are modeled after the `return` statements in C. Fourth, we distinguish among functions that return values as opposed to those that do not. Fifth, all C constructs are handled except `long jumps`. Finally, we use a *parse tree* as the basis of our SDG. This allows slicing to be more precise than if the "resolution" of the SDG was only at the statement level.

Even though the SDG and the slicing algorithm are based on the work proposed in [8], our methods are considerable extensions of previous works. First, our grammar is a superset of the grammar targeted in [8]. Second, our method of building the SDG differs in many respects. Our method eliminates the need to compute the GMOD and GREF sets of each procedure in the system and to construct a linkage grammar and its corresponding subordinate characteristic graphs of the linkage grammar's nonterminals. Third, we use a *parse tree* as the basis of our SDG. This allows slicing to be more precise than if resolution of the SDG was only at the statement level. The improved precision occurs because the algorithm for slicing [6] requires the traversal of certain edges backwards. Hence, when a statement such as `x = x + y +foo(&a)` is encountered during the computation of the slice of `a`, then the union of the slices associated with `x` and `y` will be included in the slice of `a`. It is also clear that variables `x` and `y` do not affect the value of `a`. By employing a parse-tree-based SDG, we are able to avoid this shortcoming of statement-based SDGs and therefore arrive at more precise slices and therefore more precise data dependence analysis (since the latter depends on the former).

3.1 Control and Data Dependence

Control Dependence A flowgraph (program graph) is a directed graph with an initial node from which all other nodes can be reached. Nodes correspond to basic blocks and edges represent transfers of control between basic blocks. Figure 2(a) shows an example flowgraph whose initial node has been marked


The Ghinsu Project : /net/santa/0/scroll/demo/02_Horwitz.c

```
void main()
{
    int sum, i;

    sum = 0;
    i = 1;
    while (i<11)
        A(&sum,&i);

    sum = sum;
    i = i;
}

void A(int *x, int *y)
{
    Add(x,y);
    Increment(y);
}

void Add(int *a, int b)
{
    *a = *a + b;
}

void Increment(int *z)
{
    Add(z,1);
}
```

Figure 3: The slice relative to the statement `i=i`

by "Begin." Dependences among blocks arise as the result of either control or data dependences.

A node y is said to be control dependent on x if there exists a directed path from x to y such that every node z on the path (not including x or y) is post-dominated by y, and y does not post-dominate x.

In order to calculate control dependences, a control flowgraph of a program is needed. The control flowgraph's post-dominators are then calculated which is equivalent to calculating the dominators of the reverse control flowgraph (all edges of control flowgraph are reversed). We then use the algorithm discussed in [6] to compute the control dependences.

Figure 2(b) and 2(c) show an example of the control flowgraph and its corresponding post-dominator tree.

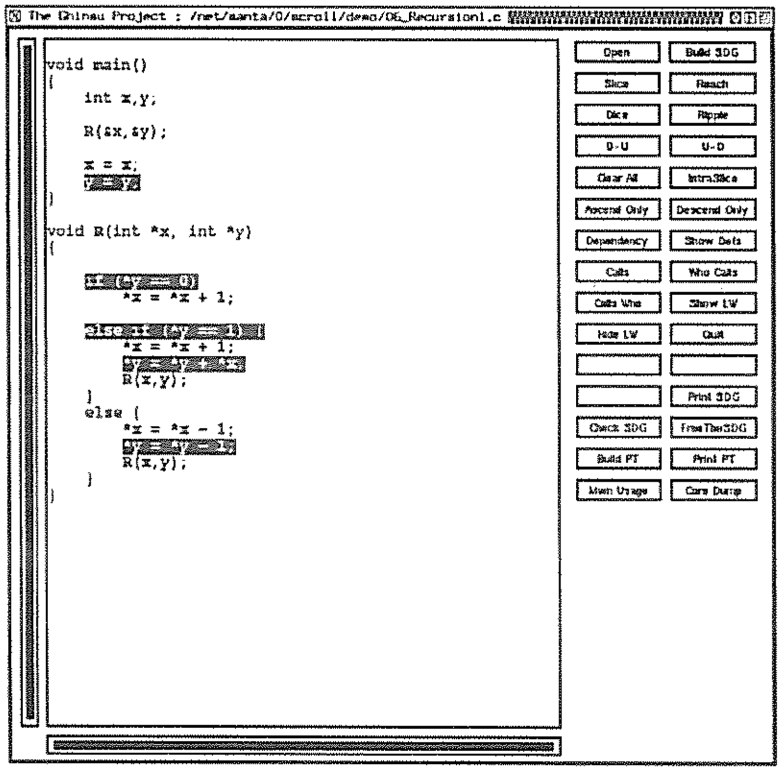
Data Dependence In order to compute the data dependences of a program, we must first calculate the reaching definitions for the entire program. We define the *GEN* and *KILL* sets [2] for each block on flowgraph. We then use the iterative algorithm presented in [2] to calculate the reaching definitions. A node x is data flow dependent from a node y if node y defines a variable that is used in em x.

We now consider the case of routine invocation. When a call-site is encountered, the flowgraph is annotated by introducing actual_in and actual_out nodes ([11]). The actual_out nodes are considered as unknown (U-nodes)[11] until the time the called procedure is solved. The assignment of the actual_out nodes depends on the corresponding formal_out nodes. Specifically, if a formal_out node is an A-node, we consider its corresponding actual_out node to be a definition. If it is an I-node, the actual_out node is considered to be a definition, but its *KILL* set is defined to be empty. Finally if the formal_out node is an IN-node, then both its *GEN* and *KILL* sets are empty by definition.

When we encounter a call-site, we suspend solution of the current procedure and descend into the called procedure and calculation of the reaching definitions is begun there. This procedure is repeated until one procedure calls no other procedure. At this point, all of the data flow dependences of the last encountered procedure can be calculated since its *GEN* and *KILL* sets are known.

4 A Tour of Ghinsu

In this section we briefly present the Ghinsu environment and the tools that we have implemented.



```
void main()
{
    int x,y;

    R(&x,&y);

    x = x;
    y = y;
}

void R(int *x, int *y)
{

    if (*y == 0)
        *x = *x + 1;

    else if (*y == 1) {
        *x = *x + 1;
        *y = *y + *x;
        R(x,y);
    }
    else {
        *x = *x - 1;
        *y = *y - 1;
        R(x,y);
    }
}
```

Figure 4: The DU-chain at statement `*y=*y+*x`.

Ghinsu accepts a source program written in a subset of either ANSI C or Pascal as input and produces the SDG as described earlier. This SDG can subsequently be used by any of the available tools.

Figure 3 presents a simple graphical user interface that we developed using X-Window library routines that facilitates user interaction with the system. A brief description of the major components of Ghinsu as well as the tools discussed earlier follows. It should be noted that except for YACC all components were built "from scratch".

YACC: The parser generator YACC is used in the Ghinsu project to generate a parser that when fed an ANSI C or Pascal source program produces a parse tree as output. This step produces the nodes and control flow edges and is the skeletal structure on which the rest of the system dependence graph is built. Each terminal node is annotated with its corresponding location in the source (a line and column number). This allows us to achieve a mapping from the source file to/from the SDG.

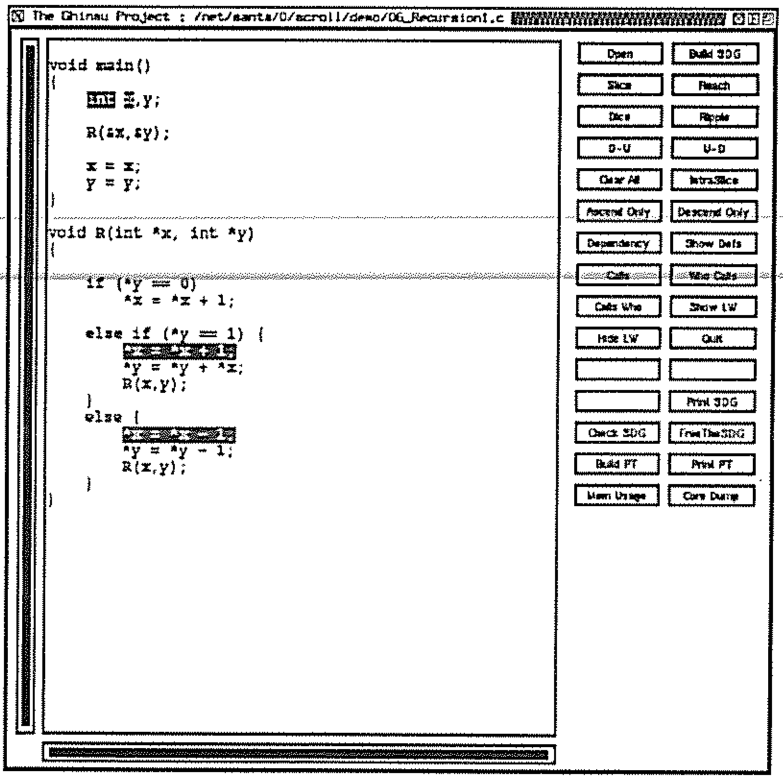
Dependency Generator: The dependency generator takes the parse tree (generated by the YACC) as its input and produces the parse tree based system dependence graph. Figure 1 illustrates the statement[2] based SDG produced by the dependency generator for the program shown in Figure 4.

TOOLS: The tools reside here and will be discussed shortly.

Graphical Interface: After the Ghinsu tool has been invoked, the user is presented with a window (not shown) where the files of the given subdirectory are displayed. The subdirectory can be changed by changing the `Path` field in this screen. The file that contains the desired program can be selected by clicking on its filename and subsequently can be opened via `open` button. Before any of the tools are invoked the user must request that the SDG corresponding to the file opened should be built; this is accomplished via the `build SDG` button. At this point, the user must position the cursor on the target statement (and variable) that he/she wishes to inspect; then he/she should invoke the appropriate module (slicer, dicer, ripple analyzer, dependence analyzer, DU-chain, etc.)[3].

[2] We illustrate the statement based SDG as opposed to the parse tree based for the sake of simplicity.

[3] The `Clear All` button is employed to clear all highlighted text; the `Quit` button is used to exit from Ghinsu. Buttons that



```
void main()
{
    int x,y;

    R(&x,&y);

    x = x;
    y = y;
}

void R(int *x, int *y)
{

    if (*y == 0)
        *x = *x + 1;

    else if (*y == 1) {
        *x = *x + 1;
        *y = *y + *x;
        R(x,y);
    }
    else {
        *x = *x - 1;
        *y = *y - 1;
        R(x,y);
    }
}
```

Figure 5: The UD-chain at statement `*x=*x-1`.

The mapping between the source code, the graphical display, and the SDG is straightforward. When the user selects some variable on a statement to have some action performed on it (such as a slice), the line and column is determined. The SDG is then searched for a match based on the line and column. If a match is found, the variable is highlighted; and, its corresponding node is "remembered". If the user subsequently chooses an action, the node remembered is used as the target node. The results of the action are reflected on the display by traversing the SDG and highlighting the source corresponding to the nodes that are marked (e.g., in the slice).

Slicer: This module calculates slices on a system dependence graph. The screen dump shown in Figure 3 illustrates the slice of the program relative to the statement `i=i` by highlighting the statements that belong to that slice. In this context, the maintainer may use the *intraprocedural slice* button whenever he/she wishes to limit his/her view to the scope of one function. Furthermore, two more buttons related to slicing are provided. The *ascend only* tool allows the maintainer to limit the slice to only the function selected and the functions that call the selected function. This operation corresponds to slicing phase one only. Correspondingly, the *descend only* tool allows the maintainer to limit the slice to only the function selected and the functions called by the selected function. This operation corresponds to slicing phase two.

DU-chain: Two more screen dumps are displayed. Both illustrate the *precision* as well as the identification of the *position* of the uses and definition of variables at a given statement even if procedure boundaries are crossed and even if recursive procedures are present. In Figure 4 the statement `*y=*y+*x` has been selected and the DU-chain has been requested, i.e., the determination of all uses for this definition of `*y`. Notice that `*y` is used in the predicate `(*y == 0)`; if this statement evaluates to false, it will be used in the predicate `(*y==1)`. Furthermore, if the latter predicate evaluates to false, then the variable will be used at the statement `*y=*y-1`. Finally, since there is an execution path that passes through the statement `*y=*y+*x` and statement `y=y`, the latter statement is captured since `y` is used there.

Similarly, Figure 5 illustrates the definitions of the variable `x` that can reach its use in statement `*x=*x-1`. In that case what reaches this statement is either the

are not discussed are used for the tool's development process. Additionally, the `object finder` button is not shown.

declaration (ud-anomaly can therefore be detected) or the statement containing the definition `*x=*x+1` depending of course on the data.

Calls: This button invokes a tool that displays the calling sequence.In addition, the user could query the system via either the *who calls* or *calls who* buttons. Specifically, the maintainer selects (via the cursor) a function such as `sample`. In the former case, all functions that call the function `sample` will be identified whereas the functions that are invoked by the function `sample` will be identified.

Dependency: This button invokes the data flow dependence analyzer. It is assumed that already a statement has been selected as we described earlier.The output indicates the line number, variable name, type of variable and the function in which each variable that may affect the value of the selected variable is visible.

Finally, the *Show definitions* selection causes all statements to be identified at which a maintainer-specified variable has been defined.

5 Acknowledgements

We wish to acknowledge Steve Croll for the implementation of the *Ghinsu* Toolset.

References

[1] SERC Industrial Affiliates. Personal communication, 1989-1992.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass.

[3] B.W. Boehm. The High Cost of Software, Practical Strategies for Developing Large Software Systems, E. Horowitz (ed.). *Addison-Wesley* Reading, Mass.

[4] V. Basili and H. Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, SE-8(3):270-283, 1982.

[5] R. Brooks. Towards a theory of the comprehension of computer programs. *Int'l J. Man-Machine Studies*, 18:543-554, 1983.

[6] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319-349, 1987.

[7] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proc. 15th ACM Symposium of Programming Languages*, 1988.

[8] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, January 1990.

[9] S. Letovsky and E. Soloway. Strategies for documenting delocalized plans. In *Proc. Conf. on Software Maintenance*, pages 144-151, 1985.

[10] P.E. Livadas and P. Roy. Program dependence analysis. In *IEEE Conf. on Software Maintenance*, 1992.

[11] P.E. Livadas and S. Croll. System Dependence Graphs Based on Parse Trees and their Use in Software Maintenance. In *Journal of Information Sciences*, (to appear).

[12] P.E. Livadas and T. Johnson. A new approach to finding objects in programs. Technical Report cis.ufl.edu:cis/tech-reports/tr92/tr92-037.ps.Z, U. Florida Dept. of CIS, 1992.

[13] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proc. 2nd International Conference on Computers and Applications*, 1987.

[14] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. ACM SIGPLAN Notices 19,5, 1984.

[15] R.E. Seviora. Knowledge-based program debugging systems. *IEEE Software*, 4(3):20-32, 1987.

[16] J. Wedo. Structured program analysis applied to software maintenance. In *Proc. Conf. on Software Maintenance*, pages 28-34, 1985.

[17] M. Weiser. Programmers use slices when debugging. *CACM*, 1982.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 1984.

[19] N. Wilde. SDTC Lecture Series 1: Software Development Environments. CENET, October 1992.

A Combined Representation for the Maintenance of C Programs *

David Kinloch Malcolm Munro
Centre for Software Maintenance
University of Durham
South Road, Durham DH1 3LE, UK

Abstract

An important aid to the problems of program comprehension has been the use of static analysis tools to provide useful and up to date information on a program. Through the use of different views a maintainer can gain a much clearer understanding of a program.

A drawback of static analysis tools is that various representations of the code are required to construct the different views of the program. A solution is to devise a single combined representation containing sufficient information to allow construction of each required view.

This paper describes research to extend an existing unified interprocedural graph to allow the representation of C programs. Techniques for the dependence analysis of pointer variables are described and the construction of interprocedural definition-use information in the presence of pointer parameters addressed. A fine grained program representation, the Combined C Graph (CCG), containing three new edge types is introduced.

1 Introduction

Whenever a change is to be made to a piece of software, it is important that the maintainer gains a complete understanding of the behaviour and functionality involved. This process of program comprehension is frequently made more difficult by the absence of useful program documentation. The maintenance programmer will often not have been involved in the development process or a significant period of time may have elapsed between development and maintenance. Documentation will then become of crucial importance.

In many cases the maintenance programmer's only reliable description of the software will be the source code itself. The process of program comprehension can then be aided by the use of static analysis tools. These tools are able to analyse a program without its execution to extract information such as call graphs, control flows, data flows, program slices and cross references. By integrating static analysis tools, software maintenance environments have been developed. The source code is analysed and the resulting information stored for later browsing by the maintainer. Multiple views of the program may be presented, helping the maintainer by providing information in more than one form and by concentrating the maintainer's attention on relevant parts of the software.

Harrold and Malloy[1] identify the problem that different representations of the source code are required to construct each view of the program. For example, a representation allowing the computation of interprocedural data flow information may be inadequate to permit the construction of a program slice. A maintenance environment providing different views of the code will therefore require a variety of independent program representations and algorithms. These representations will contain repeated information and hence will lead to inefficient use of storage space.

The solution to this problem presented by Harrold and Malloy[1] is to combine the features of existing program representations to give a single combined representation. The resulting representation will contain sufficient information to construct each of the required views of the program. A single combined representation has the following three advantages.

- Eliminates redundant information.
- Reduces access times to different representations. Any algorithm need only access one representation.
- Helps comprehension by incorporating all pro-

*This research is supported by the Science and Engineering Research Council and by the European Gas Turbines Engineering Research Centre (Whetstone).

gram relationships into one representation.

Various existing program representations and the resulting Unified Interprocedural Graph are described in section 2.

A limitation of this earlier work is the language features which may be represented by the unified graph. Only scalar variables, assignment statements, while loops, conditional statements and procedures with reference parameters are permitted. The aim of this research is to devise a Combined C Graph (CCG) to represent programs written in the C language[2]. This presents a variety of additional problems. C contains many additional language constructs such as pointer and structure variables and value-returning functions with pass by value and pointer parameters.

An important feature of a combined graph is its representation of the program's data dependencies. Data dependence information is used to calculate definition-use relationships and program slices. This paper addresses the problem of calculating data dependencies in C where dynamic memory allocation and pointer-induced aliasing create extra difficulties. Aliasing information must be taken into consideration in order to achieve accurate static analyses of the source code. Section 3 describes the problems involved and discusses a method for data dependence analysis in the presence of pointer variables and dynamic memory allocation. An example is outlined and the application of this technique to pointer parameters and the calculation of interprocedural data dependencies addressed.

The C language also allows expressions involving embedded side effects, where a variable is defined as a result of the evaluation of the expression, and embedded control flow. These features require a more refined analysis of each program statement. Section 4 describes the problems involved and introduces a fine grained graph representation with three new edge types.

Concluding remarks are given in section 5.

2 Existing program representations

To provide a maintainer with different views of a program a variety of algorithms are required. Algorithms such as those to calculate data flow information and program slices often involve the use of intermediate graph-based program representations. This section describes a number of these representations.

A simple representation of a program is the call graph. Nodes of the graph represent procedures, edges represent call relationships between procedures and edge labels represent actual parameters. Whilst the call graph itself provides important information for program comprehension, it is evident that it contains insufficient information for the creation of other program views.

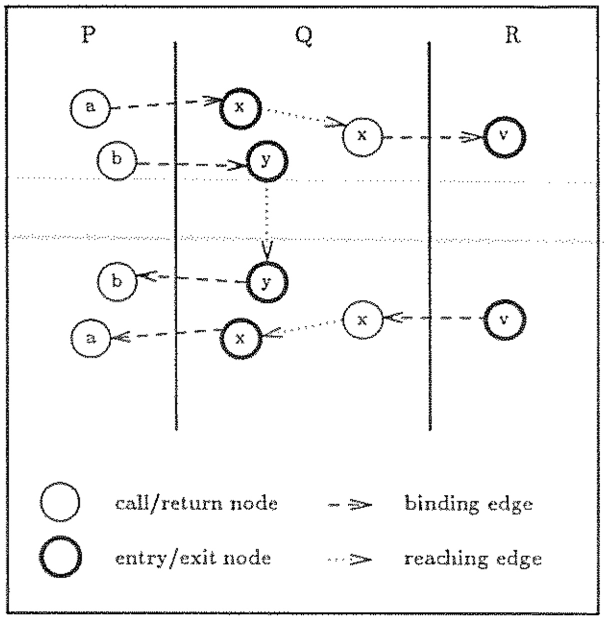


Figure 1: Example Program Summary Graph.

A variant of the call graph, the Program Summary Graph (PSG) presented by Callahan[3] permits flow-sensitive data flow analysis, i.e. control flow internal to procedures is taken into account. The PSG represents programs written in a procedural language with call by reference parameters. Call information is represented by nodes for each formal and actual reference parameter at procedure call, entry, exit and return sites, together with binding edges relating these nodes. Control flow within each procedure is represented by 'reaching edges' between the interprocedural control points. Callahan presents iterative algorithms to solve a variety of data flow problems, for example whether a reference parameter may be preserved across a call site. Figure 1 shows an example PSG. Procedure P contains a call to Q with actual parameters a and b. These are bound to the formals x and y. Q in turn calls R with actual parameter x bound to v within R.

An extension to the PSG, the Interprocedural Flow Graph (IFG)[4] allows the calculation of interprocedural definition-use pairs. Intraprocedural reachable use sets are first attached to each entry and return node. A two phase algorithm then allows this local information to be propagated throughout the graph,

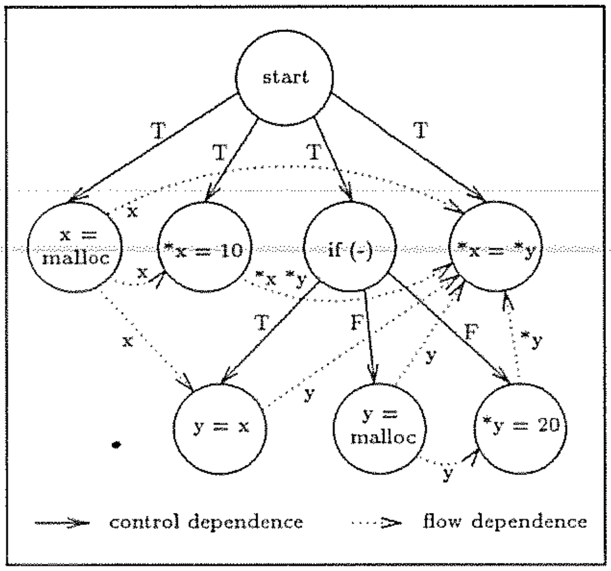


Figure 2: Example Program Dependence Graph.

making use of a new 'inter-reaching edge' to preserve calling context. The resulting sets of interprocedural reachable uses at each IFG node can then be used to calculate definition-use associations.

The Program Dependence Graph (PDG) was first introduced by Ferrante et al[5] as an internal representation for an optimising compiler. Its use in software development and maintenance has been addressed by Ottenstein and Ottenstein[6]. The PDG consists of nodes representing program statements and edges representing control and data dependencies between these statements. An example is shown in figure 2.

Ottenstein and Ottenstein present a linear time program slicing algorithm. A slice at statement s is computed by determining each statement on which s has a transitive data or control dependence. This involves a simple backwards traversal of the PDG from statement s.

An extension to the PDG, the System Dependence Graph (SDG) is described by Horwitz et al[7]. Horwitz et al tackle a language comprising scalar variables, assignment statements, conditional statements, while loops and an output statement, together with procedure calls and pass by value-result parameters. Dependence subgraphs are created for each procedure and are connected by new nodes and edges making up the call interface. Nodes are introduced to represent procedure call and entry sites together with the actual and formal parameters. Additional edges represent procedure call relationships and binding between actual and formal parameters. Figure 3 shows an example of the call interface for the call of Q from P shown in figure 1.

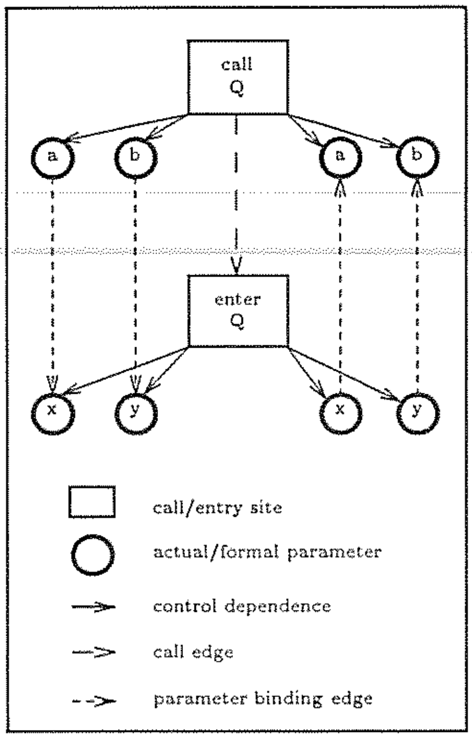


Figure 3: Example System Dependence Graph call interface.

A two phase interprocedural program slicing algorithm, again based on a backwards traversal of the graph, is presented. A new 'interprocedural transitive data dependence edge' is used to preserve calling context during this traversal and hence eliminate redundant nodes from the computed slice.

Given the useful information provided by the call graph, the applicability of the PSG and IFG to data flow calculations and of the SDG to interprocedural program slicing, Harrold and Malloy[1] identify the possibility of combining these representations to create the Unified Interprocedural Graph (UIG). Various redundancies between the nodes and edges of each individual representation are identified. For example, the call and return nodes of the PSG/IFG are equivalent to the actual parameter nodes of the SDG. Similarly the entry and exit nodes of the PSG/IFG are equivalent to the SDG's formal parameter nodes. Both the call graph and SDG contain edges representing calling relationships between procedures. In each case the redundant information can be removed and consequently savings made in the storage space required. Algorithms applicable to each graph remain applicable to the UIG by simply considering only subsets of the available nodes and edges.

3 Dependence analysis of pointer variables

When statically analysing a program the effect of aliasing must be taken into account. The precision by which these effects can be determined will be a significant factor in the usefulness of the static analysis. This is especially true of data dependence analysis, where the presence of aliasing creates additional dependencies. Imprecise aliasing information will lead to further spurious dependencies.

An important component of the UIG is the flow dependence. For a language without aliasing Horwitz et al[7] define a flow dependence from vertex v_1 to vertex v_2, $v_1 \rightarrow_f v_2$, to exist when:

- v_1 is a vertex that defines variable x.
- v_2 is a vertex that uses variable x.
- Control can reach v_2 after v_1 via a path in the control flow graph along which there is no intervening definition of x.

This corresponds to the computation of a definition-use association from v_1 to v_2. The UIG contains edges to represent intraprocedural flow dependencies.

For languages without pointer variables, aliases can be created through the use of reference parameters and non-local variables. An alias will result in procedure P whenever a call is made to procedure P of the form $P(x, x)$, where the actual parameter is repeated, or $P(g)$, where g is a non-local variable accessed within P. Further aliases can then be created within procedures called from P by calls such as $Q(a, b)$, where a and b are aliases within P. Any alias created will hold throughout the execution of a callee. Horwitz et al extend the original definition of flow dependence to deal with the possibility of 'potential aliases' which exists in a language with reference parameters whenever a procedure has two or more actuals of the same type, two or more non-locals of the same type or an actual and non-local of the same type:

- v_1 is a vertex that defines variable x.
- v_2 is a vertex that uses variable y.
- x and y are potential aliases.
- Control can reach v_2 after v_1 via a path in the control flow graph along which there is no intervening definition of x or y.

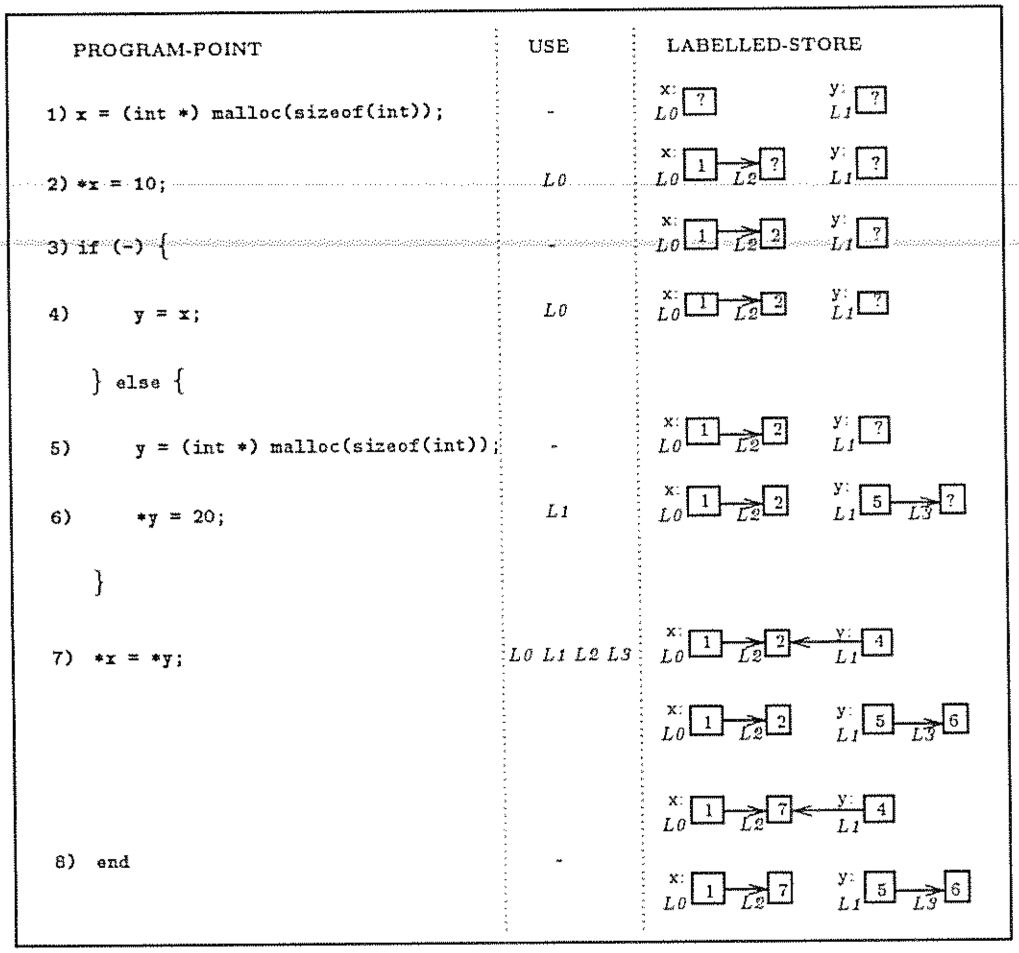
The definition presented by Horwitz et al for definition-use associations in the presence of aliasing relies on the fact that an alias holding when a variable is defined will also hold when that variable is used. There is a direct relationship between variable names and memory locations which does not change intraprocedurally. This approach, although not discussed by Harrold and Malloy[1], could be applied to the reference parameters of the UIG.

In C, assignments between pointer variables allow aliases to be created intraprocedurally and allow the aliases holding in a caller to be affected by assignments within a callee. Aliasing information can no longer be calculated for an entire procedure. Recent work by Landi and Ryder[8][9] presents an algorithm to safely approximate interprocedural pointer-induced aliasing, based on the use of conditional analysis techniques. Pande et al[10] extend this work to give an approximate algorithm for obtaining interprocedural definition-use associations in the presence of single level pointers. Pande et al first calculate the interprocedural reaching definitions, again using the conditional analysis technique. The aliasing information computed by Landi and Ryder is then used to account for the generation and killing of reaching definitions through aliasing effects.

An alternative method for the calculation of data dependencies for programs with pointers and heap allocated storage is presented by Horwitz et al[11]. Horwitz et al address the reaching definitions problem in terms of memory locations, rather than variable names and aliases:

> Program-point q has a flow dependence on program-point p if p writes into a memory location *loc* that q reads, and there is no intervening write into *loc* along the execution path by which q is reached from p.

Horwitz et al's algorithm is divided into two phases. The first phase, the 'reaching-stores phase', uses an abstract semantics to compute at each program-point a set of store graphs that approximate the possible memory layouts that could arise during execution. Program variables, together with any dynamic variables allocated during execution, are represented by abstract memory locations. Each abstract memory location is labelled by the program-point which last wrote to that location. The second phase, the 'inference phase', examines the set of stores reaching each



```
1) x = (int *) malloc(sizeof(int));
2) *x = 10;
3) if (-) {
4)      y = x;
   } else {
5)      y = (int *) malloc(sizeof(int));
6)      *y = 20;
   }
7) *x = *y;
8) end
```

Figure 4: Illustration of Horwitz et al's method for the calculation of flow dependence.

program-point and determines the locations read. A flow dependence $p \rightarrow_f q$ exists if q reads a location labelled p in any store graph reaching q. Three approximations are used to ensure that the set of store graphs at each program-point is effectively computable whenever the program contains a loop.

Figure 4 shows an example of Horwitz et al's method. The meaning of the program in the abstract semantics is the set of all (program-point, labelled-stores) shown. Flow dependencies are found by examining the locations used at each program-point. Point p is flow dependent on each point that labels a location used at p. For example, program-point 6 uses location *L1*, which is labelled by point 5. There is consequently a flow dependence $5 \rightarrow_f 6$. Figure 2 shows the resulting PDG.

This example shows how Horwitz et al avoid the need for alias analysis. The dependence $2 \rightarrow_f 7$ arises in terms of aliases because:

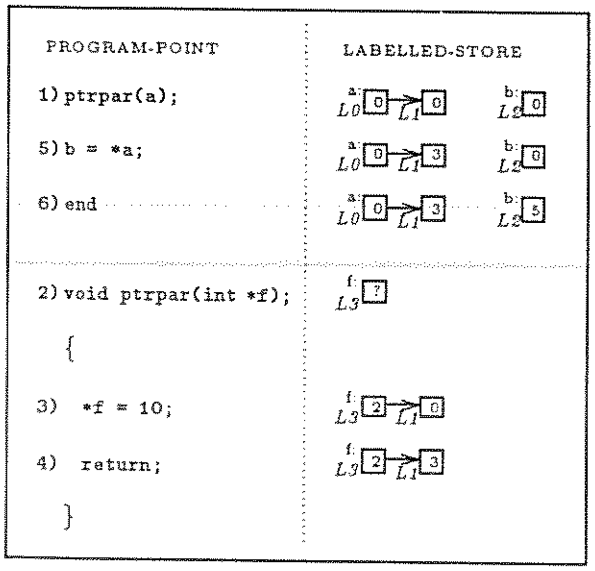
- program-point 2 defines `*x`.
- program-point 4 creates the alias `<*x,*y>`.
- program-point 7 uses `*y`.

By dealing with locations rather than object names, the dependence arises because:

- program-point 2 writes location *L2*.
- program-point 7 reads location *L2*, labelled 2.

The alias `<*x,*y>` is not computed explicitly but is implied by the store graph.

This technique can be used to deal with pointer parameters. In C all parameters are passed by value[2]. The formal parameter is a copy of the actual parameter. This copy can be changed within the callee without affecting the actual parameter in the caller. However, by passing a pointer parameter, the formal pa-



```
PROGRAM-POINT
1) ptrpar(a);
5) b = *a;
6) end

2) void ptrpar(int *f);
   {
3)   *f = 10;
4)   return;
   }
```

Figure 5: Calculation of flow dependence for pointer parameters.

rameter becomes a copy of the pointer. By dereferencing the formal parameter the objects referenced by the actual parameter within the caller can be accessed within the callee. Since these referenced objects are not copies, any changes made will remain when control returns to the caller function.

Figure 5 shows an example of a function with a pointer parameter. It is assumed that each variable visible at program-point 1 has been previously defined at point 0 (not shown). Point 1 is a call to `ptrpar` which has one parameter of type `int *`. The formal parameter `f` becomes a copy of the actual parameter `a` and hence points to location *L1*. Since `f` is a pointer, it may be dereferenced within `ptrpar`, allowing access to location *L1*. This location is defined at point 3. When control returns to the caller *L1* has been labelled by point 3. The use of location *L1* at point 5 gives an interprocedural flow dependence $3 \rightarrow_f 5$.

This example highlights a major difference between the UIG and the new CCG. The flow dependencies contained in the UIG are intraprocedural. Interprocedural dependencies are encapsulated within the call interface. In the UIG, interprocedural definition-use associations arise as a result of the call by reference parameter passing scheme. A location may be defined within one procedure and later read within another procedure only where that location is visible as a result of its use as a reference parameter. This interprocedural definition-use information is not explicitly contained in the UIG. Instead, the IFG subcomponent of the UIG allows reachable use information to be gathered intraprocedurally and then to be propagated throughout the graph, making use of the parameter interface. Interprocedural definition-use relationships can then be determined.

In C the combination of pass by value and pointer parameters means that objects can be accessed within a callee function without explicitly appearing in the parameter interface. Interprocedural dependencies are no longer encapsulated within the call interface. For example, in figure 5 location *L1* is visible within function `ptrpar` yet does not occur explicitly as an actual parameter. The resulting flow dependence $3 \rightarrow_f 5$ will pass directly from the CCG subgraph for function `ptrpar` into that for the caller function. The parameter interface has been bypassed completely. The resulting interprocedural flow dependence is shown in figure 6. Since employing this technique to determine flow dependencies explicitly uncovers interprocedural definition-use relationships, the IFG-based propagation algorithm used in construction of the UIG is no longer necessary.

4 Embedded side effects and control flow

Side effects occur when a variable is altered during the evaluation of an expression. In C, side effects can arise as a result of assignment statements, increment and decrement operators and function calls. Wherever a statement may contain an expression, side effects are possible. For example, the variable `y` is defined as a side effect of the test

```
if (x == (y = 5)) ...
```

The increment of variable `i` in

```
while (a[i++] == 0) ...
```

similarly is a side effect. A function call involves side effects if any variables are defined during the execution of the function. For example,

```
if (x == f()) ...
```

may involve the definition of variables within function `f`.

The 'reaching stores phase' of Horwitz et al'[11] dependence analysis algorithm evaluates approximate store graphs at each program-point. In each example given by Horwitzi et al a program-point corresponds directly to a single statement. However, where a statement contains an expression with embedded side effects, the variable definitions involved will lead to the

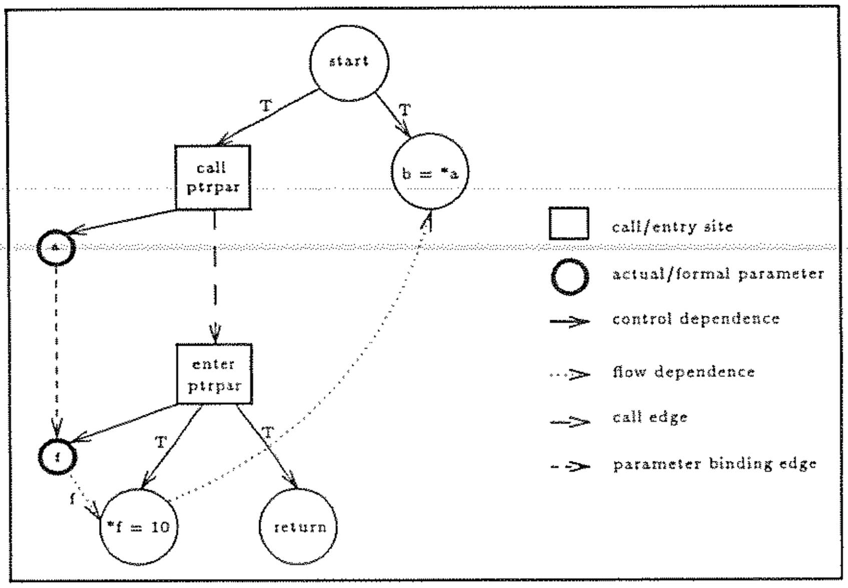


Figure 6: Dependence graph with pointer parameter.

creation of new store graphs at intermediate stages during the execution of the statement. This problem occurs in the statement

```
x = (y = 5);
```

Both variables x and y are defined but there is an intermediate stage at which only the definition of y has taken place. An additional program-point is required to represent this intermediate stage. The CCG must consequently be a more finely grained program representation than the UIG.

An expression in C may also contain embedded control flow. This occurs with the conditional expression operator `?:`. The conditional expression,

```
(a > b) ? a : b;
```

evaluates either a or b depending on the value of (a>b). The use of short-circuiting in evaluating boolean expressions similarly leads to embedded control flow. For example, in the expression

```
if (x && y && z) ...
```

if x is false, the value of the entire expression is false and y and z will not be evaluated. There is consequently a possible change in control flow associated with the `&&` and `||` operators.

A fine grained analysis is again required to deal with these embedded control flows. Additional program-points must be introduced to represent each possible flow of control. For example, the above conditional expression requires three program-points. The first point represents (a>b) and the second and third a and b respectively, each control dependent on the first. A statement in C no longer corresponds to a single program-point but must be subdivided whenever side effects or control flows are embedded.

Three new edges are contained in the CCG. The first of these edges is the *expression-use edge.* The statement

```
x = (y = 5);
```

will produce two program-points, one to represent the embedded side effect (y = 5) and the other to represent the final assignment (x =). This assignment uses the value produced by the expression (y = 5). An expression-use edge (y = 5)$\longrightarrow_{eu}$(x =) indicates this relationship. Figure 8 contains an expression-use edge representing the use of the expression `*t++` at an assignment node.

A similar *return expression-use edge* indicates the relationship between the expression evaluated at a return statement and its use within the calling expression. The graph for the call

```
x = square(a);
```

shown in figure 7 contains a return expression-use edge **(return f*f)**$\longrightarrow_{reu}$(x =).

An *lvalue* is an expression referring to a named region of storage, its name being derived from the assignment statement, the left hand side of which must be

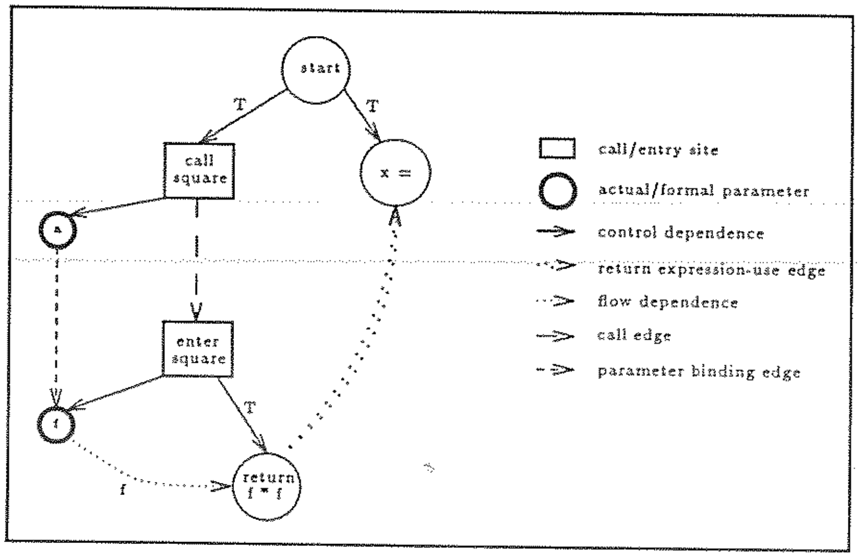


Figure 7: Example of return expression-use edge.

an lvalue. An *lvalue definition edge* is added whenever a program-point evaluates an lvalue expression which is then defined at a second program-point. This situation may arise when side effects occur on the left hand side of an assignment statement. With the exception of the boolean operators && and ||, the conditional operator ?: and the comma operator, the order of evaluation for operands within C is undefined. A compiler may therefore choose to evaluate the left hand side of an assignment statement followed by the right hand side and finally perform the assignment itself. For example, the statement

```
*p++ = (x = 10);
```

may

- evaluate the lvalue `*p`.
- increment the pointer `p`.
- evaluate the expression `x = 10`.
- assign the value of the expression `(x = 10)` to the lvalue `*p`.

An lvalue definition edge $(*p{+}{+}) \rightarrow_{lvd} (=)$ will result. Figure 8 shows the CCG for the statement

```
*s++ = *t++
```

It is assumed that the order of evaluation is left to right. The lvalue `*s` is evaluated first, with the pointer `s` incremented as a side effect. The expression `*t` is then evaluated and `t` incremented as a side

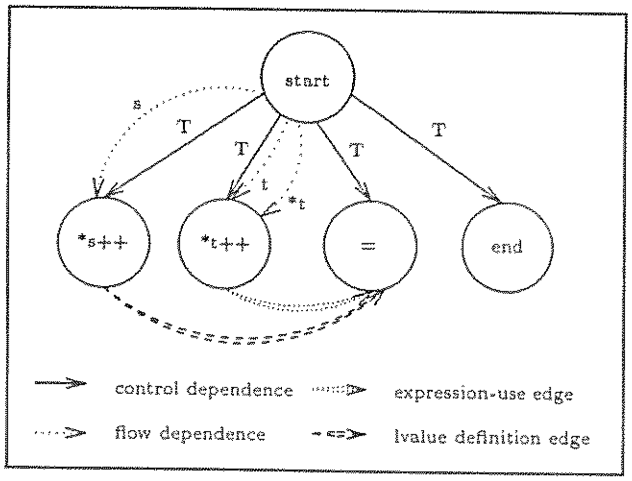


Figure 8: Resulting graph representation for `*s++ = *t++`.

effect. Finally, the assignment is performed. The expression-use edge $(*t{+}{+}) \rightarrow_{eu} (=)$ shows that the value of the expression `*t++` is used in the assignment. The lvalue definition edge $(*s{+}{+}) \rightarrow_{lvd} (=)$ indicates that the lvalue `*s++` is defined by the assignment.

5 Conclusions

A maintainer presented with a variety of views of a subject program is able to gain a better understanding of that program. Earlier work has suggested the need for a combined program representation containing sufficient information to construct each of the required views. This paper has presented extensions to an existing combined graph, the UIG, to create the CCG allowing the representation of C programs.

An existing algorithm for dependence analysis in the presence of pointers is described and examples given of its use with C programs. The possibility of pointer parameters in C is discussed and the effects on the combined graph observed. The graph's flow dependencies are no longer intraprocedural and the interface between functions is no longer encapsulated. Interprocedural flow dependencies are represented explicitly on the CCG.

The possibility of side effects and control flow embedded within C expressions requires a finer grained analysis of each source statement. New graph nodes are created where these embedded side effects and control flows are present. Three new edge types, the expression-use edge, the return expression-use edge and the lvalue definition edge are introduced to relate these new finer grained nodes.

A completed graph for the C language will require the representation of additional language constructs. Array variables are currently treated simply as aggregates and any subscript information is lost. Data dependence analysis for array variables in the context of compiler optimisations has been described by Pugh[12]. In certain cases this detailed analysis of array subscripts may be applicable.

Further work is also required to produce algorithms to extract information from the combined graph. Call graphs, control dependence and interprocedural and intraprocedural reaching definition information are contained explicitly within the CCG. Whilst calculating the graph's flow dependencies, it may also be possible to determine interprocedural 'may be preserved' information. Where an abstract location reaches across a call site with its program-point label unchanged, the corresponding variable is preserved across that call site.

The interprocedural program slicing algorithm[7] described by Horwitz et al is based on a two phase graph traversal. A similar traversal algorithm applicable to the CCG making use of the new expression-use edge, lvalue definition edge and return expression-use edge is required. Further work is necessary to investigate the problems of calling context and the elimination of redundant nodes and edges from the slice.

References

[1] M. Harrold and B. Malloy, "A unified interprocedural program representation for a maintenance environment," in *Proceedings of the Conference on Software Maintenance - 1991, Sorrento, Italy*, pp. 138–147, October 1991.

[2] B. Kernighan and D. Ritchie, *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, second ed., 1988.

[3] D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia*, pp. 47–56, June 1988.

[4] M. Harrold and M. Soffa, "Computation of interprocedural definition and use dependencies," in *Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages, New Orleans, Louisiana*, pp. 297–306, March 1990.

[5] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[6] K. Ottenstein and L. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Notices*, vol. 9, pp. 177–184, May 1984.

[7] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, January 1990.

[8] W. Landi and B. Ryder, "Pointer induced aliasing: A problem classification," in *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida*, pp. 93–103, January 1991.

[9] W. Landi and B. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," *SIGPLAN Notices*, vol. 27, pp. 235–248, July 1992.

[10] H. Pande, B. Ryder, and W. Landi, "Interprocedural def-use association in C programs," in *Proceedings of the ACM SIGSOFT 1991 4th Symposium on Software Testing, Analysis and Verification*, pp. 139–153, October 1991.

[11] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," *SIGPLAN Notices*, vol. 24, no. 7, pp. 28–40, 1989.

[12] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the ACM*, vol. 35, pp. 102–114, August 1992.

An Integrated and Interactive Reverse Engineering Environment for Existing Software Comprehension

U. De Carlini*, A. De Lucia*, G. A. Di Lucca*, G. Tortora°

* Dipartimento di Informatica e Sistemistica
University of Naples "Federico II"
via Claudio, 21 - 80125 Naples

° Dipartimento di Informatica ed Applicazioni
University of Salerno
84081 Baronissi, Salerno

Abstract

Software Comprehension is an incremental process to support the understanding of both the behaviour and the structure of software system. It uses the existing documentation of the system and completes it with information deduced from the code by means of Reverse Engineering tools.
As pointed out by several authors, the current Reverse Engineering tools produce pre-defined sets of reports and, in some cases, furnish answers to fixed kinds of queries, thus being inadequate to completely support Software Comprehension.
To this aim, in this paper, an interactive Reverse Engineering environment is described which supports information extraction and abstraction processes about a software system.

1. Introduction.

Most of the problems regarding existing software are related to maintenance operations and software reuse. These problems are due to the partial or total lack of software documentation or its inconsistency with respect to operating code. In fact, the source code is often the only available documentation on a software product. This creates obstacles for software comprehension since we know very little about its design and structure at various abstraction levels. Consequential, we cannot easily determine the components of a software systems and the relationships existing among them and it thus becomes harder to perform maintenance operations and identify reusable software components.

This situation has led to greater importance being attributed to Reverse Engineering (in the following RE) which is a set of Software Engineering theories, methodologies and techniques supporting the extraction and abstraction of information from an existing software product in order to produce software documents, at various abstraction levels, that are reliable, consistent and coherent with the operating code. The main goals [1] of an RE process can be synthesized as follows:

- identification of software components and relations existing among them;
- creation of software representations at a higher abstraction level than code (and also at various abstraction levels);
- understanding and description of software functions and of how they were implemented (i.e. "what" the software does and "how" it does it).

The extracted and abstracted information has to be arranged in such a way that it is easily understood, thus enabling a faster and more efficient comprehension of the existing software system. Generally, the information is arranged according to a chosen program representation form in order to show the program intermodular and intramodular structure.

The documents produced by the RE processes usually are the same suggested by the most widely used methodologies for software development. It is worth pointing out that different documents reconstructed by RE usually require different kind of information to be extracted/abstracted and, correspondingly, different program Intermediate representation Forms (in the following IF) as widely discussed in [2].

In this paper an interactive RE environment is described based on Webs. Webs provide a graphical, language independent intermediate representation form of programs. Information extraction and abstraction processes

This work has been supported by "Progetto Finalizzato Sistemi Informatici and Calcolo Parallelo" of CNR under grant 91.00930PF69

can be written in TDL. TDL (Tool Development Language) is a language in which Web manipulations can be naturally expressed.

The paper is organized as follows. Section 2 analyzes the main problems dealing with current Reverse Engineering processes related to software comprehension. Section 3 briefly describes the main definitions and characteristics of the Web structures that are proposed as IFs capable of effectively representing all the information embedded in the code. Section 4 presents the Tool Development Language TDL, which makes it possible to manipulate Web structures interactively using a graphic formalism. Section 5 introduces the concept of "direct relations" and "summary relations" and how they can be obtained from the Web representation. Some TDL procedures extracting direct and summary relations are reported in the Appendix.

2. Intermediate representation forms and software comprehension.

The RE processes generally used are often inadequate and unable to satisfy the user's needs, especially as far as maintenance and reuse activities of the analyzed software are concerned. There are many reasons for this but it is mainly due to:

- the lack of information in the produced documents, as compared to what is actually needed;
- the incompleteness of the information extracted and of the intermediate representation forms adopted;
- the lack of flexibility in the tools used.

This is motivated by the the following considerations.

The reconstructed documents, which are highly useful in the development phases, are not always so useful in the maintenance phase, or for identifying reusable components. The documents on the software system intermodular structure (High Level Design documents) are too general (and thus useful only for an initial understanding of the software system), while those dealing with the intramodular structure (Low Level Design documents) are too detailed (too much detail makes it harder to search for and take out the wanted information). Moreover, these documents (at any abstraction level) only point out particular views of the whole system, and therefore give only partial information about it. It will thus be up to the user to select information from each of them and collect this information into a new and unique document. Hence the user has continuously to fetch and validate information from the various reconstructed documents in order to obtain an "ad hoc" document summarizing the information he needs. Generally, the final document is not a "standard" document, like the reconstructed ones, but a mixed document in which each piece of information interweaves and complements the other information.

The Intermediate Forms (IF) generally used are unable to represent all the kinds of information that can be extracted and abstracted from source the code nor is it possible for some of them to be used to abstract any kind of information at a higher level. None of them can be used as the unique IF representing all the information and knowledge embedded in the code.

In [3] the most common IFs used in RE processes are described. Some of these can cover several abstraction levels and/or represent many kinds of information, both intramodular and intermodular, but they are always limited to a partial view of the software product.

For example, control flow graph [4] [5], nesting tree [6] [5], algebraic expression [7] give useful information about the intramodular control flow of a program and the nesting of its control structures. Although these IFs can be enriched with data in order to obtain intramodular data flow, they give us no information about data dependences or data visibility. Call graph [8], call tree [8], structure charts [9] give information about modular structure, calling/called relationships, module relationships ("uses" and "composed of" relationships) and intermodular data flow but no information about pathological connections [10] or data aliases or module visibility.

Dependence graphs [11] [12] should seem to have solved this problem. They represent information both at intraprocedural and interprocedural level: control flow, data dependencies, intermodular data flow and activation relationships. We can abstract control structure nesting, intramodular data flow, call tree and graph, etc., but dependence graphs give us no information about data and module visibility.

In particular, let us note that almost all the IFs give no information about data and module visibility. Hence they can only represent actual relationships between software components but say nothing on the potential ones. This means that we can know everything about Effective Module Relationships and Effective Data Relationships [13], but in practice nothing about Potential Module Relationships and Potential Data Relationships [13]. In other words, the information about potential relationships among software components is totally ignored by most IFs, even though they are present in the code.

Potential Relationships is the term we use to indicate the set of links between software components that are not actually implemented in the code, but which could be successively set up according to the programming language visibility rules and are therefore potential links. An example is the potential link between a subprogram and a second one when the latter is visible from the former and then potentially recallable from it.

The tools generally used to set up RE processes for the extraction and abstraction of information are typically inadequate due to their limited flexibility and can thus only produce documents according to how they were designed. They allow no interactivity to the user, who could drive the process according to the information already acquired (by extraction and/or abstraction) and by asking for the new information he still needs (turning on new extraction and/or abstraction processes and tuning the whole RE process). In general an RE process starts from static analysis of the source code, and possibly of any existing documentation. In this phase information about the single components making up the software system and information about the relationships existing between them is extracted from source code by an automatic process. This information, which in the following we will call "direct relation" [14], provide a first, language-independent representation of the software system at a low level of abstraction. From direct relations we can abstract information at a higher level of abstraction describing the other, non-direct relationships between the various software components. We will call this information "summary relation" [14]. This process is generally performed in "batch" mode without user interaction and the tools extract and abstract only the information useful for reconstructing the documents for which the tools were designed. In other words they can only reproduce the previously established documents and provide only the answers that can be obtained from them.

RE processes, on the other hand, have to work incrementally, as with the comprehension process, so that the information extracted and abstracted can be suitably integrated so as to create documents that are gradually enhanced with the information needed and/or prune out the superfluous one.

For this to be performed efficiently, the RE process must be based on an IF capable of representing all the software system information embedded in the code, and on the use of tools that allow interactive manipulation of the IF so as to obtain the information that is actually needed and in the desired form. The use of such tools also allows a more effective and easier management of the IF which, due to the big amount of information it gathers, will be more complex than the ones generally used, which include only part of the information embedded in the analyzed code.

3. Webs for program comprehension.

Let us briefly recall the definitions of relational structure [15] and Web structure [16].

A relational structure (briefly a structure) is a pair $G=(G_A,G_F)$, where G_A is a set of atoms and G_F is a set of formulas, resulting from formally applying an n-ary predicate from a set P of predicates to a list of n arguments from G_A.

A Web structure is a particular case of relational structure and consists of a supporting tree structure, whose atoms, nodes of the tree, are colored by means of predicates from a set ***Op*** of unary operator predicates (for every atom n in the structure there exists a unique formula Op(n) with Op $\in$ ***Op***). A father-son relation between atoms n and m is given by a formula S(n,m). On this tree structure, a second one is superimposed which consists of special arcs, called Web arcs, connecting nodes of the underlying tree. Web arcs are labelled by means of binary predicates from a set ***Cur*** of cursor predicates.

By properly choosing the set ***Op*** and ***Cur***, a Web can describe a program in any programming language. The tree structure represents the syntactical structure of the program, while the formulas constructed by means of the predicates in ***Cur*** provide information about both the control flow and properties that the tree alone cannot express. Predicates in ***Op*** represent items of a programming language.

Webs provide a graphical, language-independent representation of programs. The set ***Op*** of operators does not depend on a specific programming language, but it can be assumed to be large enough to represent constructs in any imperative programming language.

Fig. 1 shows the graphical representation of Web items.

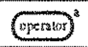
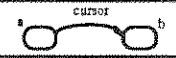
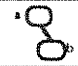
Web Item	Graphical Representation
Atom a	
operator(a)	
cursor(a, b)	
S(a, b)	

Fig. 1: Graphical Representation of Web Items

In the following we will consider the following set of operator and cursor predicates, referring to a Pascal-like language:

Op = {**program**, **procedure**, **function**, **declist**, **dectype**, **decvar**, **decproc**, **decformpar**, **formpar**, **valpar**, **varpar**, **type**, **vbl**, **int**, **real**, **bool**, **char**, **record**, **field**, **file**, **id**, **halt**, **endproc**, **endfun**, **restype**, **group**, **continue**, **while**, **repeat**, **it**, **ite**, **asgn**, **callproc**, **callfun**, **actualpar**, **open**, **create**, **read**, **write**, **eof**, **recfield**, **plus**, **minus**, **times**, **div**, **exp**, **minus1**, **and**, **or**, **not**, **ge**, **le**, **eq**, **ne**, **gt**, **lt**} $\cup$ {**numconst** | **numconst**$\in$ R} $\cup$ {**chconst** | **chconst**$\in$ **Characters**} $\cup$ {**true**, **false**}

Cur = {**start**, **next**, **yes**, **no**, **left**, **right**, **del**, **call**, **reccall**, **actualize**}.

As an example of a Web representation of a program, let us consider the program shown in Fig. 2. Figs 3 and 4 show respectively the graphical Web representations of the declarative structure and the representation referring to the piece of code enclosed in the *if-then-else* control structure of the MAIN program.

In RE processes we also need to know the names of the software components of a program. Therefore a relational representation of the symbol-table is added to the Web structure of the program in the following way:

a) a unary predicate **name** is added to the set ***Op*** and a binary predicate **ident** is added to the set ***Cur*** ;

b) each name of a software component in the program becomes a new atom of the structure and is colored by the predicate **name** .

c) each atom of the Web structure declaring a software component (in our example each atom colored by a predicate from the set {**program**, **procedure**,

function, formpar, type, vbl, field}) is linked to the corresponding atom name.
Fig. 5 shows some **ident** links for our sample program.

Web manipulation is accomplished by means of production and structure derivation. The formal definitions of productions and structure derivations rely on the algebraic framework of the theory of categories. We will give an informal description of the concepts involved, leaving out the formal details which can be found in [16]. Here, a production p has the form $B_1 \Rightarrow B_2$, where B_1 and B_2 are structures, and expresses the replacement of a substructure B_1 (left hand side of the production) with the substructure B_2 (right hand side).

As an example, the production of Fig. 6 replaces a sum operation with a multiplication one.

Given structures $G=(G_A,G_F)$ and $H=(H_A,H_F)$, a structure morphism (briefly a morphism) $g:G \rightarrow H$ is any pair $g=(g_A,g_F)$ of maps $g_A:G_A \rightarrow H_A$ and $g_F:G_F \rightarrow H_F$ such that if $\phi(x_1,\dots,x_n)$ is in G_F then $g_F(\phi(x_1,\dots,x_n))=\phi(g_A x_1,\dots,g_A x_n)$. The composition of morphisms is defined as usual.
Given a production p, say $B_1 \Rightarrow B_2$, structures G and H and a morphism $g:B_1 \rightarrow G$, a direct derivation $G \Rightarrow H$ via p based on g can be intuitively described as follows: the morphism g embeds the left-hand side of p in G by searching the substructure of G which matches B_1. The structure H is obtained from G by first deleting atoms and formulas of B_1 (not occurring in B_2) determined by g and then by glueing a proper instance of B_2.

The Web representation of programs is a good candidate for IF; in fact, with respect to the requirements reported in Tab. 1 [3], the Web representation:
- provides a transformation support that preserves information: only the code representation is changed, not the original semantics of the information included in it;
- is independent of the language used: by properly choosing the sets ***Op*** and ***Cur***, a Web can describe a program in any programming language;
- provides a complete set of information about what it is representing at that abstraction level;

```
program MAIN;
  type bill_rec = record
                        us_id:integer;
                        bill_num:integer;
                        bill_tot:integer;
                        paid:boolean;
                      end;
    bill_file = file of bill_rec;
var
  bills: bill_file;
  rec:bill_rec;
  paid_tot:integer;
  paid_bill:integer;
  delay_tot:integer;
  delay_bill:integer;
  nr,pag:integer;
  procedure Print_row (br:bill_rec; var pnr:integer; ppag:integer);
    procedure Print_head(pagnr:integer);
      begin
        ......
      end;
    begin
      ......
      print_head(ppag);
      ......
    end;
  procedure Print_Tot (T1, T2, T3, T4:integer);
    begin
      ......
    end;
  begin
    ......
    paid_bill:=0;delay_bill:=0;paid_tot:=0;delay_tot:=0;nr:=0;
    read (bills,rec);
    while not eof (bills) do
      begin
        if rec.paid then
          begin
            paid_tot:=paid_tot+rec.bill_tot;
            paid_bill:=paid_bill+1;
          end
        else
          begin
            delay_tot:=delay_tot+rec.bill_tot;
            delay_bill:=delay_bill+1;
          end;
        if nr=0 then pag:=pag+1;
        nr:=nr+1;
        print_row(rec,nr,pag);
        read (bills,rec)
      end;
    print_tot(paid_tot,paid_bill,delay_tot,delay_bill);
end.
```

Fig. 2: A simple Pascal Program

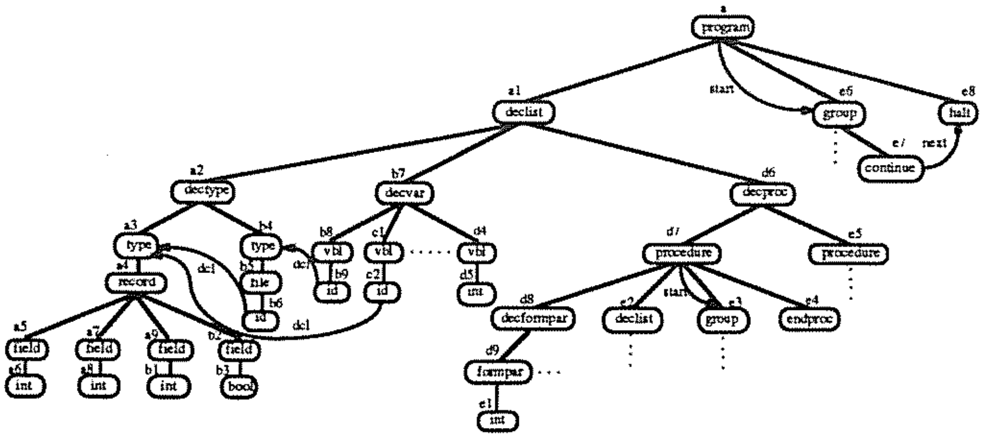


Fig. 3: Graphical Web representation of the declarative structure of the Program in Fig. 2

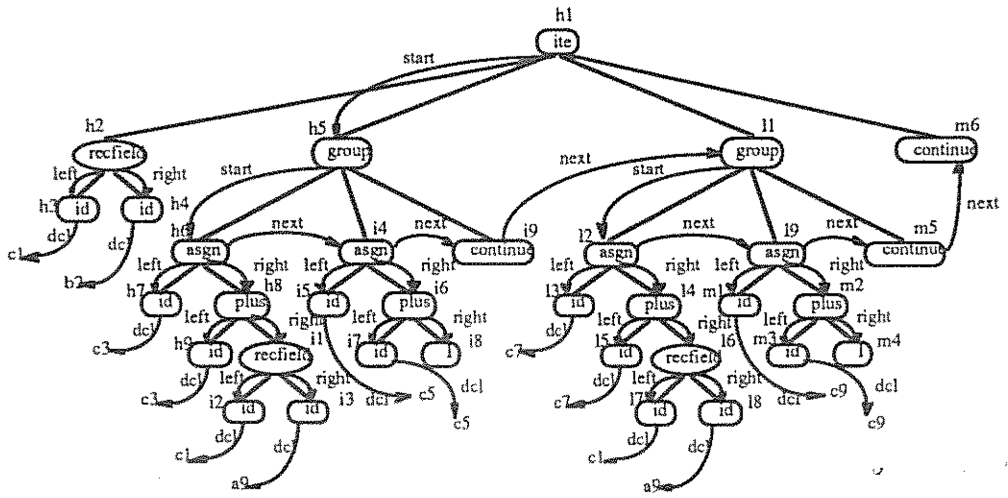


Fig. 4: Graphical Web representation of a piece of code of the Program in Fig. 2

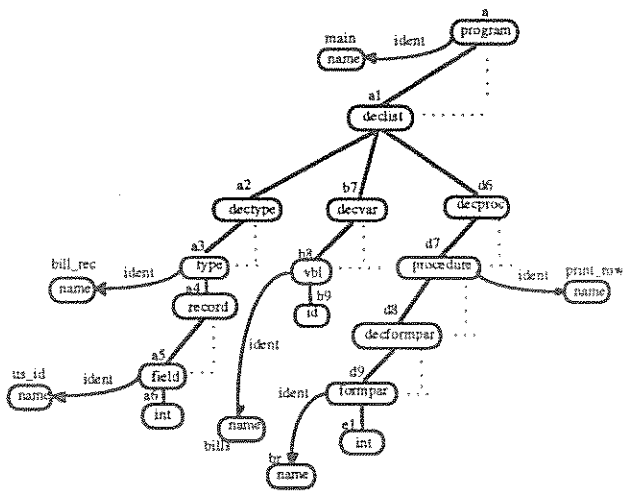


Fig. 5: Some ident links of the program in Fig. 2

An Intermediate form has:
- to provide a transformation support in an information-preserving way, without changing the original semantic of information but only their representation;
- to be independent from the language used for describing the original documents it represents (e. g. the source code) in order to the same IF can be used for different languages;
- to provide a complete set of information about what it is representing at that abstraction level;
- to be in a compact form and to represent object and information by a determinate granularity degree;
- to be able to present different objects by the same formalism and to have uniformity of representation;
- to allow to retrieve details in a easy and fast way and then it has to allow to control the degree of detail and/or abstractions represented in it;
- to have a good degree of specialization;
- to have a high expressive power.

Tab. 1: Intermediate Form Requirements

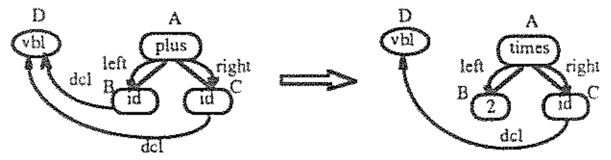


Fig. 6: An example of production

- has a fairly compact form and represents objects and information with a good degree of granularity;
- is able to present different objects with the same formalism and has uniformity of representation;
- allows details to be retrieved in an easy and fast way and makes it possible to control the degree of detail and/or abstractions represented in it;
- has a good degree of specialization;
- has a high expressive power.

Objections could be raised about the compactness of the Web and the fast and straightforward retrieval of details in it. The richness of information and relationships makes the Web representation more complex than others but it is this very richness that makes the Web representation so powerful. However we can easily select from all the relations in the Web structure, only the ones that describe the information we need.

4. The tool development language TDL.

In an RE process, successive abstractions are obtained by applying different tools so that information produced by one tool is exploited by another one. Starting from the

Web representation of a program, an RE tool enriches the structure with new information. This information, expressed in a relational way, can be useful in obtaining intermediate representations of the program at higher abstraction levels.

In this section we show how software tools can be created by means of a Tool Development Language (TDL).

In TDL [17] the fundamental data types are atoms and formulas and the elementary statement is the Web rewriting rule (Web transformation). TDL can be used to design open environments [18] supporting an expandable set of integrated tools for RE, restructuring, reuse and reengineering processes. To make the use of TDL simple and close to the user's intuitive notion, Web transformations are represented graphically.

TDL is provided with an atom type, a predicate type and a list type. The predicates in $Op \cup Cur$ are pre-defined predicate types: their names are specialized in the particular programming languages in use. The user can define new predicate types by specifying name and arity.

The type ***list*** is recursively defined. The empty list is denoted by < >. A non-empty list *L* is denoted by *L* = <*a*, *sublist*>, where *a* is an atom and identifies the first element (the *head* of *L*) and *sublist* identifies the sublist of remaining elements (the *tail* of *L*). A list used in a TDL program is declared in a box at the beginning of the program specifying the name of a variable of type *List*. The operator *member* is available to verify the membership of an atom in a list. A list can be added to a Web as a new unary predicate specified in the right hand side of a production. Then, it can be transformed like any other predicate in the Web. Fig. 7 shows an example of a production transforming a list, namely deleting its first element.

Constant predicate names and atom labels are written in lowercase, while variables are written in uppercase. User-defined data types and variables must be declared at the beginning of a program.

Productions are the elementary statements of TDL. If

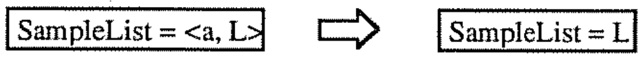


Fig. 7: An example of a production

$B_1 \Rightarrow B_2$ is a production, its execution implies a direct derivation via a morphism g determined by a Prolog-like unification mechanism in such a way that g(B_1) is an instance of B_1 in the Web G. We say that an elementary statement $B_1 \Rightarrow B_2$ *succeeds* if there exists an instance of B_1 in G and the application of the production modifies G (in other words, if B_2 contains some nodes or some relations which are not in B_1). Besides the elementary statements, the language provides compound statements, conditional statements and iterative statements.

Compound statements

TDL provides a sequential compound statement and a selective compound statement. The first one has the form:

```
execute-all
  S1;
  .
  .
  Sn
end
```

The successful statements among $S_1,..., S_n$ are intended to be executed sequentially. The compound statement succeeds if at least one statement among $S_1,..., S_n$ succeeds (and then at least one statement is executed).

The selective compound statement has the form:

```
execute-the-first
  S1;
  .
  .
  Sn
end
```

This statement is performed by selecting and then executing the first successful statement among $S_1,..., S_n$ and it succeeds if such a successful statement exists.

Conditional statement:

The conditional statement in TDL has the form

```
if C then S;
```

where C is a set of relations and S is a statement. The statement S is executed if it succeeds and the relations in C appear in the structure: in this case the conditional statement succeeds.

Iterative statements

TDL provides two iterative statements.The first one has the form:

```
while C do S
```

where C and S have the same meaning as in the conditional statement. The statement S is intended to be iteratively executed while it succeeds and the relations in C appear in the structure.

The second iterative statement has the form:

```
repeat S until-modify
```

The statement S is intended to be iteratively executed while it succeeds.

Both the iterative statements succeed if S is executed at least once.

TDL allows procedure calls with formula parameter passing. A formula passed as a parameter can only be used as a condition of if or while statements.

5. Direct and summary relations.

We call direct relations the information that can been extracted directly from the code by simple static analysis of the code itself. They include the simple relations existing between pairs of software items without taking into account "dynamic" or "transitive" relations. The dynamic or transitive relations are successively obtained by abstraction processes starting from the direct relations. This abstracted information is referred to as "summary relation".

In a traditional RE process, during the static analysis phase each module is automatically analyzed independently from the others, and for each module direct relations provide information about:

a) module names, and types and names of formal parameters ;
b) variables names and types;
c) called module names and passed actual parameters;
d) variables definition;
e) variables use;
f) constant names;
g) control structures, their nesting and control flow.

Thus, we describe a software system by its components and the direct relations among them. For the sake of simplicity and without loss of generality, in the following we will refer to a software system composed of a program with subprograms (procedure and/or functions) declared in it, that does not use external modules (such as ADA packages or PASCAL units).

Let us define the following set:

- PP is the set of program, procedures and functions names;
- TT is the set of user defined type names;
- DD is the set of variables names;
- KK is the set of constants names;
- CS is the set of control structures.

Among these components there exist direct relations characterized as follows:

a) call relation: the relation between the calling component and the called one. It is defined on (PP x PP). From this relation we can abstract the call tree, or call graph, and it is useful in abstracting the structure chart.
b) declare relation: the relation between the declaring component and the declared one. The declaring component will be in the set PP, while the declared one is in the set ii, where $ii=PP \cup TT \cup DD \cup KK$. Thus the relation is defined on (PP x ii). The relation is useful in defining the scope of a component, according to the source code language visibility rules. It allows us to know if a procedure can refer a data item or call another procedure/function and so on.
c) use relation: the relation holding between an item refers to another without changing the latter value. It will be a relation defined on:
    - $(PP \times (TT \cup DD \cup KK \cup CS))$, in this case the relation means that a data item, a user type, a constant or a control structure is used in the main program or in a procedure/function;
    - (TT x TT) when another previously declared user type was used in a user type declaration;
    - (CS x DD) to mean that a data item was used in a control structure.

   The use relation is useful for determining interprocedural and intraprocedural data dependencies.
d) definition relation: the relations holding between a variable and the place (program unit, resp. control structure) in which its value is defined or changed. The relation is defined on (PP x DD), resp. (CS x DD). This relation is useful in determining interprocedural and intraprocedural data dependences.
e) user_type data relation: the relation existing between a variable name and its corresponding user type name. It is defined on (DD x TT) and it is useful in finding out all variable names of a given user type in order to identify data abstractions (object, generic object, abstract data type, generic abstract data type).
f) value dependence relation: the relation existing between two variable names when one is used for defining the value of the other one. It is defined on (DD x DD) and is useful in determining data dependences.
g) formal parameters relation: this relation holds between a variable name and a procedure/function when the first one is a formal parameter for the second one. It is defined on (PP x DD) and is useful in determining interprocedural data_flow.
h) actual parameter relation: it says what variable names are passed as actual parameter in a procedure/function call. It is defined on (PP x PP x DD) and is used for determining interprocedural data_flow, together with the relation of the previous point g).
i) control_structure relation: it is the relation holding between control structures, of a given program unit, when one of them is nested in the other one. It is defined on (PP x CS x CS) and is useful in reconstructing the nesting of control structures and thus the control flow of each component in PP.

These direct relations are the starting point for all successive abstractions. They are useful in obtaining and abstracting information about the other non-direct relations between the various items of the analyzed software and representing them according to some IF. For example, from the relations a), b) and h) we can abstract the call tree, or the structure chart reporting also the intermodular data flow. Alternatively we can abstract the dependence graph using the relation c), d) and i).

Direct relations and then summary relations can be extracted from the Web: the Web structure and the TDL manipulation language provide a powerful tool to perform this process. A tool for extracting the direct relations that we need is actually a TDL program, which can be written and executed interactively.

As an example, let us consider the procedure *call_extr* shown in fig. 8.

The procedure *call_extr* is used to extract the call relations from Web. It works as follows:

- the first repeat ... until modify TDL statement looks for all the Web atoms representing call statements and for the corresponding called subprograms;
- in the second repeat ... until modify TDL statement, the calling subprograms are found, starting from each "call node" and going back along the father-son arcs of the Web. When the calling subprogram is found, a *callpf* arc, linking the *name* atom of the calling subprogram to the *name* atom of the called one, is created. The *callpf* arc represent the "call" direct relations described at a).

The Figure 9 shows the call graph of the sample program in Fig. 2, abstracted using the procedure *call_extr*.

In the Appendix other TDL procedures for extracting direct relations from the Web representation of a program are shown. Figure 10 shows the user-type relations (see use relation at c) obtained by applying the TDL procedure *typeuse_extr* given in the Appendix to the Web representation of the sample program of Fig. 2.

In this way the typically "static RE process becomes a "dynamic" one.

In fact the user interactively decides which are the relations to be extracted and/or abstracted depending on the acquired knowledge about the software system.

Figure 11 shows the corresponding overall diagram of the RE process: in the environment a syntax direct translator takes care of translating of the source code to be analyzed into the Web representation. After the Web has been generated the user starts to inquire and manipulate it in order to obtain the information he needs about the software system under examination.

The information the user needs is expressed by TDL procedures describing the manipulation to be performed on the Web in order to extract/abstract the desired information.

The user can generate TDL procedures and store them in a TDL Library for future use. The TDL procedures are written in a graphical way by means a graphic editor.

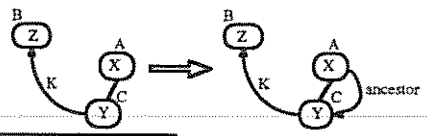
6. Conclusions

Current RE processes supporting activities for the comprehension of existing software have been shown to be inadequate to the needs of the comprehension process. This is principally due to
- the lack of information, compared to what is needed to reach the goals, contained in the documents produced by RE;
- the incompleteness of the extracted information, which restricts the possibilities for successive abstractions;
- the lack of flexibility of the tools adopted, which allow no interaction with the user.

The enhancement we proposed in this paper is an interactive RE process that enables the incremental comprehension of a software system.

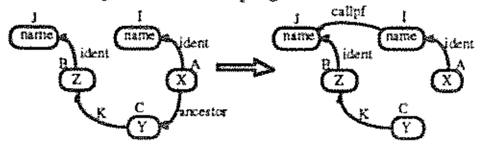
This process is based on the use of:
- the Web structure, an intermediate form (based on a relational structure) for representing programs that

```
Procedure call_extr
{The procedure extracts the call relations from the Web}
    execute-all
        repeat
{Find "call" atoms}
```



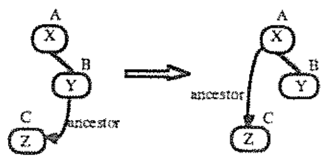
LIST_FORMULAS
Name : member
Arguments : K, <call, recall>

```
        until-modify
        repeat
            execute-the-first
{Find calling and called subprograms and insert the "callpf" arc }
```



LIST_FORMULAS
Name : member
Arguments : K, <call, recall>

LIST_FORMULAS
Name : member
Arguments : X, <program, procedure, function>



```
            end
        until-modify
    end
```

Fig. 8: The TDL procedure *call_extr*

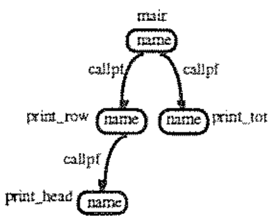


Figure 9: The call graph of the program in Fig 2

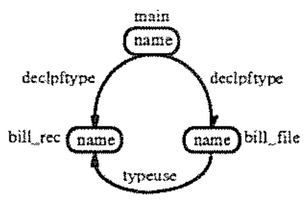


Fig. 10: The user-type relations of the program in Fig 2

makes it possible to represent all the information embedded in the analyzed source code;
the Tool Development Language TDL, a tool for manipulating the Web structure that makes it possible to extract and abstract interactively the information needed for the comprehension of the software system under examination. The required information is described according to a graphic formalism and the results obtained are also graphically.

The interactivity of such environment supports the incremental comprehension of software systems thus leaving to the user the ability of deciding which kind of analysis is needed and possibly how it must be performed, depending on the specific context at hand.

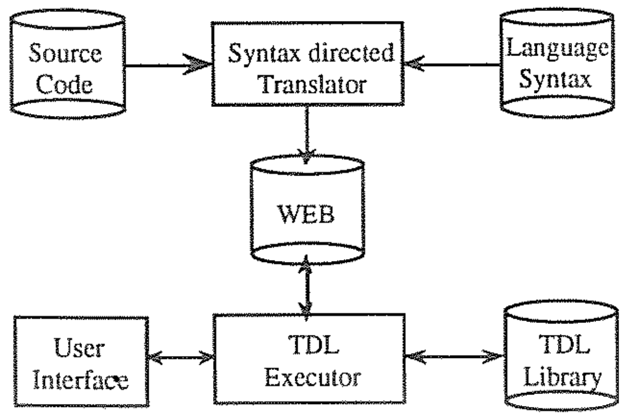
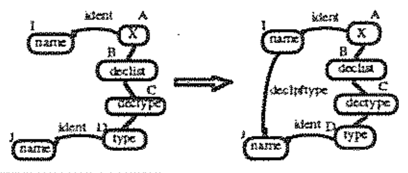


Fig. 11: The new interactive RE process

Appendix.

In the following some TDL procedures for detecting direct relations are shown.

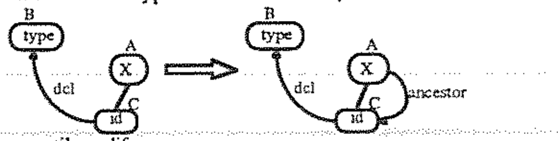
```
Procedure dcltype_extr
    {The procedure extracts the relations holding between the declaring
    component and the declared user defined type name}

    repeat
    {Find the declaring component and the declared user type and insert
    the "declpftype" arc}
```



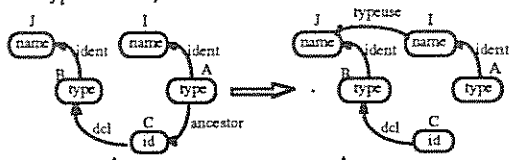
LIST_FORMULAS
Name : member
Arguments : X, <program, procedure, function>

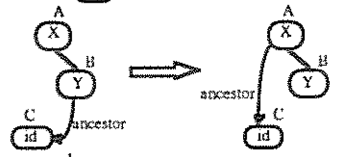
```
    until-modify
end
```

```
Procedure typeuse_extr
{The procedure extracts the use relations holding between pairs of user
defined type names when a previously declared user type was used in a
user type declaration}
    execute-all
        repeat
{Look for all user type declaration nodes}
```



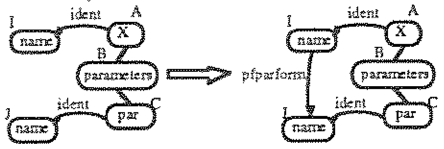
```
        until-modify
        repeat
            execute-the-first
{Find all the user type declarations using another user type and insert
the "typeuse" arc}
```





```
            end
        until-modify
    end
end
```

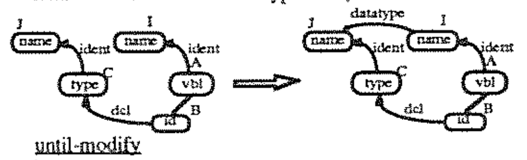
```
Procedure parform_extr
{The procedure extracts the formal parameters relations holding
between a variable name and a subprogram when the former is a
formal parameter of the latter}

    repeat
{Find subprogram names and parameter names and insert the
"pfparform" arc}
```



LIST_FORMULAS
Name : member
Arguments : X, <procedure, function>

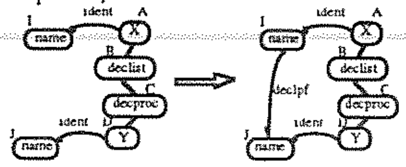
```
    until-modify
end
```

```
Procedure datatype_extr
{The procedure extracts the relations holding between a variable name
and its corresponding user type name}

    repeat
    {Find declared variable names and the corresponding user type
    names and insert the "datatype" arc}
```



```
    until-modify
end
```

```
Procedure dclpf_extr
    {The procedure extracts the declare relations holding between
    subprograms when the second one is declared by the first one}

    repeat
{Find for each subprogram the names of the declared ones and insert
the "declpf" arc}
```



```
LIST_FORMULAS
Name : member
Arguments : X, <program,
                procedure,
                function>

LIST_FORMULAS
Name : member
Arguments : Y, <procedure,
                function>
    until-modify
end
```

References.

[1] E. J. Chikofsky, J.H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, vol. 7, n. 1, Jan. 1990.

[2] P. Benedusi, A. Cimitile, U. De Carlini, "RE Methodology to Reconstruct Hierarchical Data Flow Diagrams", Proc. of IEEE Conf. on Soft. Maint. 1989, Miami, Oct. 1989.

[3] "Research report on Reverse Engineering approaches, Tools and Intermediate Forms", Esprit Project 5111 DOCKET, Workpackage A, Apr. 1991;

[4] B. Beizer, "Software Testing Techniques", Von Nostrand Reinhold, 1983.

[5] G. Cantone, A. Cimitile, L. Sansone, "Complexity in Problem Schemes: the Characteristic Polynomial", ACM Sigplan Notices, vol. 8, n. 3, 1983.

[6] G. Cantone, A. Cimitile, P. Maresca " A new Methodological Proposal for Program Maintenance", The Euromicro Journal, vol. 18, Nos 1-5, 1986.

[7] A. Cimitile, U. De Carlini, "Reverse Engineering: Algorithms for Program Graphs Production", Software Practice and Experience, vol. 21, n. 5, 1991.

[8] J.L. Baer "Graph Models in Programming Systems", Current Trends in Programming Methodologies, vol. 3, K. Mani Chandy and R.T. Yeh eds., Prentice Hall, Englewood Cliffs, N.J., 1978.

[9] E. Yourdon, L.L. Constantine "Structured Design", Prentice Hall, Englewood Cliffs, N.J., 1979.

[10] P. Benedusi, A. Cimitile, U. De Carlini, "Reverse Engineering Processes, Design Recovery and Structure Charts", Journal of Systems Software, vol.19, n.4, 1992.

[11] K.J. Ottenstein, L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment", ACM Sigplan Notices, vol. 19, n. 5, May 1984.

[12] S. Horowitz, T. Reps, D. Binkley "Interprocedural Slicing Using Dependence Graphs", ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, Jan 1990.

[13] A. Cimitile, G. A. Di Lucca, P. Maresca "Maintenance and Intermodular Depedecies in Pascal Environment", Proc. of IEEE Conf. on Soft. Maint. 1990, San Diego - California, Nov. 26-29 1990;

[14] G. Canfora, A. Cimitile, U. De Carlini "A Logic Based Approach to a Reverse Engineering Tools Production" IEEE Tans on Software Engineering, vol. SE-18, n.12, 1992.

[15] H. Ehrig, H.J. Kreowski, A. Maggiolo Schettini, B.K. Rosen, J. Winkowski, "Transformations of Structures: an Algebraic Approach", Math. Syst. Theory, vol 14, 1981.

[16] A. Maggiolo Schettini, M. Napoli, G. Tortora, "Web Structures: A Tool for Representing and Manipulating Programs", IEEE Trans. Soft. Eng., vol. 14, no.11, Nov. 1988.

[17] A. De Lucia, M. Napoli, G. Tortora, M. Tucci, "The Tool Development Language TDL for the Software Development Environment WSDW" to be published in the Proceedings of the 5th Intern. Conf. on Software Engineering and Knowledge Engineering, S.Fransisco (U.S.A.), 17-19 June 1993.

[18] A. De Lucia, A. Imperatore, M. Napoli, G. Tortora, M. Tucci, "The Software Development Workbench WSDW", in Proc. 4th Intern. Conf. on Software Engineering and Knowledge Engineering, Capri, Italy., June 15-20, 1992.

Session H: Integrating Documents and Code for Program Comprehension

Chair: Horst Zuse

DOCKET: Program Comprehension-In-The-Large

P.J.Layzell, R.Champion, M.J.Freeman

Department of Computation, UMIST, PO Box 88, Manchester, M60 1QD, UK

Abstract

With the growing awareness of the importance of software maintenance, has come a re-evaluation of software maintenance tools. Such tools range from source code analysers to semi-intelligent tools which seek to reconstruct systems designs and specification documents from source code. However, it is clear that relying solely upon source code as the basis for reverse engineering has many problems. These problems include poor abstraction, leading to over detailed specification models and the inability to link other parts of a software system, such as documentation and user expertise, to the underlying code.

In the last decade the mainstream software engineering community has recognised the need to develop strategies for programming-in-the-large. This paper proposes the need for *program comprehension-in-the-large* and describes the work of the Esprit DOCKET project which seeks to provide such a support capability.

The DOCKET project has developed a prototype environment to support the development of a system model linking user-oriented, business aspects of a system, to operational code using a variety of knowledge source inputs: code, documents and user expertise. The aim is to provide a coherent model to form the basis for system and program understanding and to support the software change and evolution process.

1. Introduction

In recent years, there has been an awareness that the software industry is entering a new phase in which a major proportion of IT resource is directed to the maintenance and support of existing systems. Whilst precise figures vary, there is general agreement that over 50% of software costs relate to ongoing maintenance and support [Parikh & Zvegintzov, 1983]. A number of tools are commercially available which assist maintainers support large, commercial applications, however these primarily focus upon a syntactic analysis of source code and hence they fail to adequately address the semantic content of such representations [Layzell & Macaulay, 1990]. The result is that current maintenance support tools offer little support to program comprehension in a business-context, thus making it difficult for software maintainers to place their own decisions and actions in context. The result in many cases is poor maintenance, decisions on code changes and change request prioritisation being based upon short-term expedients, rather than a properly justified business-oriented case.

This paper describes the work undertaken by the Esprit DOCKET project, which aims to build upon the current research and tool base and provide an intelligent reverse engineering toolset, capable of addressing semantic-related issues. By supporting *program comprehension in-the-large*, DOCKET seeks to provide a more complete context for maintenance decision-making and thus enhance the quality of the maintenance process.

The DOCKET project consortium consists of four industrial partners and two academic partners: Computer Logic (Greece), CRIAI (Italy), Software Engineering Service (Germany), SOGEI (Italy), UMIST (UK) and Universidade Portucalense (Portugal). The project ran from October 1990 to January 1993.

2. Knowledge Sources

The aim of the DOCKET project was to produce an architecture, toolset and method which enabled organisations to gain a better understanding of their software assets and associated products. This is achieved by maintaining explicit links between the source code of operational systems, technical documentation and system and domain expertise held by humans.

For the purposes of DOCKET, each of the knowledge sources are referenced by the nature of their representation, level of formality and degree of reliability in describing a system. This latter factor is particularly important since it must recognised that human experts may be mistaken and documentation can be notoriously out of date. Three general categories of knowledge source were defined, as follows.

Dynamic Knowledge Sources

The term dynamic knowledge was used to relate to system knowledge obtained directly from humans. Within this classification, a number of subdivisions exist: domain experts, system IT experts and user experts.

The knowledge obtainable from a domain expert such as an administrator concerns general aspects of the domain and is likely to contribute to the higher-level understanding of a system, such as its goals, general components and how it relates to the business.

Knowledge obtainable from a system IT expert such as a systems analyst or a system designer concerns the design and implementation aspects of a computer-based system.

The knowledge obtainable from user experts such as clerical staff concerns the user's view of the use and implementation of a system. As such it is likely to contribute to the knowledge about the general design of a system, its interface and use.

Informal Knowledge Sources

Informal knowledge is defined in DOCKET to be anything represented in a semi-structured or informal notation, other than directly from humans. This gives the source certain characteristics in terms of consistency, although not necessarily accuracy, and as such can be distinguished from human experts.

A variety of documented knowledge exists about a system and can be classified as belonging to one of three general categories: domain documentation, system documentation and change request documentation.

The knowledge obtainable from domain documentation, such as a shareholders' report, is of a general nature and relates to the domain in which the computer system being analysed by DOCKET operates, rather than to the system itself. System documentation such as system and program specifications or user guides, on the other hand, relates to the design and implementation aspects of a computerised system and can therefore form the basis of a more formalised view of the internal aspects of a system. Finally, the knowledge obtainable from change requests relates to alterations requested by the users of a system. These could arise either because the system is operating in a manner other than that intended or because of requested enhancements or alterations to the system.

Formal Knowledge Sources

Formal knowledge is used to refer to documentation-based descriptions, but represented in a formal, machine-processable language. Whilst it is recognised that natural language documents may fall into this category, a pragmatic distinction is made between source code programs written using well defined languages and natural language documents whose semantics are much more difficult to comprehend.

Formal knowledge sources thus include source code, structured designs and specification documents represented in notations such as ER or DFD, together with any formalised representation of test cases. Such knowledge contributes to the understanding of a system's implementation and design limitations.

3. Background Technologies

In order to capture output from the various knowledge sources and build an integrated model of a system, a variety of background technologies had to be employed.

Source Code Analysis and Reverse Engineering Techniques

Source code analysis can provide the basic information for knowledge acquisition about an operational software system. It is in effect a set of *information extraction* processes, consisting of static analysis of sources (object code analysis is excluded) and testing (including object code).

The output of these processes is a programming-language-independent, low-level representation of the information extracted from the operational software system. This means that the information extractors also perform a *first-level abstraction* from the original sources, whose goal is to preserve the original information content while decoupling all other, subsequent analysis tools within the DOCKET toolset from the syntactic details of the sources. This enables existing source code analysers to be employed within the toolset, provided that a translator is written to convert to the intermediate representation.

Once processed, the intermediate representations of a source program can be further analysed by reverse engineering tools- an *information abstraction* process which, by combining the results of low-level source code analysis and additional knowledge, aims to reconstruct the following typology of targets:

- representations of software which can be used directly by the maintainer, and possibly compared with artefacts of the forward engineering process; candidate outputs include JSP or Warnier diagrams, data flow diagrams and structure charts, etc.
- representations of the information extracted from code at a level of aggregation, synthesis and abstraction such that the population of DOCKET's knowledge base and the integration with other sources of knowledge would be facilitated.

In *first-generation* reverse engineering tools, the issues of reconstructing relevant links between software components and the selection and application of appropriate information hiding, aggregation and encapsulation techniques are addressed only mechanically, and any choice is made on the basis of standard, implicit and generic default assumptions; even many of the existing knowledge-based prototypes tend to search in the code for occurrences of very general programming clichés and concepts. The DOCKET goals include second-generation *intelligent* reverse engineering tools, which tend to drive and refine the production of abstractions, as well as the search for occurrences of specific concepts in the operational system, based on multiple sources of information which are specific to the user and application domain.

Knowledge Elicitation

The human knowledge sources available to DOCKET have been classified into three types: domain expert, system IT expert, and user expert. The knowledge held by these groups differs, but each of these sources contains knowledge about the system which is valuable for DOCKET to access, and which may not be available in any other form. In eliciting knowledge from these sources, account has to be taken of ergonomic and practical considerations.

It is widely reported that people find it hard to know what they know. DOCKET therefore needed to create the role of a *knowledge facilitator* to acquire knowledge from the experts, and to coordinate the input and representation of the knowledge within DOCKET. The knowledge facilitator will acquire knowledge from the dynamic sources by conducting informal interviews with experts, as a means of acquiring overviews of the domain and the system and by acquiring specific knowledge such as organisation hierarchies from the experts.

Text Processing and Analysis

Central to the processing of informal sources is the ability to handle natural language. The text processing element of DOCKET therefore had three primary concerns: (i) the investigation of methods for extracting knowledge from the variety of textual sources which will typically be associated with a software system; (ii) the development of appropriate analysis tools and (iii) the subsequent presentation of the document base back to the maintainer, in a structured manner.

A UMIST study showed that maintainers whilst relying upon source code for system understanding, would make use of documentation if the search space of the documentation could be reduced- by making the documentation on-line and by maintaining explicit links to aspects of an operational system.

It was established at an early stage of the research that a complete syntactic and semantic analysis of the text was not feasible, given the quantities of documentation to be processed and the current state-of-the-art in computational linguistics. Short-cuts which still achieve a sufficient degree of *understanding* have to be used in the analysis process to locate the key *concepts* and *events*. Thus, there is a critical distinction to be made between the *Text Processing*, approach taken by DOCKET, and the more detailed analysis techniques seen in most *Natural Language Processing/ Understanding* (NLU) systems.

To achieve the desired short-cuts, a system using a combination of some of the ideas of *sublanguage analysis* and *text skimming* was used. A set of heuristics, based on indicator phrases and textual format allows the key features of a text to be located by pattern matching and only a very restricted syntactic analysis. These heuristics included the following items:

- the location and marking of headings and subheadings, which often introduce new concepts and events
- the examination of text rendered in particular typographical styles such as boldface, italics and special fonts, since these are often used to highlight important items
- the location and marking of lists of items: these may occur in several forms and can provide information about both the structure and the meaning of concepts; they can be easily identified from format clues and often have a preceding explanatory sentence
- the identification of concept definitions, which are often introduced by indicator phrases such as: *this <concept> is defined as...* or *a primary objective is...*
- the identification of events, which are often introduced by indicator words such as if and when
- more detailed examination of sentences mentioning an already identified key concept since such sentences are often used to introduce a new key concept.

Due to the complex, time-consuming nature of the process of extracting information from textual sources, as well as the unrestricted nature of its inputs, the text analysis element of DOCKET is less able to contribute a comprehensive model of a software system than source code analysis or dynamic source analysis. The analysis approach taken is, essentially, to locate and mark significant concepts and events in the text and to provide a list of these to the other DOCKET processing streams in support of their activities.

System Modelling Notations

At the heart of the DOCKET architecture lies a modelling formalism capable of representing any system description, from its implementation view through to a high level organisation view. Analysis of existing work revealed that four levels are identifiable [Black et al, 1987; van Griethuysen, 1982; Olle et al, 1988] as follows.

The *world level* represents a real world view of an information system identifying the long-term objectives of an organisation. This level also contains a description of the functional areas of the organisation and maps these to its structure and human resources. External entities which influence the organisation are included, capturing their effect on the organisation's activities.

The *conceptual level* represents an implementation-independent abstract view of an information system. It contains concepts capable of representing the three different information system perspectives of activity, data modelling and behavioural modelling.

The *design level* represents a pure functional decomposition of the target, computer-based information system, including user interface aspects.

The *implementation level* represents a physical view of the implemented software system. It is included in the global system model in order to allow DOCKET users to reason about the physical system components and the relationship of these components to the higher levels of representation.

Existing information system modelling schema were analysed and a set of common concepts were identified covering the four levels. This work formed the basis of the development of a *global system model* which would form the core of the DOCKET repository mechanisms.

4. The DOCKET Architecture

Architecture Overview

Figure 1 shows a conceptual and linear view of the DOCKET system architecture. It is a conceptual view as it is essentially a static view, highlighting the main architectural components, rather than how the components are used. This latter, dynamic view, is explained in the following section on the DOCKET Method. The main aspects of the architecture are as follows.

Importation Phase

In the importation phase, raw knowledge about the system from original knowledge sources (termed *external sources* in DOCKET) will be processed into internal, machine-readable forms using input tools. In the case of source code, this is through a set of source code analysers. In the DOCKET project, analysers for COBOL and C are supported and converts code to a language-independent representation scheme called CSI.

For documents, industry-standard scanning and text conversion tools are used, followed by a purpose-built tool to convert text to the standard markup language, SGML. The markup tool is written in C and runs in a PC environment and provides the facility of manipulating all document-based input using a standard representation language.

Finally, expert input is made through a set of tools supporting graphical hierarchy and network models. These models include: goal and organisation hierarchies, activity/task hierarchies, dialogue graphs and conceptual graphs (Sowa, 1984).

The model editors are implemented on a Sun/4 workstation using a mixture of C and Prolog, with Motif handling interface issues. C was selected to implement the main model editor operations, which outputs Prolog predicates representing edited models. Additional Prolog statements are then used to automatically verify the models, with C handling any error reporting.

Extraction and Abstraction Phase

In extraction and abstraction phase, analysis tools use the internal forms as input in order to produce, through abstraction, a *structured form* of the essential knowledge from each source; the analyser tools may consult already existing structured forms in this phase. The term structured form was employed to distinguish between *how* a knowledge source represents something and *what* it is actually representing, i.e. a distinction is made between *form* and *content*.

For source code, the structured form is an enhanced form of the CSI language, which contain higher levels concepts beyond the immediate source representation. For documents, the enhanced form is an SGML representation in which section, headings and key concepts have been marked using the analysis heuristics described earlier. And for human expert representations, the structured form consists of all input models converted to a set of standard conceptual graphs.

Integration Phase

In the integration phase, a consolidation and resolution tool will integrate concepts from the different structured forms to populate a *global system model* and will resolve conflicts in the knowledge supplied from the different structured forms. This latter process is not fully automated; intervention is possible by a DOCKET analyst via the DOCKET model administration tool.

At the heart of the architecture is the global system model which incorporates and link concepts relevant to the system at different levels of abstraction- from an organisational view of the system, down to an implementation view. This is described in more detail in the next section. A browser allows the DOCKET user to access the knowledge in the global system model and to navigate through it by means of a hypertext-like facility.

Other Processes

In addition, as concepts relevant to the domain and to the system being analysed are acquired, they will be stored in a concept thesaurus which can be accessed by any of the tools. The thesaurus provides a rapid access directory of all concepts known to the system and provides facilities for synonym and homonym specification, central definition handling and link management between associated concepts.

The Global System Model

The Global System Model is key to the integration aspects of the DOCKET architecture. The model can be regarded as a multi-tiered, meta-model capable of representing aspects of an information system in an original notation-independent fashion. Furthermore, the global system model can maintain links between different abstract levels of description of a system, so that an explicit link can be made between a data file, the conceptual level entity that the file implements and an external institution. Links can also be made to the knowledge source origins of an element of the global system model, thereby providing traceability to the knowledge source. Figure 2 provides an ER model summary of the model's concepts and relationships.

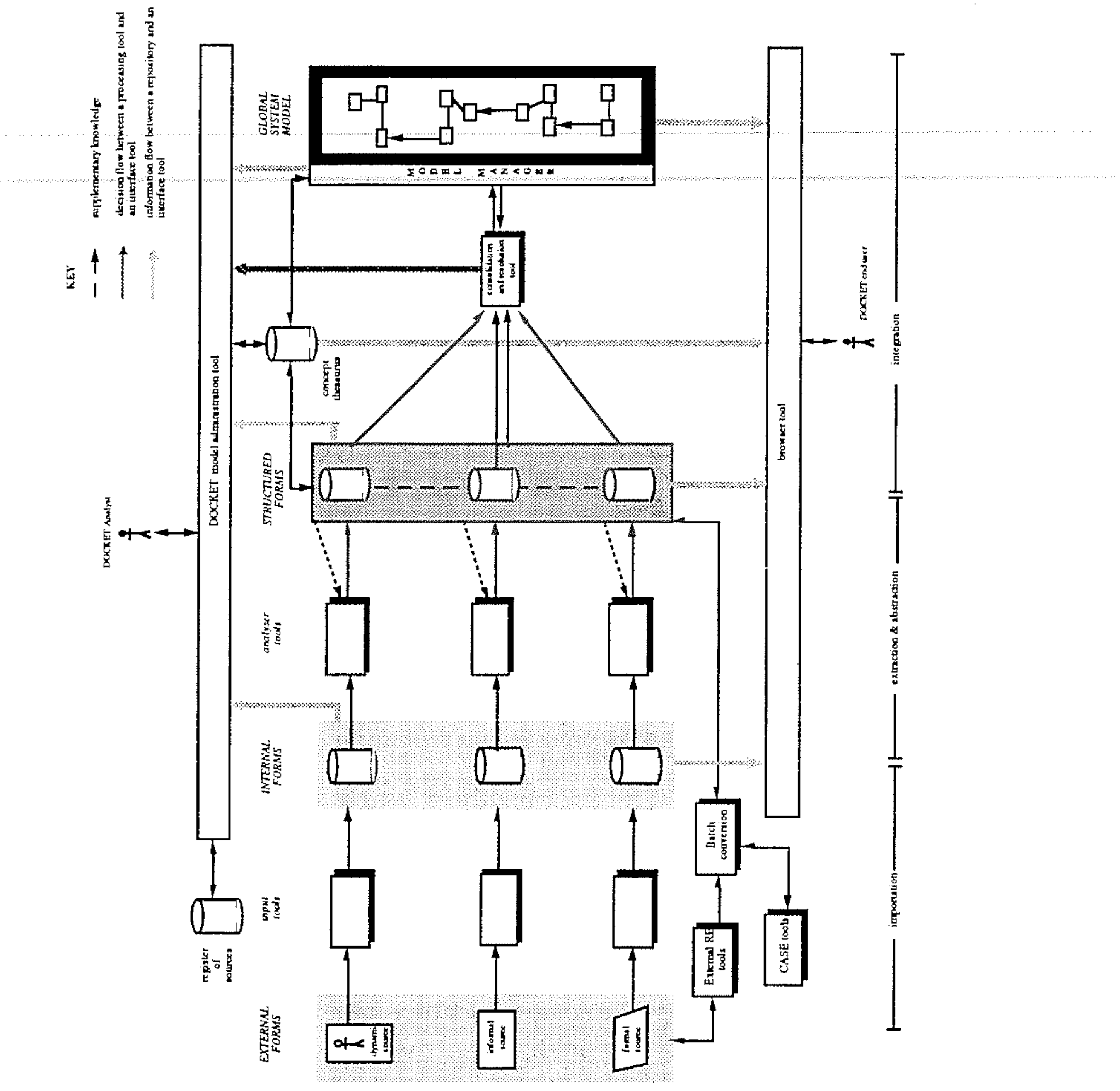


Figure 1: The DOCKET Conceptual Architecture

The global system model is implemented in the DOCKET architecture using SML (System Modelling Language), based upon the *Conceptual Modelling Language* produced by the Esprit DAIDA project. It is an object-centred language which supports the organisation of knowledge into general classes (concepts) and particular instances of classes (objects). Like similar languages, it supports the notions of classification, specialisation and aggregation.

In addition to the main global system meta-model, a concept abstraction hierarchy was also produced to allow for the partial classification of concepts. The concept abstraction hierarchy contains all the global system model concepts as leaf nodes on the tree. The concepts are then abstracted to higher level, more general concepts. This additional structuring therefore allows DOCKET analysis tools to start to insert concepts into the global system model, even when a full classification of a concept is not possible. This is particularly the case with text analysis tools, which, without full semantic analysis, cannot easily classify a term such as *book* without reference to other knowledge sources (is it a process or object?).

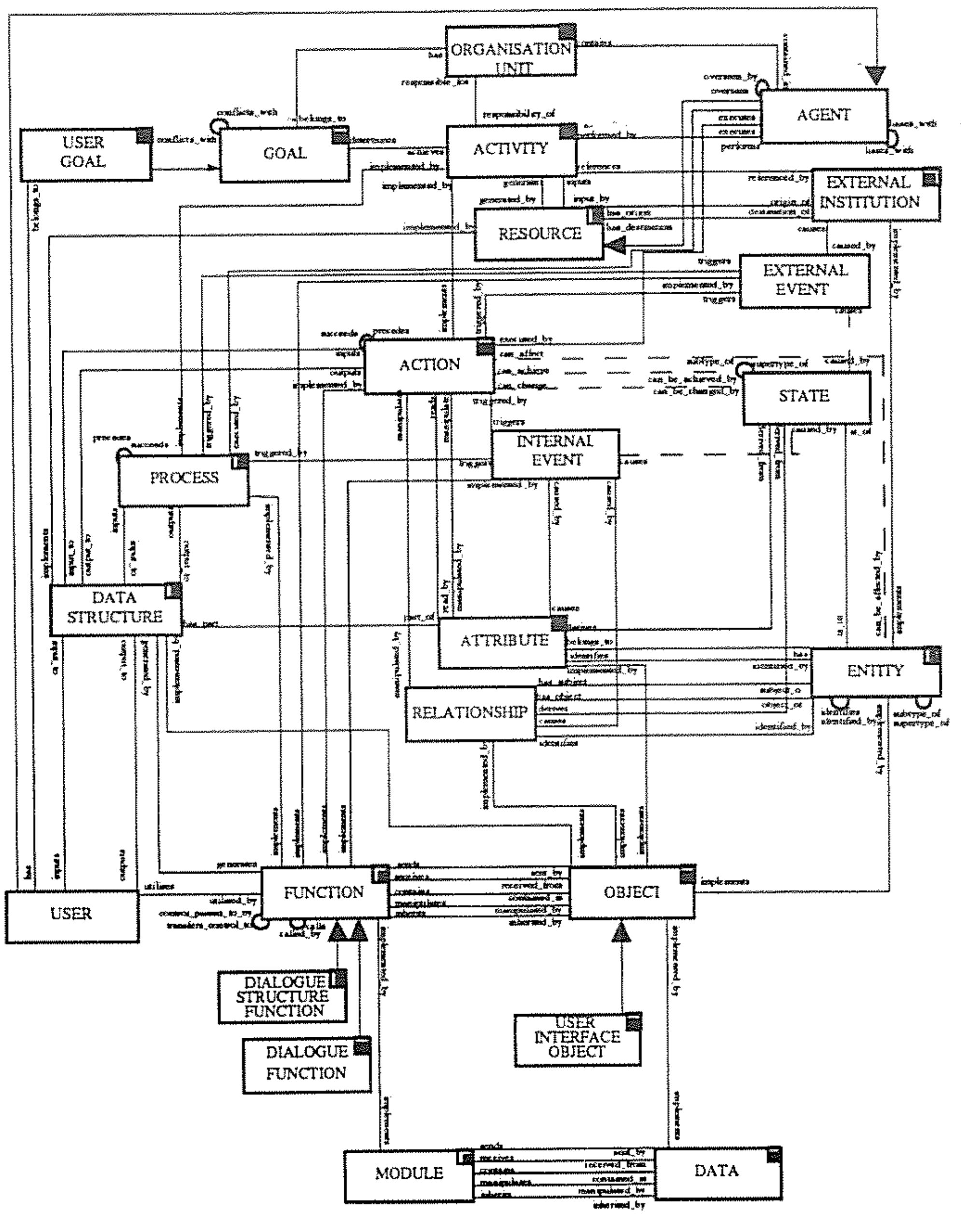


Figure 2: The Global System Model

Versioning

Given that the global system model is intended to support the understanding of an evolving system, upon which maintenance and enhancements are conducted, it is important that the model supports versions and variants of the application system under study.

There are essentially four choices available to support versioning. The first case concerns the application of reverse engineering where the current information system is analysed and the design recaptured. From here either the design is kept and used to restore design documents as a baseline for further maintenance, or the system is re-implemented from the resulting Global System Model. In either case the Global System Model is only created once, has a point where work on it stops, the model is treated as if complete (and immediately out of date) and is thrown away after use.

The next case is the one found in most contemporary programming environments, where only the currently saved version of a system is worked on. This does not mean that backups are not kept. They are kept but they are not reasoned about. This prevents historical reasoning such as: why is this sub-system included, what depended on it in the past? Does the system that depended on it still exist? In addition, there is often the case where a maintainer has the current system, the target system, and the target system implementation which is halfway between. In this case historical information is essential.

The third case relates to when many discrete snapshots of the Global System Model are taken. After the latest version is established, all new changes to the Global System Model are collected but not included until the next version is established. This batch method means that the Global System Model will be out of date and lose information concerning time-ordering of changes between selected check points. In addition, for any object in the Global System Model that changes more than once between checkpoints, information will be lost.

The final case is the *never delete* option, where the objects in the model are never deleted, only updated. The inadequacy of the above approaches and the clear advantage of the last case: the continuous model of the Global System Model, are decisive in its selection as the best option. Whilst being the most technically difficult, it maps best to the maintainers' needs for the reasons above and hence is the version model selected in the DOCKET project.

5. The DOCKET Method

In addition to the basic toolset, the DOCKET project has also developed a method of use of the toolset. This method can be regarded as a more general technique of managing an organisation's software assets, although clearly this is assisted by the toolset. The method consists of six phases, from initial planning through to the exploitation of the DOCKET global system model.

The overall objective of the domain definition phase is to identify the domain for which DOCKET will be used and the applications contained therein. DOCKET is intended to support only a single domain, although it can support many applications within that domain. Therefore, in this initial phase, strategic decisions must be made to determine what should be supported. The tasks within this phase aim to identify possible domains and applications and, through a procedure of analysing the critical relationships between domain and application, select a target domain and application pool. Factors to be considered include domain and application complexity, application stability/ rate of change and personnel turnover.

Inventory Phase

The overall objective of the inventory phase is to identify the main knowledge sources within the selected domain and application set and produce a detailed inventory of such sources. As a precursor to populating the DOCKET knowledge bases, the DOCKET Analyst needs to identify all the available knowledge sources and their interdependencies, so that a proper knowledge elicitation plan can be constructed. This will subsequently improve the knowledge capture processes, ensuring maximum integration in the global system model and minimum, unnecessary redundant knowledge capture (although some duplication will be desirable).

Initial Population Phase

The overall objective of the initial population phase is to prime the various DOCKET repositories, prior to use in task-oriented activities. The initial population of the repositories will be based upon the knowledge acquisition plan defined in the previous phase. It is in this phase that the majority of the DOCKET tools will be used, including the initial knowledge source capture tools. As part of this process, a systematic approach to ensuring the consistency of the various DOCKET models must be employed.

Two types of consistency checks are applied to the global system model. The first which may arise is one where an entity or relationship is missing from the global system model. For example, every **goal** must have a **relationship** with at least one **activity**, i.e. a goal must be achieved through some activity. This type of constraint is imposed by the relationship cardinality constraints shown for each relationship type in the global system model documentation. Violation of these constraints indicates that something is missing from the global system model and can be used to formulate questions to dynamic sources of the form, "what activities contribute to achieving the goal ...".

The second type of check concerns the explicit meta-rules applied to the global system model's underlying description, such as:

If a **goal** belongs to an **organisation_unit** then no subgoals of that **goal** may belong to a parent of that **organisation_unit**.

Clearly, it is in this checking procedure where the DOCKET toolset can save the most time and ensure consistency and model quality.

Task-Oriented Phase

Within the task-oriented phase of DOCKET, emphasis changes from the general acquisition of knowledge about the application and its domain, to a more task-oriented approach. This change is reflected in the manner by knowledge is acquired and who drives that acquisition process. The main tasks are as follows.

Identify Knowledge Requirements. The completion of a maintenance task will depend on the utilisation of information pertinent to that task. The first step is therefore to recognise the knowledge requirements of the task in hand, and to identify the repositories in the DOCKET system where the relevant information may be located. The type of maintenance or system-understanding activity will largely determine the nature and type of knowledge required.

Locate Knowledge. A successful retrieval request on the relevant repositories will produce the information required for the current maintenance task. If this is achieved, then the DOCKET user may proceed to perform the task in hand.

Acquire Additional Knowledge. If the required information is currently not available from the DOCKET repositories, this step is intended to acquire the missing knowledge. This will involve the use of the DOCKET toolset in any of the input, analysis or consolidation steps. The precise activities will be based upon the task in hand and the current state of the DOCKET repositories.

Perform Task. Depending on the nature of the maintenance task being performed, the knowledge contained in the DOCKET system may be enhanced by actual execution. Examples include the establishment of new links in the repositories or the insertion of new information into an experience base.

Consolidation Phase

At any stage in the use of the DOCKET system, the knowledge content of the repositories may be checked for consistency and completeness. This process is independent of any maintenance tasks and is intended to (a) produce a coherent Global System Model of the current target system, where this has not yet been achieved in the task-oriented phase and (b) identify those areas which require further knowledge elicitation. This step is to be performed either as a clarification procedure during the use of DOCKET within a given domain, or as a prerequisite for the exportation of the repositories to a new domain.

Reuse/ Export Phase

The contents of the DOCKET repositories may be reused within a new domain, as part of the initial population phase for that domain, or exported for use by other software tools (e.g. a CASE development tool). In either situation, it is assumed that the information to be reused is in a consistent state, produced by the consolidation phase.

6. Exploiting the Global System Model for Program Comprehension-in-the-Large

The exploitation of the Global System Model can be considered at two levels. Firstly, there is a requirement on the DOCKET system to allow the retrieval of information from the various knowledge repositories in such a way as to provide the investigator with necessary and sufficient information to satisfy the initial query. The second level is based upon the assumption that the access paths and information content associated with such retrieval operations may themselves be used in deriving useful, reusable information concerning the understanding of a system. The *landmarks* encountered in the search for information concerning a query may thus provide reference points for subsequent searches. In this way, an *experience base* of exploitation knowledge can be developed as a useful side effect of the exploitation process itself.

The DOCKET Browser provides facilities for navigating either between the various knowledge forms or within a particular knowledge repository, using the links which have been established during the population and task oriented phases. Intermediate forms are used to mark derivation links but are not directly accessible from the Browser. In this way the user of the tool is protected from unnecessary details contained in the models and can concentrate on those aspects of system understanding directly relevant to the current task. Entry points to the navigation system are therefore restricted to the external sources, the Global System Model itself and to the concept thesaurus. In navigating through the links held within the DOCKET system, the Browser maintains a list of beacons, marking the linked concepts. These may be edited by the user and may be given a special rating of anchor, if they are considered to be especially important. At the end of a session, the user of the tool may store the entire navigation history, or an edited version ignoring side tracks and dead ends. These routes may then be reused during subsequent exploitation.

Figure 3 shows a partial navigation route through the DOCKET repositories shown in the window headed Beacons. It shows a route from the external textual source, 'How to be a Warden' to the Global System Model. Links within the model have connected the activity 'room allocation', through the action 'allocate room' and the function 'assign_student_room', to the module 'ROOMCOMP'. The marked anchors can be saved to provide a direct link between the text describing one of the activities of a Warden, with the specific module from the source code which implements this function.

A note on the population of the DOCKET repositories must be made in the context of exploitation. It is recognised that populating the DOCKET repositories is a non-trivial task.

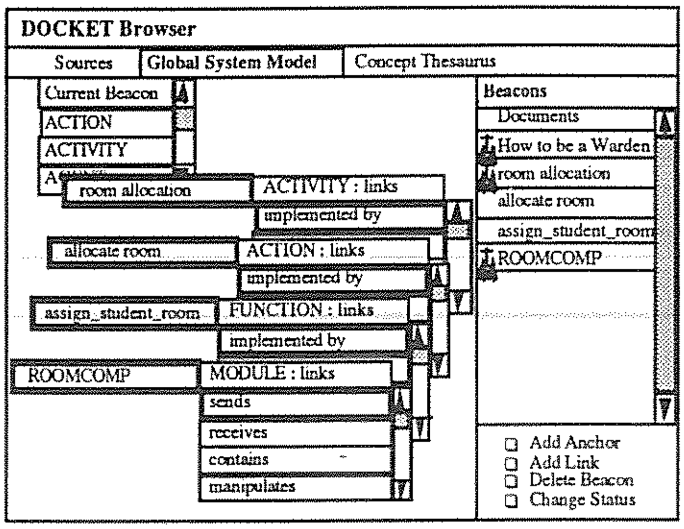


Figure 3: A Browser Session

In particular, there is considerable overhead in acquiring non-source code knowledge. An initial evaluation of DOCKET has shown however, that there are many benefits from the acquisition process in its own right. In particular, the *stock-taking* of software assets enables organisations to have a greater awareness of their IS development strategy. The DOCKET method is therefore carefully constructed to enable such benefits to accrue early in the population process.

7. Conclusions and Future Directions

In much the same way that the mainstream software engineering community recognised the importance of programming-in-the-large in the 1980s, so too must that subset interested in existing systems and the attendant need to understand programs recognise that program comprehension is more effective when conducted in a wider, knowledge-richer context.

The DOCKET project seeks to provide an environment in which there is a much closer link between operational program units and a model of the underlying enterprise it seeks to support. This is achieved through the input of multiple sources of knowledge about a system, which are integrated into a single, coherent model of a system. The provision of such a model provides the opportunity to better understand a program suite and make more informed software maintenance decisions.

Under the Esprit funding programme, the DOCKET project achieved a number of prototype tools and developed an overall population process which seeks to assist organisations in producing an inventory of software assets and providing the means to exploit them in program understanding. In a subsequent stage, the project consortium hope to continue the work to develop an industrial-level implementation and full, quantified evaluation.

Acknowledgements

The authors wish to acknowledge the contributions made to the DOCKET project by members of the project consortium: Computer Logic R & D (Greece), CRIAI (Italy), Software Engineering Service (Germany), SOGEI (Italy), Universidade Portucalense (Portugal); and the UMIST team members- Rosemary Woolfitt, Andy Hurt, Bill Black and Angela Balkwill. The work described in the paper was partially funded by the European Commission under the Esprit research and development programme.

References

Black, W.J., Layzell, P.J., Loucopoulos, P. and Sutcliffe, A.G. (1987) AMADEUS Project: Final Report, Esprit project 1229(1252), UMIST, UK.
van Griethuysen, J.J. (ed.) (1982) Concepts and Terminology for the Conceptual Schema and the Information Base, ISO/TC97/SC5.
Layzell, P.J. and Macaulay, L. (1990) An Investigation into Software Maintenance- Perception and Practices, in Conference on Software Maintenance, San Diego, IEEE, pp.130-140.
Olle, T.W., Hagelstein, J., Macdonald, I.G., Rolland, C., Sol, H.G., van Assche, F., Verrijn-Stuart, A.A. (1988) Information Systems Methodologies: A Framework for Understanding, Addison-Wesley.
Parikh, G. and Zvegintzov, N. (1983) Tutorial on Software Maintenance, IEEE Press.
Sowa, J. F. (1984) Conceptual Structures, Addison-Wesley.

"The Role of Testing and Dynamic Analysis in Program Comprehension Supports"

P. Benedusi ,V.Benvenuto, L.Tomacelli

CRIAI, P.le E. Fermi, 80055 Portici Italy, tel.+39 81 7863174

Abstract

Test cases, and the human activities involved in testing and error analysis play a peculiar and considerable role in the comprehension of a software product and process. This subject was systematically explored in the context of the Esprit DOCKET[1] Project (DOcument and Code Knowledge Elicitation Toolset), designing and experimenting second-generation Reverse Engineering processes which combine multiple knowledge sources. This paper describes the approach, and the architecture which have been designed for the capitalisation of the knowledge products of both existing Software Life Cycle activities and proposed knowledge-elicitation oriented testing strategies.

1. Introduction.

The role of testing and dynamic analysis in the elicitation of knowledge on programs is receiving increasing attention in the research community. The initiatives reported in the literature include the reconstruction of dynamic call graphs [1], the animation of Data Flow Diagrams reconstructed from code [2], the dynamic analysis of Entity-Relationship schemes with real data [3], the location of functionalities in old code [4], and the reconstruction of business rules from test cases [5]. In the Microscope system [6] execution histories and slices become part of a knowledge base, where they can be subject to reasoning processes and used to answer various classes of questions about the monitored program. It is thus important to research general conceptual frameworks, and supporting architectures for: i) the definition of the various types of knowledge obtainable by testing and dynamic analysis, and their integration with other knowledge sources involved in the comprehension process; ii) the definition of their specific contributions to Reverse Engineering processes; iii) the formalisation of test repositories combining the benefits of error detection and knowledge elicitation; iv) the introduction of systematic knowledge capture strategies in existing Software Life Cycles, for the cost-effective reuse of past comprehension efforts. In this paper we illustrate some contributions given to the above goals by the ESPRIT Docket research project, which aimed to support software comprehension by combining multiple sources of knowledge. The results include the method, the design and partial implementation of a knowledge elicitation and browsing system. Section 2 overviews the types of contribution of testing to various aspects of the software comprehension. Section 3 positions testing-based knowledge elicitation processes in the context of the Docket architecture; section 4 describes the conceptual schema proposed for test case repositories onto which the results of many types of testing and dynamic analysis can be mapped. The conclusions include brief considerations on the costs and benefits of the proposed strategy.

2. Test-related knowledge typologies

Traditionally, testing is conceived as the execution of software for error detection purposes [7]; dynamic analysis traces the internal software components at run time for white-box testing, for debugging or for the acquisition of knowledge on run-time performances. In the Docket project we explored the potential of these activities for software comprehension objectives. The starting points of this research were four basic observations:

i) test suites should be available at maintenance time at least for regression testing purposes;

ii) maintenance programmers regularly use program execution as a comprehension support, even when manuals and Reverse Engineering models are available. Documents written in natural language contain fundamental semantic clues, but may be partially out of date with respect to the current operational software; they may also contain many ambiguities and imprecisions;

iii) test cases are the passive or active subject of existing SLC activities (specification, test case design, debugging & error analysis, change request formulation, change analysis, regression testing) producing various kinds of peculiar and valuable knowledge;

[1] The EP.5111 Docket project was partially funded by the Commission of European Communities; it was carried out by a Consortium comprising UMIST(UK), CRIAI(I), SOGEI (I),Universidade Portucalense (P), SES (D), Computer Logic R&D (GR).

iv) test cases are important starting points for the capture and reuse of knowledge acquired empirically and lessons learned during the operational life of the SW product.
In the Docket method, test cases and dynamic analysis are used as indispensable or convenient support for software comprehension, for one or more of the following reasons:
a- to allow the acquisition or highlighting of relationships which are hard or impossible to obtain from static analysis alone
b- to allow the direct acquisition of functional knowledge
c- to aid in mapping functions to structures or vice versa
d- to focus and narrow the scope of investigations in system's structure and related documentation
e- to validate assumptions, and provide objective support or disconfirmation to information coming from other sources (code static analysis and Reverse Engineering, documents, experts);
f- to provide a bridge between different sources of knowledge about the software product
g- to provide examples which are particularly effective in the comprehension and learning process
h- to provide examples which highlight and characterise critical aspects of the software system which are the subjects of misunderstandings and comprehension faults.

The above subjects have been examined from two perspectives: the improvement of Reverse Engineering processes and models, and the reuse of knowledge produced in the Software Life Cycle.

2.1 Improvement of Reverse Engineering processes.

Models derived from static analysis of the internal structure of software components are limited in the precision and completeness of structural knowledge, in their semantic content and abstraction level. In this respect, two basic principles are exploited in Docket:
- testing may reveal relationships which are visible only at run time;
- testing can exercise only (a subset of) the feasible paths in the control structure of programs.
This second characteristic of testing can be considered both in terms of limitations and advantages. In positive sense, test cases *narrow* the scope of software analysis in terms of both semantic-functional focusing and structural constraining. Software analysis can in fact start from specific, actual behaviors to be explained by focused investigations in the software structure.
Dynamic analysis gives then a peculiar contribution to the systematic *acquisition and refinement of structural knowledge*. Static analysers, such as optimizing compilers, are often forced to make conservative assumptions on the control and data flow [8]. For instance, one typical reason is that the internal data flow of a called subroutine is not known, and even interprocedural static flow analysis is not applicable since only the object code is available; another is that the static analyser cannot know which control paths are actually feasible at run time [9]. The completeness and precision of models formally derived by static analysis is limited by the presence in code of more problems: table-driven mechanisms; dynamic calls to subroutines whose name is known only at run time (expecially in cases when this name may be defined by an external process); jumps to label variables; SQL statements built by concatenation of string variables depending on program input, etc.
By utilising runtime information for specific input values, for example, R.Gopal [10] defined dynamic slices containing only those statements that *actually* affect the value of a variable at a specified program location, in contrast to static slices which include the components that *may* affect it. More in general, in Docket we define *the dynamic version of a structural model* derived by the analysis of code, and compare it with the corresponding static version. With reference to generic dependence graphs [11], the set Ds of dependencies between software components represented in a static model may only partially intersect with the set Dr of dependencies obtained by dynamic analysis. In particular we define:
1) Da = Ds - Dr the difference set containing the dependencies which appear only in the static model.
Da may contain infeasible dependencies, resulting either by undesired *overestimation effects* (such as the assumptions that all paths are feasible), or by the deliberate decision to *highligh potential bindings* between software components, which may become effective at run time after a minimal change in the software [12,13]. Da may also contain feasible dependencies, which were not revealed at run time since the testing was not exhaustive in respect to them.
2) Db = Dr - Ds is the difference set containing the dependencies which appear only in the dynamic version of the model; Db contains the dependencies which were revealed only by dynamic analysis.
3) Dc = Dr $\cap$ Ds is the intersection set containing the dependencies revealed by both static and dynamic analysis. Even in this case, dynamic analysis may record additional information, such as the specific values of information flowing between the user and the software or between software components. The same set of exercised structural components may in fact perform totally different functions, depending on the value assumed by some data (i.e. table-driven mechanisms). Also some values, expecially those involved in man-machine interactions, may be particularly meaningful to the human reader when they evoke domain concepts, expecially when these values appear in user documents

containing descriptions of concepts and events. Dynamic analysis thus provides an important aid in linking structural and functional aspects of software.

Mapping function to structure is defined as the process that starts from a function which is either known or expected and tries to locate the software components responsible for it. Test cases provide *observed behaviors* to be explained in terms of internal structures; the observations may also concern *unexpected behaviors*, which were neither predicted nor assumed based on other investigation processes.

Mapping structure to function is defined as the process that starts from the examination of internal components and tries to discover their functional role and purpose. This investigation can be done by examining the run-time direct or indirect influence of a given set of components on certain outputs, or input/output behaviours.

However, the *acquisition of functional knowledge* is itself a problem for both the end user and the maintenance programmer. One aspect of functional comprehension is to know exactly what a program does in terms of mapping between phisical inputs and outputs; another is the comprehension of the meaning and conceptual goals of this concrete behavior, i.e. its mapping to user domain concepts. Testing is thus exploited in Docket for the *direct acquisition of cause-effect relationships*, by recording and formalizing the joint, complexive result of many software components in combination. Especially for complex systems, the perception of the whole effect of their components in a given situation is in fact computationally hard to obtain by formal reasoning on system models. As in many problems of physics, simulation or even direct exercising of the real-world system may be indispensable to know what happens. In addition, the *comparison of the simulated model's behaviour with that of the real system* is fundamental to validate, refine and perfect the model.

The goal of testing techniques as boundary value analysis [7] is redefined here to acquire *boundary an constraint knowledge*, i.e to know which are the actual boundaries of input and output data domains and the user operation sequences considered valid by the system, and/or to know the behavior around boundaries. This kind of exercising often reveals subtle aspects of system structure and behaviour, and helps in understanding the role and criticality of some parts of it.

Test cases can directly reproduce examples reported in user manuals, or be defined to check conceptual descriptions found in them. In the Docket Method this is used to establish a network of semantic links between documents and code [14,15]. This is an example of using *informal knowledge sources* to aid the mapping of code to domain concepts. According to Biggerstaff's DESIRE approach [16], informal sources are indispensable to reach higher levels of abstraction in design recovery; in the Desire system and its evolutions, neural networks were trained with examples combining static structural patterns, meaningful code identifiers and comments. This approach is competitive with respect to abstractions based on totally formal derivation techniques [17], or cliche recognition using only formal knowledge [18]. Among the problems of this approach are the presence of meaningless or misleading internal names, false synonims, the ambiguity of natural language sentences, and the possible lack of meaningful comments. Enhancements and perfections are possible with *dynamic examples* providing both "meaningful" and precise behaviors; this is evident expecially when components of the man-machine interface are exercised, since its external language and behaviors should be meaninfgul in the user domain, and the terminology very close to that used in user domain documents.

Examples however are fundamental to train humans. Thus test cases can provide real examples, to be selected and recorded for their explanation power, or their determinant role in validating/rejecting assumptions.

2.2 Test-related knowledge produced in the Software Life Cyle.

In addition to knowledge elicitation oriented testing, Docket sustains the saving and reuse of the peculiar knowledge produced by many activities of existing Software Life Cycles related to testing. In *alpha testing*, test case design and error analysis force the analysts to reflect on the software product being released from a different point of view than that of the developer. The design of successful test cases or code inspection sessions involves considerable human ability; deciding what is an error and what is not refines the comprehension of functional and non-functional specifications. In the debugging phase, further test cases can be used to provide the additional information requested by inductive or deductive assumption-based processes [7].

Error analysis is a bottom-up process that starts from observed failures, tries to locate the faults in the implementation level of the product, and then possibly investigates their causes in higher levels of software: the error might have been originated in the coding phase, or low-level design, high-level design, and so on. This is a sort of *reverse engineering process starting from test cases*. The documents produced by alpha testing thus yield specialist knowledge which is very important to the comprehension of both the product and production process. *Beta testing* and *corrective change requests* concern test cases which were not explicitly designed, but empirically gathered from problems occurred in the user's operational environment. These cases are important in that they represent situations, and possibly hidden aspects of the product, which were not envisaged or explored in alpha testing; they may not only reveal ordinary problems, but also empirically higlight aspects of software which are the most challenging and difficult to comprehend. The analysis of what and why something was failed to test or to fix, higlights *what is important to know* to maintain the system. The information sources are in this case: the user problem report; the problem test data possibly provided together with that report, if any; the test cases made by the maintenance analyst to reproduce the problem and fix it; the parts of system model and documents traversed in studying the problem; the hypoteses made, the diagnoses and motivations of the solutions; the results of regression testing after the changes. These sources are first used in Docket to establish links between the test cases and the document parts involved in problem analysis and fixing, and then to create semantic links between these document parts and the code parts interested by the test cases [14]. Also empirical observations made during the operational life of a product may be naturally expressed in form of test cases and assertions related to them. Finally, as proposed in [17,20] test case repositories can play an *active role* in the maintenance process: the change requests can be partially expressed in terms of new test cases and modifications to existing ones; these test cases can be effectively used in change and impact analysis, in order to focus and narrow the scope of investigations in both documents and code. This possibility raised further justifications and requirements for the design of the test case repository schema.

3. Testing in the Docket architecture.

Fig.1 shows the placement of testing-based knowledge acquisition in the general Docket architecture. The basic classes of knowledge sources used in Docket are dynamic, informal and formal. *Informal knowledge* is expressed in natural language in various kinds of documents, including user domain documentation, user manuals, technical documentation, problem reports and change requests.
Test cases and their related information may embody formal, informal or dynamic sources of knowledge. Test data constitutes a formal source in so much as it is often machine-readable and processable. The Docket internal forms level includes all the already machine-readable objects: source code, executable code, and the existing user test case repository (if there are any) containing test data. However, test case documentation, such as checklists and user manuals used as oracles, are regarded as semi-formal or informal sources of knowledge. Human experience about the system and its behaviour is also regarded as a non-formal, ie. dynamic source as is the result of the problem solving activities involved in the testing process. The knowledge contained in Internal Forms is organised and abstracted into a set of Structured Forms by analysis tools, which may make use of knowledge from Structured Forms of other sources. The integration phase comprises the construction of the *Global System Model (GSM)* and the resolution of conflicting information. The GSM formalism [21] is capable to syntetise knowledge expressed at a variety of levels of abstraction, from high level statements of business goals and objectives to low level concepts such as data and procedures used to implement the system. Traceability links interconnect the GSM, the Concept Thesaurus, Structured forms, and Internal forms. The operational software system is the executable product on which knowedge is to be acquired; its exercising produces test case results. Before testing-based knowledge acquisition activities start, it is important to build an initial baseline model of the software product, by activating a first, system-wide Reverse Engineering process starting from static analyis of source code. This creates a reference structural skeleton, onto which to map the results of further investigations involving testing, dynamic analysis, and more focused reverse engineering steps. Individual source units are submitted to static information extraction processes, which produce programming-language independent, low-level structured forms(IF1) in the Code Structured Form repository [22]. This is a first level abstraction step, whose goal is to preserve the original information content while decoupling all the other Docket processes from the syntactic details of the sources. The IF1 preserve links allowing the cross-referencing of its components with the original source components. The source analysis process tags those parts of IF1 output form that may be affected by incompleteness and/or indetermination due to the limits of code static analysis [23], such as those indicated in section 2. This information can be used to activate some compensative processes invoking additional knowledge sources: testing and dynamic analysis, or documents and human experts. Information abstraction steps post-process the IF1 representation, and produce further levels of structured forms (IF2); these represent the structure of the software product at various levels of granularity. The RE process includes the identification of the basic data and processing components, the reconstruction of various kinds of relationships between software components [24, 25], and the production of System Graphs such as Structure Charts and Hierarchical Data Flow Diagrams [26, 21]. *Filtering and enrichment* steps produce views of the Re models which improve their information discrimination power, and enrich their information content & semantics with test case-related

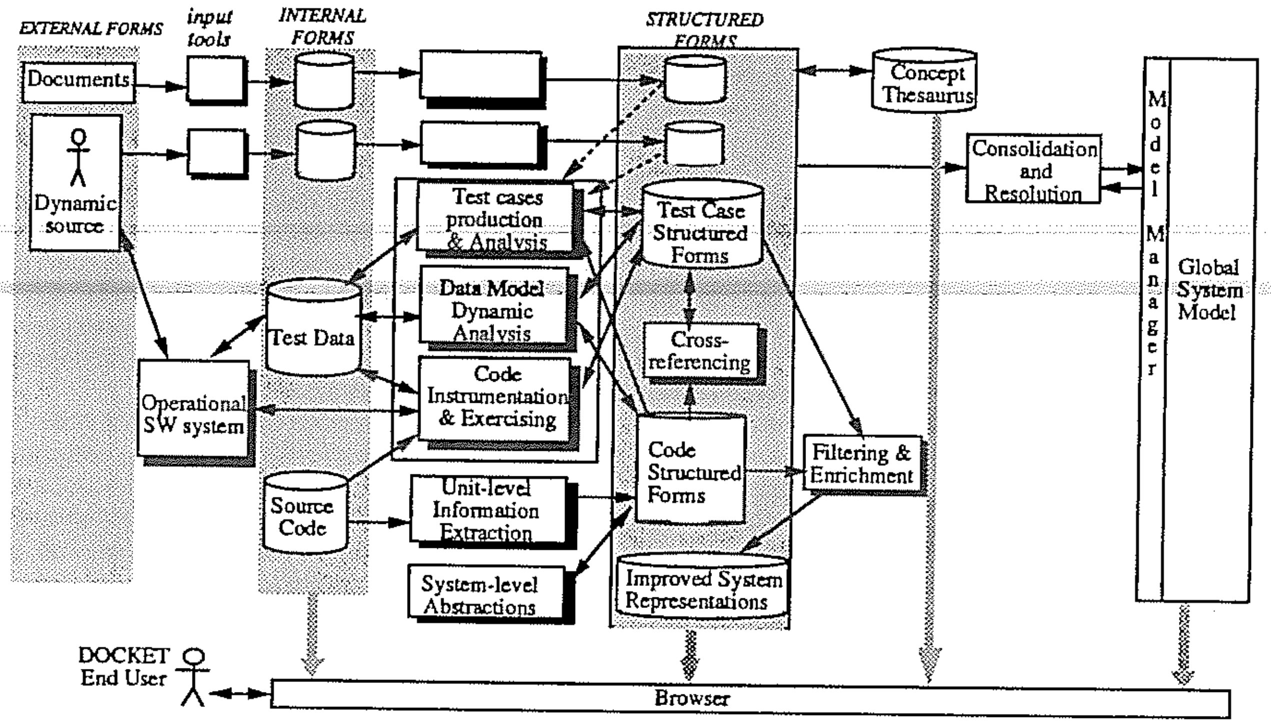

EXTERNAL FORMS
input tools
INTERNAL FORMS
STRUCTURED FORMS
Documents
Dynamic source
Concept Thesaurus
Consolidation and Resolution
Model Manager
Global System Model
Test cases production & Analysis
Test Case Structured Forms
Data Model Dynamic Analysis
Test Data
Cross-referencing
Code Instrumentation & Exercising
Operational SW system
Code Structured Forms
Filtering & Enrichment
Source Code
Unit-level Information Extraction
System-level Abstractions
Improved System Representations
DOCKET End User
Browser

fig.1

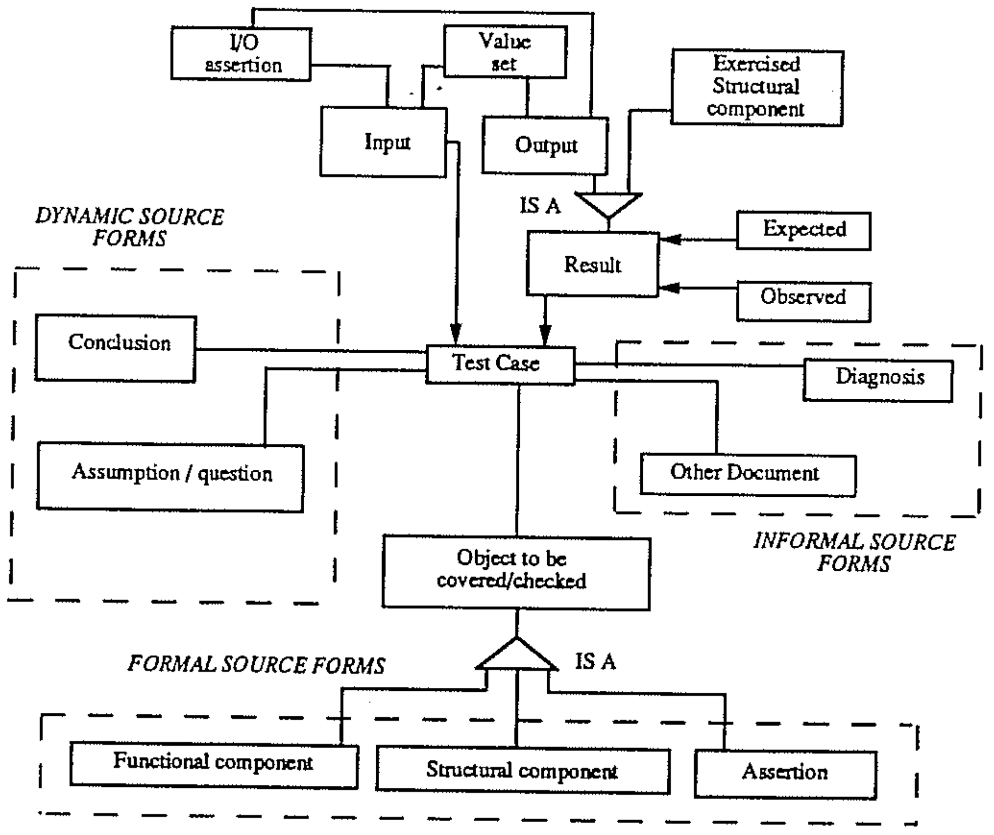

I/O assertion
Value set
Input
Output
Exercised Structural component
IS A
DYNAMIC SOURCE FORMS
Result
Expected
Observed
Conclusion
Test Case
Diagnosis
Assumption / question
Other Document
Object to be covered/checked
INFORMAL SOURCE FORMS
FORMAL SOURCE FORMS
IS A
Functional component
Structural component
Assertion

fig.2

knowledge. *Filtering* steps are designed to discriminate details to be aggregated, encapsulated, hided or highlighted in the information coming from either static Reverse Engineering or dynamic analysis. The filtering is aided by technical-domain knowledge on the characteristics of the HW/SW platform, Operating System, DBMS, TP Monitor, System Libraries and utilities, standard packages used by the application, and their typical influence on program and system composition. This knowledge makes it possible to classify and discriminate various kinds of software components at different levels, ranging from entire modules or data structures to sets of individual elements inside them. Experiments of such filtering were conducted on a COBOL Microfocus application [27], obtaining consistent simplification of structural models, and facilitating the selection of the relevant information from the details produced by dynamic analysis. The *enrichment* of Reverse Engineering Models results from the combination of static model building results with those of testing and dynamic analysis of the software product. Test cases are used here to confirm or reject relations, assumed roles and semantics of components existing in the model derived from static analysis. This model can be augmented with new relations and component roles derived from test cases. Test cases also contribute in providing objective support to the information passed to the GSM building and consolidation process. The enrichment process is aided by cross-references between the results of dynamic analysis, which are stored in the test case repository, and the code structured forms. The production of new test cases may be based upon i) the information derived from informal and dynamic sources, and ii) the integrative knowledge elicitation needs emerging from the static analysis. The analysis of existing test cases aims to exploit and improve the existing error-detection-oriented test cases for knowledge elicitation. Where the information on existing cases is not complete, code exercising is activated to rerun the test based on the original input data; then the results are analysed in order to reconstruct the relevant information. If the original input data are missing, a substitute instance has to be reconstructed based on the available (informal or formal) test case description. Both old and new test cases need to be classified, selected and qualified based on their value for knowledge elicitation. This activity may apply both in the design of new test cases and in the attempt to maximise the reuse of existing ones, if any. In regression testing after maintenance operations, it is possible to apply a selective revalidation process: the analysis of changes and existing test cases defines the test cases which are still valid and relevant for knowledge elicitation about the system, and those which need to be re-executed and possibly modified, and identifies any possible need for new test cases. For this purpose, strategies originally conceived for error-oriented retesting [28, 29, 30, 31] can be profitably applied.

Code-level testing may be performed in many ways: i) directly executing the operational code; ii) instrumenting the source code with probes, and then compiling and executing it; iii) submitting the source code to an assertion-based dynamic analyser [32].

Code instrumentation, compilation and execution can be performed in Docket by exploiting the same tools and environment used by the developer or maintainer in traditional testing activities. Two types of assertion-based testing strategies are possible:

1- given a set of assumptions and expectations on a software component, specify them in terms of assertions on the inputs and on the results; then validate the assumptions made by observing which assertions are verified or violated at run time, and which software parts are exercised correspondingly;

2- given samples of the actual input-output behavior of a software component, transform them into sets of relationships between assertions, involving data transformations and program paths.

The first strategy has been experimented in Docket by exploiting the PC-TEST tool [32] which exercises the code using both data provided by the user and statements written in a formal assertion language. For each software module under test, the results of this kind of dynamic analysis include: assertions (either verified or violated at run time), a log of the data exchanged during the man-machine interaction, a trace of the paths traversed (exercised branches) and the dynamic usage of data structures. The second strategy has been explored by H.Sneed [5] with a prototype tool including, in addition to program instrumentation and test monitoring facilities, an assertion generator and a database auditor, and was used to derive business rules at the transaction level from test case results. Both the above strategies stress the role of test cases in connecting functional and structural aspects of the software. The assertions can be used as a compact and expressive means for denoting valid and invalid data domains; they may involve both data of the external user interface and internal variables exchanged between software modules. However, the tools mentioned work with implementation-level assertions, i.e. assertions directly referring to components of the implementation.

In a comprehension process, a set of implementation-level assertions may be the corresponding of assumptions and conclusions referred to higher software levels (fig.3), or user domain concepts. Higher levels of testing may be performed using the data model dynamic analysis ER-TOOL [21,3]; it makes it possible, for instance, to exercise the ER schema of a user's application data with sample transactions and real data values.

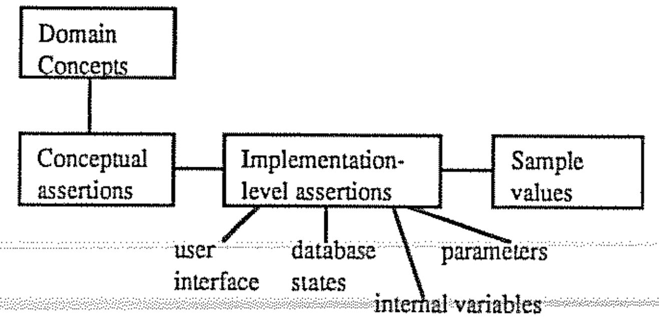


fig. 3

The schema is represented in a local knowledge base, which is capable to represent assertions also on the state of the database, and to support reasoning on the different transaction behaviors which may depend on different initial states. The ER schema can be exercised with the same real-world test data used to test the operational product; it is thus possible to compare the logical behaviours of the system and of the schema. This way of checking assumptions at ER level can be used both to validate and perfect a schema derived from Reverse Engineering, and to improve the human comprehension of the schema itself. In addition, Docket provides more general ways for the acquisition and formalisation of assertions made by dynamic sources. A *Transcript management tool* [21] analyses machine-readable text to extract concepts relevant to the domain. The text may be the transcript of an interview with an expert, or it may consist of notes entered directly to a notepad by a maintenance programmer. The concepts are stored in the Concept Thesaurus, and the network of semantic links between them is represented in form of Conceptual Graphs [33,34]. A *Dialogue Graph Editor* has been designed to allow the representation of knowledge referred to the man-machine interface of the product in the form of dialogue graphs, which are a specialisation of conceptual graphs. With reference to the schema of fig.1, it is important to note that *multiple knowledge acquisition lines can be activated togheter.* For example, an expert interview can be organised involving the observation and commenting of system's behavior, and/or the reading and commenting of parts of available documents; an instrumented version of some modules can be used in this process, in order to trace the internal structures traversed during the interaction with the product. As an alternative, these multiple knowledge acquisition lines can be activated in different moments with reference to the same set of test cases: for example, some test cases may be derived from the transcript of previous expert's interviews, document reading or code reading sessions; or the expert may reason off-line on batch input-output examples.

4. A conceptual schema for test case repositories.

The results of various types of testing and dynamic analysis strategies can be mapped onto the entity-relationship conceptual schema of fig.2; it may be used for the products of both error-oriented and knowledge elicitation oriented activities. The schema was defined in Docket for the specification of the test case repository; this term is used to refer to the repository as a whole, including the test data internal forms and the test case structured forms. The test cases internal form can be the existing test repository of the user's production environment; it can include raw test data files, scripts used to capture and playback user interactions [37], and test case descriptions in formatted documents linked to the test data. The test case structured forms [22] were defined as extensions and generalisations of the test tables proposed in [29]. The distinction between the internal forms and structured forms levels entails decisions on the degree of formalisation of test cases. The mass of all raw data sets(input and output batch files, scripts, transaction logs, etc.) belongs to the internal forms level; at the structured forms level it is possible to highlight the subsets of data and relationships that are essential to characterise a test case, and the kind of knowledge obtained from it. For example, the entities INPUT and OUTPUT represent program data which assume some values at run time; at the minimal degree of formalisation, the relationships between a test case and its INPUT and OUTPUT are represented only by references between a test case identifier in a test table and entire data sets in internal forms. More formalisation implies that the test case structured form includes references to a selection of specific program data, and possibly assertions on them. Whenever relevant, it is also possible to specify an attribute of INPUT and OUTPUT entities which distinguishes between variables containing stimuli or responses and those playing respectively the role of initial/final state variables in the test case. The other data attributes, which are obtained by static analysis of the software product, are represented in the code structured forms. Depending on the tool used, the results of code-level dynamic analysis may include both the information that certain internal program variables have been exercised, and information on the specific values assumed by them. If considered relevant, these internal variables can be included in the input and output sets of test cases, and their initial and final states represented by specific values or assertions. The schema of fig.2 highlights the role of a test case in interconnecting formal, informal and dynamic sources of knowledge. Most relationships depicted in the diagram have to be intended as optional. This may happen for two reasons: some information is not available at maintenance time and has to be reconstructed (eg. for an old test case only raw input data might be available, and not the testing strategy, the test case description, and so on); or the type of testing strategy involves only few types of information (eg. in structural testing the function to exercise is not defined). With respect to *dynamic sources*, one or more test cases may be necessary to answer a specific question or validate a specific assumption, and similarly one or more of them may be determinant to reach a given conclusion. A test case may provide the objective support to the interconnection of an hypotesis to a conclusion. With respect to *informal sources*, a test case may have an associated diagnosis report, containing the description of the results of the related error analysis activity; other documents may be the problem report, change request or user manual from which the test case was derived, and specific test case design documents. Empirical observations on the operational software system may be saved in form of assumptions and conclusions referred to cause-effect relationships which were revealed by test cases, where causes and effects are described in terms of formal source components. In terms of *formal sources*, a test case is characterised by: the set of software objects to be covered/checked; the set of its inputs; the set of its expected results; the set of its observed results. The goal of a black-box test case may be to exercise a component of a formal functional decomposition tree; the same test might have been run using a dynamic analyser, in order to reveal the internal structural components which are exercised when a given functional behavior is observed. Similarly, the goal of a white-box test case can be the exercising of a certain set of structural components, but it is possible to monitor even the corresponding run-time behavior. Taking into account the above possibilities, the set of observed results may be decomposed into two subsets: OBe, including those output variables and structural components which were explicitly mentioned in the expected results; and OBf, including any additional results which were in fact monitored, but not mentioned in the expected results. The OBe subset is supposed to be compared with the set of expected results, when this is not empty. The knowledge obtained from the analysis of the possible discrepancies may be contained in a diagnosis report. A structural component can be a source code component, or a component of a structured form representing a view of the software product at a certain level of abstraction. For representations derived by static code analysis, traceability links in the code structured forms allow to map in fact code-level execution traces to higher representation levels. Thus the test case structured form can refer to source code lines, slices, variable definitions, uses, definition-use pairs in flowgraphs, control branches, paths or decision-to-decision subpaths, modules, call sites, components of dependency graphs, structure charts, data flow diagrams, etc. In general, a test case can be referred to different formal representations of a software system at different abstraction levels. Even the objects to be checked/covered may be at different abstraction levels; the objects actually exercised may be either operational system components, or abstract schema components if such a schema is made directly exercisable, as in the case of the ER-TOOL. The possible type of both input and result depends on the abstraction level of the objects to exercise. For example, if the object to exercise is a given functionality of a subroutine, the input set will consist of its input variables, while the result set will include the output (or input-output) variables, and possibly paths exercised in the subroutine's internal structure. If, instead, the object to exercise is a transaction on the ER data schema of the system, input and output will refer to entities and relationships; the result will include paths exercised on the ER schema. The attributes of a test case include: test case identifier; textual description; and a set of *classification facets*. The main facets identified are *level, origin, strategy, and knowledge-classes*. The *level* of a test case is the software abstraction level to which its results are referred, while the *origin* records the type of activity producing it: alpha testing, beta testing, change analysis, regression testing, empirical observation made during operational life, knowledge-elicitation oriented testing. The *strategy* specifies whether, based either on explicit test design information or on the type of data collected, the testing process can be considered black-box, white box, grey-box, or other. Refinements of this attribute can specify particular strategies such as boundary value analysis, debugging by testing [7], all-du-paths coverage [35], and so on. The *knowledge-classes* attribute is a list of keywords that can be used to highlight the main types of knowledge obtained from the test case, ranging from the error detection point of view to the various kinds of comprehension targets indicated in section 2. Such an attribute should be used to provide the main justifications for recording the test case in the repository, or keeping it in time. The users of the repository should be allowed to signal the test cases and the type of knowledge they found more useful to solve specific problems, thus contributing to the perfection of test case classification in time and to the improvement of the repository value.

5. Conclusions

Testing-based knowledge elicitation was experimented in the Reverse Engineering of a COBOL administrative system, consisting of 39000 LOC in more than 100 source modules [27]. People having no previous knowledge of this system combined the results of some static and dynamic analysis tools with manual processing and the reading of user manuals. Entity Life Histories and Dialogue Graphs were produced from code and compared to those provided by human experts; also sentences found in user manuals were checked in terms of paths on these graphs, and linked to code parts selected from those traversed at run-time in the corresponding internal paths. Document fragments containing the definition of user-domain concepts and events were linked to data structures and instructions; further test cases were necessary to discriminate between different possible assumptions deducted from the manual, and their results were recorded as integrative assertions supported by objective behaviors. Testing was perceived as indispensable, even though the code was well-structured and contained meaninfgul names. According to the Docket schema, test case structured forms can be seen just as an intermediate stage in the construction of a Global System Model, or become also a permanent repository available for direct consultations through the Docket browser. This second form of exploitation entails many costs and benefits. The costs will include the formalisation of test cases, and the overheads of their persistent storage and possible update for any change in the programs. The benefits will include the reuse of objects which are not only necessary for error-oriented software revalidation, but also effective as software comprehension aids. The overheads can be minimised or justified by appropriate test case selection criteria; for this purpose, cost models which were developed for selective regression testing [36] should be enhanced to take into account the software comprehension point of view.

References.

[1] R.G.Reynolds et al. "Extracting procedural knowledge from software systems using inductive learning in the PM system", Proc.of 4th Int.Conf.on Softw.Eng. and Knowledge Eng.SEKE92, pp.131-139, IEEE CS Press

[2] G .Canfora, L. Sansone, G. Visaggio "Data Flow Diagrams: Reverse Engineering Production and Animation", Proc. of CSM92 IEEE Conf. on Soft.Maint., Orlando, FL Nov 9-12, 1992, pp.366-375

[3] P.Benedusi, V.Benvenuto, M.Caporaso "Maintenance and Prototyping at the Entity-Relationship level: a Knowledge-based support" Proc. of CSM90, San Diego CA Nov.26-29, 1990 p.161

[4] N.Wilde, J.Gomez, T.Gust, D.Strasburg "Locating User Functionality in Old Code", Proc. of CSM92

[5] H.Sneed "Reverse Engineering Programs via Dynamic Analysis" EP.5111 Docket Esprit Project, IP/A4 App.C,CR.083/REP/A.3/2.0, Dec. 1992, SES, Munich

[6] J.Ambras, V.O'Day "Microscope: a knowledge-based programming environment", IEEE Software, May 1988, p.50

[7] G.Myers "The Art of Software Testing" J.Wiley &Sons 1979

[8] A.Aho, R.Sethi, J.Ullman "Compilers: Principles, Techniques, and Tools", Addison-Wesley 1986

[9] D. Hedley, M.Hennell "The causes and effects of infeasible paths in computer programs" Proc. of 8th Int.Conf.on Software Engineering, London, Aug.1985

[10] R.Gopal "Dynamic Slicing Based on Dependence Relations" Proc. of CSM91, Sorrento, Italy, Oct.1991

[11] J.Ferrante, K.Ottenstein, D.Warren "The program Dependence Graph and its use in Optimization" , ACM Trans. on Programing Lang. and Systems, vol.9 no.3, pp.319-349, Jul.1987

[12] D.H.Hutchens, V.R.Basili "System structure Analysis: clustering with data bindings" IEEE Trans.Soft.Engr., SE-11(8), Aug.85

[13] A.Cimitile "Re-Use Re-Engineering: the RE^2 project" Proc. of Workshop on Reverse Engineering, Portici Italy, Dec.1991

[14] P.Benedusi, L.Casalini, V.Benvenuto "Additional Knowledge requirements on Operational Systems", EP.5111 Docket, IP/A4 CR.044/REP/A.2/2.0,1992

[15] "The Docket Method Manual", EP.5111 Docket UMIST, Manchester UK, Dec.1992

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

[17] P.Hausler et al. "Using Function Abstraction to Understand Program Behavior", IEEE Software Jan.1990, p.55

[18] A. Engberts, W.Kozaczynski, J.Ning "Concept Recognition-Based Program Transformation" Proc. of CSM91, Sorrento Italy, Oct.1991, p.73

[19] D.J. Newman "A Test Harness for Maintaining Unfamiliar Software", Proc. of CSM88, Phoenix, Arizona Oct. 24-27, 1988 pp. 409-416

[20] B. Watchel, Series "Software Analysis Notes", Software Maintenance News, July 87-Dec 88

[21] "The Docket Architecture Manual", EP.5111 Docket, UMIST, Manchester UK, Dec.1992

[22] "The Docket Formal Sources Structured Forms Manual" EP.5111 Docket,CR.084/MAN/A.3/2.0, Dec. 1992

[23] "Source Code Analysers for Structured Forms", EP.5111 Docket, IP/A4, CR.083/REP/A.3/2.0,Dec. 1992

[24] H. M. Sneed, G. Jandrasics "Inverse Transformations on Software from Code to Specification" Proc. of CSM88 , Phoenix, AZ Oct. 24-27 1988, pp.102-109

[25] H.Sneed, SES, "SOFT-RESPEC Reverse Engineering System", Docket Internal Report, 30 Sept 91, pp.10

[26] P.Benedusi, A.Cimitile, U.De Carlini "Reverse Engineering Processes, Design Document Production, and Structure Charts" Journal of Systems and Software, Vol.16 n.4, 1992, pp..225-245

[27] Paolo Benedusi, Vincenzo Benvenuto, L.Tomacelli "Experimentation in Operational System Knowledge Acquisition" EP.5111 Docket, IP/A5, CR.085/REP/A.4/2.0, Dec. 1992

[28] H.K.N. Leung, L. White "Insights into Regression Testing", Proc. of Conf. on Software Maintenance, Miami, Florida, Oct. 16-19 1989

[29] P. Benedusi, A. Cimitile, U. De Carlini "Post-Maintenance Testing based on Path Change Analysis", Proc. of CSM88, pp.352-361

[30] D.Binkley "Using Semantic Differencing to Reduce the Cost of Regression Testing" Proc. of CSM92 pp.41-50

[31] R.Gupta, M.J.Harrold, M.L.Soffa "An Approach to Regression Testing using Slicing" Proc. of CSM92 , pp.299-308

[32] PC-TEST Static and Dynamic Analyser User Manual, ver.3.10 1989 4D SOFT Bt. Budapest

[33] J. F. Sowa "Conceptual Structures: Information Processing in Mind and Machine" Addison-Wesley 1984

[34] R.E.M.Champion "The Development of a Unified System Model from Multiple Knowledge Sources" Proc. of Workshop on Reverse Enginering, Portici Italy, Dec.1991

[35] S. Rapps, E.J. Weyuker "Selecting Software Test Data Using Data Flow Information", IEEE Trans. on Soft. Eng. Vol. 11 n° 4, April 1985

[36] H.K.N. Leung, L. White "A Cost Model to Compare Regression Test Strategies", Proc. of CSM91, pp.201

[37] A.Serra, M.Vercelli "A Remote Terminal Emulator for Software Quality Control", Proc. of AQUIS'91 Conf. Pisa, Italy, Apr. 22-24, 1991

Session I:
Tools for Program Comprehension

Chair: Malcom Munro

Understanding Concurrent Programs using Program Transformations

E. J. Younger
Centre for Software Maintenance Ltd
Unit 1P, Mountjoy Research Centre
Durham, DH1 3SW

M. P. Ward
Computer Science Department
Science Labs, South Rd
Durham DH1 3LE

Abstract

Reverse engineering of concurrent real-time programs with timing constraints is a particularly challenging research area, because the functional behaviour of a program, and the non-functional timing requirements, are implicit and can be very difficult to discover. In this paper we present a significant advance in this area, which is achieved by modelling real-time concurrent programs in the wide spectrum language WSL. We show how a sequential program with interrupts can be modelled in WSL, and the method is then extended to model more general concurrent programs. We show how a program modelled in this way may subsequently be "inverse engineered" by the use of formal program transformations, to discover a specification for the program. (We use the term "inverse engineering" to mean "reverse engineering achieved by formal program transformations").

1 Introduction

This paper describes extensions to the inverse engineering techniques developed at the University of Durham, to enable these to be applied to programs with interrupts and concurrency. An example of the use of the methods to derive a specification for a simple concurrent system is given.

The paper is organised as follows. In sections 2 and 3 we give a brief introduction to the WSL language and transformation theory. Then in Section 4 we show how to model a real-time interrupt-driven program in the WSL language. In Section 5 we show how the principles may be extended to model more general concurrent programs. Finally, we use this information to derive a specification of the example program in Section 6.

2 The Language WSL

In this section we give a brief introduction to the language WSL [2,9,12] the "Wide Spectrum Language", used in Ward's program transformation work, which includes low-level programming constructs and high-level abstract specifications within a single language. For brevity we will only define the language constructs used in this paper.

A *program transformation* is an operation which modifies a program into a different form which has the same external behaviour (it is equivalent under a precisely defined denotational semantics). Since both programs and specifications are part of the same language, transformations can be used to demonstrate that a given program is a correct implementation of a given specification. In [10,13,14] program transformations are used to derive a variety of efficient algorithms from abstract specifications. In [15,16] program transformations are used in reverse engineering and program comprehension tasks, including the derivation of concise specifications from program source code.

2.1 Sequences

Sequences are denoted by angled brackets: $s = \langle a_1, a_2, \ldots, a_n \rangle$ is a sequence, the ith element a_i is denoted $s[i]$, $s[i\,..\,j]$ is the subsequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, where $s[i\,..\,j] = \langle\rangle$ (the empty sequence) if $i > j$. The length of sequence s is denoted $\ell(s)$, so $s[\ell(s)]$ is the last element of s. We use $s[i\,..]$ as an abbreviation for $s[i\,..\,\ell(s)]$. $\mathit{reverse}(s) = \langle a_n, a_{n-1}, \ldots, a_2, a_1 \rangle$, $\mathit{head}(s)$ is the same as $s[1]$ and $\mathit{tail}(s)$ is $s[2\,..]$.

The concatenation of sequences s_1 and s_2 is denoted $s_1 \mathbin{+\!\!+} s_2 = \langle s_1[1], \ldots, s_1[\ell(s_1)], s_2[1], \ldots, s_2[\ell(s_2)] \rangle$.

2.2 Specification Statement

The statement $\langle x_1, \ldots, x_n \rangle := \langle x'_1, \ldots, x'_n \rangle.\mathbf{Q}$ assigns new values $x'_1, \ldots, x'_n$ to the variables $x_1, \ldots, x_n$ such that the formula $\mathbf{Q}$ is true. If there are no values which satisfy $\mathbf{Q}$ then the statement aborts (does not terminate). For example, the assignment $\langle x \rangle := \langle x' \rangle.(x = 2.x')$ halves x if it is even and aborts if x is odd. If the sequence contains one variable then the sequence brackets may be omitted, for example: $x := x'.(x = 2.x')$. The assignment $x := x'.(x' = t)$ where x' does not occur in t is abbreviated to $x := t$.

2.3 Unbounded Loops

Statements of the form do S od, where S is a statement, are "infinite" or "unbounded" loops which can only be terminated by the execution of a statement of the form exit(n) (where n is an integer, *not* a variable or expression) which causes the program to exit the n enclosing loops. To simplify the language we disallow exits which leave a block or a loop other than an unbounded loop. This type of structure is described in [6,8].

3 Program Refinement and Transformation

The WSL language includes both specification constructs, such as the general assignment, and programming constructs. One aim of our program transformation work is to develop programs by refining a specification, expressed in first order logic and set theory, into an efficient algorithm. This is similar to the "refinement calculus" approach of Morgan et al [4,7]; however, our wide spectrum language has been extended to include general action systems and loops with multiple exits. These extensions are essential for our second, and equally important aim, which is to use program transformations for reverse engineering from programs to specifications.

Refinement is defined in terms of the denotational semantics of the language: the semantics of a program S is a function which maps from an initial state to a final set of states. The set of final states represents all the possible output states of the program for the given input state. Using a set of states enables us to model nondeterministic programs and partially defined (or incomplete) specifications. For programs $\mathbf{S}_1$ and $\mathbf{S}_2$ we say $\mathbf{S}_1$ is refined by $\mathbf{S}_2$ (or $\mathbf{S}_2$ is a refinement of $\mathbf{S}_1$), and write $\mathbf{S}_1 \leq \mathbf{S}_2$, if $\mathbf{S}_2$ is more defined and more deterministic than $\mathbf{S}_1$. If $\mathbf{S}_1 \leq \mathbf{S}_2$ and $\mathbf{S}_2 \leq \mathbf{S}_1$ then we say $\mathbf{S}_1$ is equivalent to $\mathbf{S}_2$ and write $\mathbf{S}_1 \approx \mathbf{S}_2$. Equivalence is thus defined in terms of the external "black box" behaviour of the program. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [9] and [12] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations. We use the term *abstraction* to denote the opposite of refinement: for example the "most abstract" program is the non-terminating program **abort**, since any program is a refinement of **abort**.

A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program. See [9] and [12] for a description of the semantics of WSL and the methods used for proving the correctness of refinements and transformations.

4 Modelling Interrupt-Driven Programs in WSL

WSL has no notations for parallel execution or interrupts. We chose not to add such notations to the language, since this would complicate the semantics enormously and render virtually all our transformations invalid. Consider, for example, the simple transformation:

$$x := 1;\ \underline{\text{if}}\ x = 1\ \underline{\text{then}}\ y := 0\ \underline{\text{fi}}\ \approx\ x := 1;\ y := 0$$

which is trivial to prove correct in WSL. However, this transformation is not universally valid if interrupts or parallel execution are possible, since an interrupting program could change the value of x between the assignment and the test. Instead, our approach is to *model* the interrupts in WSL by inserting a procedure call at all the points where the program could be interrupted. This procedure tests if an interrupt did actually occur, and if so it executes the interrupt routine, otherwise it does nothing. Although this increases the program size somewhat, the resulting program is written in pure WSL and all our transformations can be applied to it.

One of our aims in transforming the resulting WSL program is to move the interrupt calls through the body of the program, and collect them together and merge them as far as possible. The body of the main program would then be essentially sequences of statements from the original program, separated by the processing of any interrupts which occurred during their execution.

In order to model time within WSL we add a variable *time* to the program which is incremented appropriately whenever an operation is carried out which takes some time. We can then reason about the response times of the program by observing the initial and final values of this variable. We can also model the times when interrupts occur by providing an input sequence consisting of *pairs* of values $\langle t, c\rangle$ where t the time at which the interrupt occurs and c is the character. Naturally we should insist that the sequence of t values be monotonically increasing. Such a sequence can model input from an external device, a concurrent process, or even a hardware register. We make this explicit in our model of the program: the array (or equivalently, sequence) *input* consists of pairs of times and characters to represent the inputs, and is sorted by times. The interrupt routine tests the *time* variable against the time value associated with the first element of the input sequence to see if that interrupt is now "due". If so, then it removes a pair from the head of the sequence and processes the result. The program is modelled as follows:

```
S₁;        ⇝   S₁;
S₂;            interrupt(time); time := time + 1;
etc ...        S₂;
               interrupt(time); time := time + 1;
               etc ...
```

If we assume a discrete model of time, i.e. that the value of *time* is an integer, and we assume that time is incremented by one between each potential interrupt, then the test for validity of a call to the interrupt routine is simply:

$$interrupt(time_0) \leadsto \underline{\text{if}}\ (time_0 = input[1][1])\ \underline{\text{then}}\ process\_interrupt\ \underline{\text{fi}}$$

where *process_interrupt* corresponds to the original interrupt service routine. Note that *input*[1] is the first element of the input sequence: this element is a pair of values (a time and a character), so *input*[1][1] is the first element of the pair, i.e. the time of the first interrupt.

However, a better model, which does not require a discrete model of time, and which allows different "atomic" (i.e. non-interruptable) operations to take different amounts of time, is the following:

$$interrupt(time_0) \leadsto \underline{\text{while}}\ (time_0 \geqslant input[1][1])\ \underline{\text{do}}\ process\_interrupt\ \underline{\text{od}}$$

This revised model of the interrupt routine allows more than one interrupt to occur between atomic operations, and has the advantage that a call to *interrupt* can be merged with a second call which immediately follows it:

$$interrupt(t_1);\ interrupt(t_2) \approx interrupt(t_2)$$

provided $t_2 \geqslant t_1$. This follows from the transformation:

$$\underline{\text{while}}\ B_1\ \underline{\text{do}}\ S\ \underline{\text{od}};\ \underline{\text{while}}\ B_2\ \underline{\text{do}}\ S\ \underline{\text{od}} \approx \underline{\text{while}}\ B_2\ \underline{\text{do}}\ S\ \underline{\text{od}}$$

provided $B_1 \Rightarrow B_2$. This transformation is proved in [9]. If $t_2 \geqslant t_1$ then we have $(t_1 \geqslant input[1][1]) \Rightarrow (t_2 \geqslant input[1][1])$, and we can merge the two while loops and hence the two procedures.

Thus, once we have moved a set of interrupt procedure calls to the same place, we can merge them into one statement equivalent to "process all outstanding interrupts", which is much closer to a specification level statement than is a series of calls to the same procedure.

Note that the addition of interrupt calls is defining the "interruptable points" in the program, or equivalently, the "atomic operations". The increments to *time* define the processing time for each atomic operation. For real programming languages, e.g. Coral, the atomic operations may well be machine code instructions, rather than high-level language statements, and it is at the machine code level that the model needs to be constructed, for it to accurately reflect the real program. This will inevitably lead to a large and complex WSL program; however automatic restructuring and simplifying transformations can eliminate much of the complexity before the maintainer even has to look at the program.

5 Concurrency

Interrupts may be regarded as a special type of concurrent processing on a single processor. When an interrupt occurs, the "main" program is suspended and the interrupt routine is executed in its entirety, possible changing the state of the main program in the process. Execution of the main program then resumes. It is the fact that the interrupt routine is executed in its entirity that makes interrupts a special case from the point of view of modelling in WSL; we are able to insert a copy of the interrupt routine wherever an interrupt occurs, and hence the effect of the interrupt is deterministic.

The analogy with a single-processor multitasking system is obvious: here the running program executes until it is suspended by the operating system. Other tasks are then (partially) executed, and may change the state of the original program; eventually execution of this program is resumed. From the perspective of the original task, this looks like a call to a procedure which executes sequences of instructions from the operating system and the other tasks in the system, and subsequently returns. The analogy also applies to more general forms of concurrency, including fully-parallel multiprocessor systems. In principle, the state of any program or task in such a system may be changed, between one atomic operation and the next, by other concurrently executing tasks. Again this could be modelled by procedure calls between each pair of atomic operations, which perform the appropriate processing and change the state accordingly.

5.1 Rely and Guarantee conditions

Rely and guarantee conditions were introduced to augment the pre- and postconditions of VDM when developing parallel systems [5]. They provide a means to specify the interaction between a program and its execution environment (concurrent tasks). A guarantee condition is a condition on the state shared by the program and its environment, which the program will at all times preserve. Similarly, a rely condition is a condition on the state of the program which any interference from the environment will preserve. As an example, consider the abstraction:

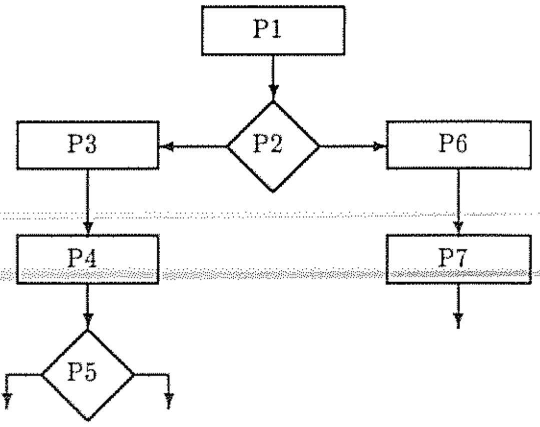
$$\begin{array}{lcl} x := 1; & \geq & x := x'.(x' \geqslant 1); \\ \underline{\text{if}}\ x \geqslant 1\ \underline{\text{then}}\ y := 0\ \underline{\text{fi}} & & y := 0 \end{array}$$

In pure WSL this is trivially correct, but if we allowed interference from the environment to change the value of x between the assignment and the test, the transformation is not necessarily valid. However, if we have a rely condition which specifies $x \geqslant \overline{x}$ then this transformation will be valid. (Here $\overline{x}$ and x represent respectively the initial and final values of x)

In the following section we use rely and guarantee conditions to model the effects of concurrency in pure WSL. Again this means that our program transformations are applicable since the resulting program is purely sequential.

5.2 Modelling Concurrency in WSL

Consider a sequential program T which can be represented by a flow chart. Any program can be restructured into this form. For example:



where each of the Pn represents an atomic instruction or an atomic test. Suppose we model Pn by Pn; $next := n{+}1$; $time := time{+}1$; , with the obvious extension for branch instructions. Now suppose **T** is a task in a concurrent system. Consider first a system with only two tasks, $\mathbf{T}_1$ and $\mathbf{T}_2$. We rewrite $\mathbf{T}_2$ as a procedure as follows;

```
proc T_2(steps) ≡
    while (steps > 0) do
            steps := steps − 1;
            if next = 1 → P1; next := 2
            □ next = 2 → if P2
                              then next := 3
                               else next := 6 fi
            ...
            □ next = 0 → skip fi od;
```

This procedure executes a sequence of *steps* instructions from $\mathbf{T}_2$, beginning from the last instruction executed in the last invocation of the procedure. If *steps*=0 then the procedure call returns immediately.

If we then model $\mathbf{T}_1$ by

$$\begin{array}{lll} \mathbf{T}_1 & \rightsquigarrow & steps := steps'.(steps' \in \mathbb{N}^0); \\ & & \mathbf{T}_2'(steps); \\ & & \text{S1};\ time := time + 1; \\ & & steps := steps'.(steps' \in \mathbb{N}^0); \\ & & \mathbf{T}_2'(steps); \\ & & \text{S2};\ time := time + 1; \\ & & \dots \end{array}$$

then we have modelled the semantics of the original system. By interposing calls to $\mathbf{T}_2'$ between the atomic instructions of $\mathbf{T}_1$ we have modelled the execution of $\mathbf{T}_2$ and its interference with $\mathbf{T}_1$.

This is a complete model of the system, even though it has the same surface structure as $\mathbf{T}_1$. It is non-deterministic due to the presence of the non-deterministic assignment statements in $\mathbf{T}_1$, which cause values to be assigned to the variable *steps*. This reflects the fact that we cannot know *a priori* how many instructions from $\mathbf{T}_2$ will be executed between Pn and P$n+1$ in $\mathbf{T}_1$. In order to proceed, we want to replace:

$$\textit{steps} := \textit{steps}'.(\textit{steps}' \in \mathbb{N}^0);\ \mathbf{T}'_2(\textit{steps});$$

with an abstraction, which specifies $\mathbf{T}'_2$ for any value of *steps* and any initial value of *next*. This will be the strongest condition preserved by the execution of *any* sequence of instructions from $\mathbf{T}_2$, and corresponds to a guarantee condition for $\mathbf{T}_2$. This specification is recoverable from $\mathbf{T}_2$ in isolation.

We can repeat this process, interchanging the roles of $\mathbf{T}_1$ and $\mathbf{T}_2$. We then will have two models, one based on $\mathbf{T}_1$ with interference from $\mathbf{T}_2$ inserted, the other based on $\mathbf{T}_2$ with interference from $\mathbf{T}_1$. However these are guaranteed to be equivalent, since they both capture the logic of the entire system.

Now suppose we have more than two tasks $\mathbf{T}_1 \ldots \mathbf{T}_n$. Let $\mathbf{T}_2 \ldots \mathbf{T}_n$ be rewritten as procedures as above. Then we can write

$$\begin{array}{ll} \mathbf{T}_1 \rightsquigarrow & \textit{steps} := \textit{steps}'.(\textit{steps}' \in \mathbb{N}^0); \\ & V(\textit{steps}); \\ & \mathrm{S1};\ \textit{time} := \textit{time} + 1; \\ & \textit{steps} := \textit{steps}'.(\textit{steps}' \in \mathbb{N}^0); \\ & V(\textit{steps}); \\ & \ldots \end{array}$$

where

```
$\underline{\text{proc}}$ $V(steps)$ $\equiv$
    $\underline{\text{for}}$ $k := 1$ $\underline{\text{to}}$ $steps$ $\underline{\text{step}}$ 1 $\underline{\text{do}}$
        $\underline{\text{if}}$ $\mathbf{true} \rightarrow \mathbf{T}'_2(1)$
        $\Box$ $\mathbf{true} \rightarrow \mathbf{T}'_3(1)$
        ...
        $\Box$ $\mathbf{true} \rightarrow \mathbf{T}'_n(1)$ $\underline{\text{fi}}$ $\underline{\text{od}}$ $\underline{\text{end}}$
```

and each $\mathbf{T}'_j$ is derived from the corresponding $\mathbf{T}_j$ as before.

Procedure V causes *steps* instructions to be executed; these are chosen non-deterministically from the remaining tasks $\mathbf{T}_2, \ldots, \mathbf{T}_n$ by the $\underline{\text{if}}$ statement, which calls one of the corresponding procedures with parameter one, resulting in the execution of one instruction from the task.

The above is a complete specification for the system when taken together with the definitions of the procedures $\mathbf{T}'_j$. However it is also non-deterministic due to the assignment statements; as before we need to replace the assignments and procedure calls by abstractions.

Consider the procedure V. Here the loop body implements a non-deterministic choice from the set of procedures $\mathbf{T}'_j$, where $j = 2 \ldots n$. One of the procedures is selected non-deterministically, and a single instruction from the corresponding task is executed. A total of *steps* atomic instructions from the set of tasks is executed; individual instructions or sequences of instructions may be executed from any of the tasks.

We can abstract $\mathbf{T}'_j(1)$ to a non-deterministic choice over all sequences of instructions in $\mathbf{T}_j$ — in so doing we abstract away the fact that the instructions are executed in a specific order. This is equivalent to removing the guards from the $\underline{\text{if}}$ statement in the procedures $\mathbf{T}'_j$, which we can always do as an abstraction step. We can therefore write

$$\mathbf{T}'_j(1) \ \geq\ \prod_i \mathrm{P}^i_j$$

where P^i_j are the instructions in the task $\mathbf{T}_j$, and $\prod_i$ indicates a non-deterministic choice. This is the strongest specification which is an abstraction of every instruction in $\mathbf{T}_j$. In fact this specification may be very large for a task which is even moderately complex, making it difficult to use in practice. To reason about its effect on $\mathbf{T}_1$ however we are only concerned with the elements of the specification which affect the state of $\mathbf{T}_1$; we may therefore if necessary abstract away other details of the $\mathbf{T}_j$'s and so simplify the specifications of the $\mathbf{T}'_j$s.

Having found a specification for each of the $\mathbf{T}'_j$, we can use these to find the specification for the loop body of the procedure V. This is a non-deterministic choice from the set of $\mathbf{T}'_j(1)$'s, whose specifications are given above. The procedure V executes the loop body *steps* times, where the value of *steps* is not determinable, representing as it does the number of instructions executed between two sequential instructions in $\mathbf{T}_1$. The strongest specification for V which we can find is therefore the strongest abstraction which holds for any value of *steps*, i.e. any number of concatenations of the loop body. This will in general be an abstraction of the specification for the loop body.

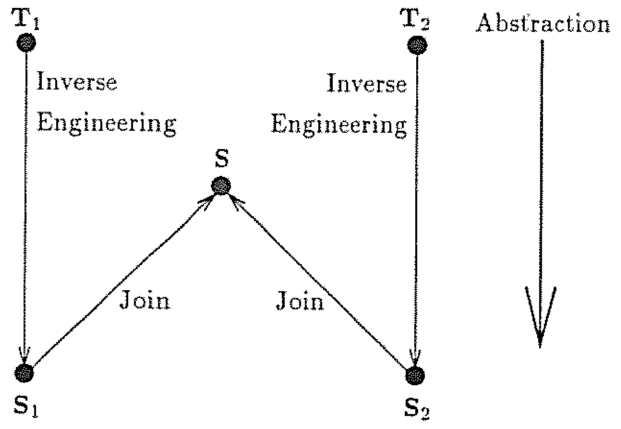
To summarise, we have the following abstraction/refinement relations:

$$\begin{gathered} \mathbf{T}'_j(1) \ \geq\ \prod_i \mathrm{P}^i_j \\ \mathrm{L} \ \approx\ \prod_{j>1} \mathbf{T}'_j(1) \ \geq\ \prod_{j>1} \Big(\prod_i \mathrm{P}^i_j\Big) \ \approx\ \mathrm{S}_L \\ \mathrm{S}_L \ \geq\ \mathrm{S}_V \end{gathered}$$

where L is the body of the loop, S_L is a specification for the loop body, and S_V is a specification for the procedure V. The crucial factor in this method is that this specification is derived by analysis of the individual tasks in isolation, without the need to take account of the interaction between them. It can therefore be found using methods developed for purely sequential systems.

S_V therefore specifies the effects of the remaining tasks on the state of T_1. It is in effect a rely condition for T_1. By substituting this specification for the calls to procedure V in our model of T_1, we can inverse engineer T_1 to recover its specification, S_1, which includes the effects of the other tasks upon it. Since we may have abstracted away those parts of the specification which do not affect the state of T_1, in order to simplify the specifications of the $T'_j(1)$'s, we no longer have a complete specification for the entire system. Repeating this exercise for the remaining tasks allows us to recover a set of specifications S_j for the tasks, each of which include the effects of the other tasks upon their state.

The specification S for the complete system must incorporate all the properties of the set of S_j's. This is expressed in WSL by the join operation. The join of a set of specifications is the most abstract specification which refines each member of the set. Thus the specification S is more refined than the specifications S_j, unless the specifications S_j are equivalent; this arises since, in combining the specifications for the tasks, we are restoring information originally abstracted away in the derivation of the individual S_j's: S_i will include information abstracted away in the derivation of S_k (for any i and k).



Although S will be more refined than the set of S_j's, it may still be too abstract to be useful if too much abstraction is performed in deriving the S_j's. This will be a problem particularly if the objective is to validate a system against an existing specification, or to re-implement an existing system. If too much information is lost in deriving the S_j's then the specification S may be more abstract than the specification against which we wish to validate, or more abstract than the actual system requirements specification.

5.3 Summary of method

In order to model a concurrent system in WSL we proceed as follows:

For each task, we derive a specification for the conditions on its state which are guaranteed to be preserved by the other tasks in the system. To achieve this, we consider each of the remaining tasks in isolation, and derive for each of these a specification of the conditions on the state of the first task which are guaranteed to be preserved by the execution of any instruction from the second task. The resulting specifications, derived from all the remaining tasks, are then combined by a non-deterministic choice to give us a specification of the conditions on the state of the first task, guaranteed to be preserved by the execution of any instruction from any other task in the system. The full specification for the effect of the remaining tasks on the first is then given by any number of concatenations of this specification.

We substitute this specification between each atomic operation in the task: this gives us an abstraction of the task itself including the effect of interference from the rest of the system. We may then inverse engineer this model of the task using transformations, to find a specification for the task incorporating the effects of interference from the other tasks.

We can repeat this process for each of the tasks in the system. Having derived a specification for each of them, we can derive the specification for the complete system by combining the individual specifications using the WSL join operation.

6 Example of method

In this section we give an example of the use of the method to inverse engineer a simple concurrent system. This consists of only two tasks sharing a single processor under the control of a scheduler, though we ignore the details of the scheduler. One task receives characters from an input stream, and writes these into a buffer; the second process takes characters from this buffer and writes them to the standard output.

```
T_1 = do if empty(input) then exit(1) fi;
         while (time < input[1][1]) do
                 suspend1 od;
         buf := buf ++ ⟨input[1][2]⟩;
         input := tail(input) od

T_2 = do if empty(input) ∧ empty(buf)
            then exit(1) fi;
         while (empty(buf)) do
                 suspend2 od;
         std_out := std_out ++ ⟨buf[1]⟩;
         buf := tail(buf) od
```

The input consists of a sequence of pairs: a character and its arrival time. If there is no character waiting to be read, i.e. the arrival time of the next character in the input has not yet been reached, $\mathbf{T}_1$ suspends itself by means of the system call *suspend1*, which results in $\mathbf{T}_2$ being reactivated. Similarly, if the buffer is empty, $\mathbf{T}_2$ suspends itself by the system call *suspend2*. If there is no more input to be received, then $\mathbf{T}_1$ terminates; similarly if there is no further input and no characters left in the buffer $\mathbf{T}_2$ terminates.

Execution of the two tasks alternates on a timeslice basis: at regular intervals, the executing task is halted and the second task is restarted. We can model this using a procedure: we define

```
proc timeslice1 ≡
      if (time − last ⩾ P_t)
        then last := time; suspend1 fi.
```

where P_t is the timeslice period, and an equivalent procedure *timeslice2* for $\mathbf{T}_2$. This procedure tests the time since the last context switch, and if it is greater than or equal to the timeslice period suspends the active task and reactivates the halted one via the system call *suspend1*. To incorporate the effect of interference from the other task we insert this procedure call between each statement in the task, together with an assignment which increments *time*. (For the purposes of this example we treat each WSL statement as an atomic instruction). The result, following the necessary restructuring of the while loops to accommodate these additions, is

```
T̂_1 ≈ do if empty(input) then exit(1) fi;
         do if (time ⩾ input[1][1])
                then timeslice1; time := time + 1;
                       exit(1)
                  else timeslice1; time := time + 1 fi;
              suspend1 od;
         buf := buf ++ ⟨input[1][2]⟩;
         timeslice1; time := time + 1;
         input := tail(input);
         timeslice1; time := time + 1 od

T̂_2 ≈ do if empty(input) ∧ empty(buf)
            then exit(1) fi;
         do if ¬empty(buf)
                then timeslice2; time := time + 1;
                       exit(1)
                  else timeslice2; time := time + 1 fi;
              suspend2 od;
         std_out := std_out ++ ⟨buf[1]⟩;
         timeslice2; time := time + 1;
         buf := tail(buf);
         timeslice2; time := time + 1 od
```

In order to proceed, we now need definitions for the procedures *suspend1* and *suspend2*. These specify the effect of executing the other task for one timeslice period, and must refine the conditions which are preserved by execution of any sequence of statements from the tasks, i.e. any number of whole or partial executions of the respective loop bodies. We can recover these conditions, and define abstractions of *suspend1* and *suspend2* which preserve the conditions but are otherwise unrestricted. We denote these by *G1* and *G2* respectively. We then know that $G1 \leq suspend1$ and $G2 \leq suspend2$.

During its timeslice, a task executes without interference from the other task. We therefore recover *G1* and *G2* from $\mathbf{T}_1$ and $\mathbf{T}_2$. As these are small in this example, we can do this by inspection. Consider first $\mathbf{T}_2$. We are interested only in how it affects the variables in $\mathbf{T}_1$: in fact the only variables which are changed by $\mathbf{T}_2$ are *std_out* and *buf*, (and also *time*, though this is not shown in the definition of $\mathbf{T}_2$ given above). *std_out* is not accessed or updated in $\mathbf{T}_1$, so we can disregard it. Since at least one statement in $\mathbf{T}_2$ will be executed, *time* will be incremented. Also, we can see that zero or more characters will be removed from the head of *buf*, up to a maximum of $\ell(input)$, in which case the buffer is emptied and $\mathbf{T}_2$ is then suspended. Using these facts we can write:

$$\begin{aligned} G1 = \langle time, buf \rangle := \langle time', buf' \rangle.\ & (time' > time \\ & \wedge \exists j, 0 \leqslant j \leqslant \ell(buf).\ buf' = buf[j\,..]) \end{aligned}$$

Clearly, $G1;\ G1\ \approx\ G1$. As $G1 \leq suspend1$ we can also write

$$\begin{aligned} & timeslice1 \\ & \geq \underline{\text{if}}\ (time - last \geqslant P_t) \\ & \qquad \underline{\text{then}}\ last := time;\ G1\ \underline{\text{fi}} \\ & \geq \langle time, buf \rangle := \langle time', buf' \rangle. \\ & \qquad (time' > time \\ & \qquad\quad \wedge \exists j, 0 \leqslant j \leqslant \ell(buf).\ buf' = buf[j\,..]) \\ & \geq G1 \end{aligned}$$

Having found $G1$, we are able to inverse engineer $\hat{\mathbf{T}}_1$:

```
$\hat{\mathbf{T}}_1 \geq$
$\underline{\text{do}}$ $\underline{\text{if}}$ $empty(input)$ $\underline{\text{then}}$ $\underline{\text{exit}}(1)$ $\underline{\text{fi}}$;
    $\underline{\text{do}}$ $\underline{\text{if}}$ $(time \geqslant input[1][1])$
          $\underline{\text{then}}$ $G1;\ time := time + 1;$
                $\underline{\text{exit}}(1)$
          $\underline{\text{else}}$ $G1;\ time := time + 1$ $\underline{\text{fi}}$;
       $G1$ $\underline{\text{od}}$;
    $buf := buf +\!\!+ \langle input[1][2] \rangle;$
    $G1;\ time := time + 1;$
    $input := tail(input);$
    $G1;\ time := time + 1$ $\underline{\text{od}}$;
```

Replacing $G1$ by its definition, transforming the loop to a <u>while</u> and simplifying gives:

```
$\underline{\text{while}}$ $\neg empty(input)$ $\underline{\text{do}}$
    $\langle time, buf \rangle := \langle time', buf' \rangle.$
      $(time' > input[1][1]$
        $\wedge\ \exists j, 0 \leqslant j \leqslant \ell(buf).$
            $buf' = buf[j\,..] +\!\!+ input[1][2]);$
    $input := tail(input)$ $\underline{\text{od}}$;
```

This loop can be replaced by a single assignment:

$$\begin{aligned} & \hat{\mathbf{T}}_1 \geq \\ & \langle time, buf \rangle := \langle time', buf' \rangle. \\ & \quad (time' > input[\ell(input)][1] \\ & \qquad \wedge \exists j, 0 \leqslant j \leqslant \ell(buf) + \ell(input). \\ & \qquad\qquad buf' = (buf +\!\!+ \pi_2 * input)[j\,..]); \\ & input := \langle\rangle \\ & = \mathbf{S}_1 \end{aligned}$$

Here $\pi_2 * input$ is the sequence of "second elements" from the sequence of pairs $input$. $\mathbf{S}_1$ is a specification for $\mathbf{T}_1$ with the interference from $\mathbf{T}_2$ included. All the input characters are read and so the input is emptied; the time at which the task terminates is some time after the last character is received; on termination, the buffer consists of the initial contents of the buffer with all the input characters appended to the end, and an unspecified number of characters (less than or equal to the number of characters in the buffer) removed from the head. This quantity cannot be determined solely from $\hat{\mathbf{T}}_1$ as it depends on the number of iterations of $\mathbf{T}_2$ which are executed before $\mathbf{T}_1$ terminates, and this is unknown to $\mathbf{T}_1$.

We now inverse engineer $\hat{\mathbf{T}}_2$ by the same method to find a specification $\mathbf{S}_2$. Inspection of $\mathbf{T}_1$ provides us with a suitable abstraction. Define:

$$\begin{aligned} G2 = \langle buf, time \rangle := \langle buf', time' \rangle.\ & (time' > time \\ & \wedge \exists j, 0 \leqslant j \leqslant \ell(input). \\ & \qquad buf' = buf +\!\!+ \textstyle\sum_{i=0}^{j} input[i][2]) \end{aligned}$$

We have: $G2;\ G2\ \approx\ G2$ and $timeslice2\ \geq\ G2$.

This is an abstraction of $suspend2$. Substituting this in $\hat{\mathbf{T}}_2$, we can inverse engineer to find:

$$\begin{aligned} & \hat{\mathbf{T}}_2 \geq \\ & \langle std\_out, buf, time \rangle := \langle std\_out', buf', time' \rangle. \\ & \quad (time' > time \\ & \qquad \wedge std\_out' +\!\!+ buf' +\!\!+ \pi_2 * input' \\ & \qquad\qquad = std\_out +\!\!+ buf +\!\!+ \pi_2 * input \\ & \qquad \wedge buf' = \langle\rangle \wedge input' = \langle\rangle) \\ & \approx \\ & \langle std\_out, buf, time \rangle := \langle std\_out', buf', time' \rangle. \\ & \quad (std\_out' := std\_out +\!\!+ buf +\!\!+ \pi_2 * input \\ & \qquad \wedge time' > time) \\ & \approx \mathbf{S}_2 \end{aligned}$$

The specification for the complete system is then given by:

$$\mathbf{S} =_{\text{DF}} \underline{\text{join}}\ \mathbf{S}_1 \sqcup \mathbf{S}_2\ \underline{\text{nioj}}$$

For two specifications, the <u>join</u> operation reduces to and-ing the conditions in the specification statements. Therefore:

$$\begin{aligned} & \mathbf{S} \approx \\ & \langle std\_out, buf, time \rangle := \langle std\_out', buf', time' \rangle. \\ & \quad (time' > input[\ell(input)][1] \\ & \qquad \wedge std\_out' = std\_out +\!\!+ buf +\!\!+ \pi_2 * input \\ & \qquad \wedge buf' = \langle\rangle \wedge input' = \langle\rangle) \end{aligned}$$

This specification tells us that, for any initial values of buf, $input$ and std_out, the system terminates in a state in which buf and $input$ are empty, and std_out consists of its initial value with the initial contents of buf and the sequence of characters from $input$ appended, in the order in which they occurred in the input. Additionally, the time at which the system terminates is greater than the time of arrival of the last character. In this example, the specification S_2 is almost a complete specification for the system, reflecting the fact that very little high level information was abstracted away in deriving a specification for the interference of T_1 with the state of T_2.

7 Conclusions

This study shows that by using an appropriate models we can represent interrupt-driven and concurrent programs within the (purely sequential) WSL language. With such models, the inverse engineering techniques of [12,15] can be applied to extract the specification of the original program. Program transformations are sufficiently powerful to cope with these, often complex, models. Although a fairly large number of transformations are required to deal with these models, results from case studies indicate that these are used in a systematic way: this suggests that much of the work can be automated by a tool such as the Maintainer's Assistant [2,17] and this is currently being investigated under a SMART II (Small Firms Merit Award for Research and Technology) project at the Centre for Software Maintenance Ltd., and as part of a three-year SERC project at the University of Durham.

Acknowledgements

The research described in this paper has been funded partly by a Department of Trade and Industry SMART award to the Centre for Software Maintenance Ltd., and partly by SERC (The Science and Engineering Research Council) project "A Proof Theory for Program Refinement and Equivalence: Extensions".

References

[1] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, 1980.

[2] T. Bull, "An Introduction to the WSL Program Transformer," *Conference on Software Maintenance 26th–29th November 1990, San Diego* (Nov., 1990).

[3] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[4] C. A. R. Hoare, I. J. Hayes, H. E. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey & B. A. Sufrin, "Laws of Programming," *Comm. ACM* 30 (Aug., 1987), 672–686.

[5] C. B. Jones, "Specification and design of (parallel) systems," in *Proc. IFIP 1983*, R. E. A. Mason, ed., North Holland, 1983, 321–332.

[6] D. E. Knuth, "Structured Programming with the GOTO Statement," *Comput. Surveys* 6 (1974), 261–301.

[7] C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[8] D. Taylor, "An Alternative to Current Looping Syntax," *SIGPLAN Notices* 19 (Dec., 1984), 48–53.

[9] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[10] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990.

[11] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," forthcoming, 1993.

[12] M. Ward, "Specifications and Programs in a Wide Spectrum Language," Submitted to J. Assoc. Comput. Mach., Apr., 1991.

[13] M. Ward, "A Recursion Removal Theorem," BCS Refinement Workshop, 8th–11th January, Jan., 1992.

[14] M. Ward, "Derivation of Data Intensive Algorithms by Formal Transformation," Submitted to IEEE Trans. Software Eng., May, 1992.

[15] M. Ward, "Abstracting a Specification from Code," To appear in Journal of Software Maintenance: Research and Practice, Sept., 1992.

[16] M. Ward & K. H. Bennett, "A Practical Program Transformation System For Reverse Engineering," *Working Conference on Reverse Engineering, May 21–23, 1993*, Baltimore MA (May, 1993).

[17] M. Ward, F. W. Calliss & M. Munro, "The Maintainer's Assistant," *Conference on Software Maintenance 16th–19th October 1989, Miami Florida* (Oct., 1989).

Charon: a Tool for Code Redocumentation and Re-Engineering

Oreste Signore - Mario Loffredo

CNUCE - Institute of CNR - via S. Maria, 36 - 56126 Pisa
Phone: +39 (50) 593201 - FAX: +39 (50) 904052 E.mail: oreste@vm.cnuce.cnr.it

Abstract

The maintenance of applications constitutes the most relevant issue of the overall life cycle activities. CASE tools claim to be effective in producing efficient and error free software, but usually the maintainer doesn't want to produce new system applications, but just to modify the existing ones. Re-engineering appears to be a suitable way of getting the advantages of the automated CASE tools, without facing the costs involved in a complete redevelopment of the existing systems, whose specifications are sometimes obsolete and no more corresponding to the actual version of the software.

In this paper we present[*] a totally automatic approach towards the reconstruction of the software documentation and possible code re-engineering. We move from the source code by using a static code analyser and capture information pertinent to higher level design phases that are subsequently imported into the ADW CASE tool.

Keywords:
Software reverse engineering - Maintenance - Software documentation - CASE Tools

1. Introduction

It is well known that the most relevant part, up to 95% according to some estimates ([1]), of the EDP departments' activities is dedicated to the maintenance of applications. In particular, we have to outline that in the latest years the maintenance is enhancing in order to delete errors, meet new requirements, improve both design and performance, interface new programs, modify data structures and formats, test new hardware and software techniques. However, it must be noted that maintenance interventions may alter the features of the original software applications, and may therefore substantially contribute to their degrade. In fact, alterations to the data structures, the documentation out of date, the turn over of the personnel in charge of the project (and it is well known that in many cases only the people involved are aware of the real features of the application), make the application system itself less reliable and maintainable, error prone, difficult and expensive to be modified. Application systems presenting poor or even not existing documentation cause further problems during the maintenance work because the user has to rebuild the knowledge before making any change. In addition, old technology or poor quality code based systems show other problems (low performance, limited integration with other systems, difficulty to test) which decrease the productivity.

This situation faces the development of new application systems and the adoption of the modern CASE tools, which are claimed to be the most effective way of developing new applications, at lower costs and at a higher quality level ([2], [3], [4], [5]). As a consequence, there is a tendency towards the implementation of tools that can help in understanding, improving and managing existing systems, even if implemented in non CASE environments, in order to recover existing software portfolio and manage it by using CASE techniques.

In this paper we present an automatic approach towards the reconstruction of the software documentation and the re-engineering of code; we collect data from the source code which are, subsequently, manipulated and imported in a CASE tool, thus rebuilding information pertinent to higher level design phases and giving the possibility of implementing a new engineering phase.

2. The re-engineering

Up to the end of the last decade, users who intended to enhance the features or the performances of their software, had only two alternatives: leave all the software unchanged, or face the costs of a complete re-development. A new alternative, which is presently object of a relevant research effort, is available since the beginning of this decade: the *re-engineering* ([6], [7]). The re-engineering gives the possibility of updating the technology the application system is based upon, without incurring relevant management costs.

More precisely, the software re-engineering consists in the analysis of an existing software, for the sake of giving a more readable structure to it or, perhaps, to represent its

* This work has been partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R

structure in an alternative form, thus allowing a new, different implementation. This process is normally accomplished in two successive steps: first of all, we analyse the system, capturing information about its components, afterwards, we operate a restructuring, thus getting a more standardised form. In some cases, it could happen that the user intends to operate a transformation of the existing application, either because a different language or a different data management system are to be used, or because an interface towards a different tool must be implemented. In these cases, we can clearly identify two different phases, as depicted in fig. 1. In the first phase we operate a reverse engineering (RE) where we abstract the characteristics of the product that we have to re-configure. Subsequently, we operate a forward engineering (FE) process, which is in charge of implementing the new product.

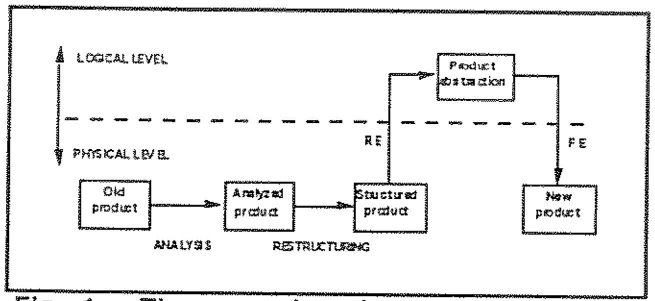


Fig. 1 - The re-engineering process

The reverse engineering phase is the focus of the re-engineering process. Starting from any life cycle level and with the automatic tools' support, such process does not aim at modifying system objects, but only at *improving comprehension* by generating and synthesising independent implementation abstractions.

3. Related work

Recently, the software engineering market has been enriched by the appearance of many reverse engineering and re-engineering tools. These tools allow to solve partially the problem of recovering the design of existing systems by capturing information that is explicitly represented in the source code.

Bachman/Data Analyst is part of the Bachman Product Set, an integrated collection of CASE tools for maintaining and developing IBM mainframe applications. This tool permits to reconstruct the E-R model by examining COBOL, IMS, PL/1 data structures and by extracting, via Bachman/DBA, information from DB2 or SQL/DS catalog. Thus we can modify the old application at a high level (analysis) and, then, generate a new application by executing a forward engineering step. In other words, Bachman Product Set enables the maintainer to make an almost completely automatic re-engineering cycle even if it is limited to the system data architecture.

Cscope, for the UNIX environment, is an interactive tool that makes possible to abstract information from a program by cross-referencing various source code entities.

TIBER (Techniques and Instruments to Build Encyclopaedias of Redevelopment engineering) is a set of methods, techniques, instruments and know-how through which existing COBOL applications are re-documented, reorganised and restructured inside the ADW repository named "Encyclopaedia". ADW is a CASE tool implemented by KnowledgeWare (USA) for the OS/2 environment, selected by IBM as one of the International Alliance products for the AD/Cycle platform.

TIBER analyses COBOL source code, JCL, TP maps (CICS or IMS/DC) and DDL and produces a system documentation composed by relational metamodel, record layouts, structure charts, action diagrams and video maps. Therefore, such tool can be considered more as a re-systemisation tool rather than a re-engineering one because the number and the quality of the output documents are strictly related to the programming language of the source code analysed.

Furthermore, we want to outline a re-engineering case study towards the IEW (Information Engineering Workbench) CASE tool, that is the MS-DOS version of ADW. This experience consists in an environment that supports the re-engineering of Pascal programs at the design level starting from the information abstracted by the static analyser ATOOL and a subsequent import into IEW via a tool named AIEWTOOL ([8]).

4. Charon: the architecture

Charon is a tool implementing a re-engineering cycle that has a C program, where EXEC SQL statements are embedded, as source and has a different version of the same program, written in COBOL/CICS/DB2 environment, as target (fig. 2). The tool takes its name from the son of Erebus who in Greek myth ferries the souls of the dead over the Styx.

The main task fulfilled by *Charon*, and this is another reason for its name, is to act as a bridge between a reverse engineering tool and a forward engineering one, collecting information from different sources. The aim of the whole task is to comprehend the behaviour of the programs and their relationships with other components of the application system. Last, but not least, *Charon* can capture information pertaining to higher level of abstraction, namely the design and analysis phases. This can be of considerable help in the subsequent phases of maintenance and/or forward engineering.

The reverse engineering step is performed by the static C code analyser C-TOOL, the forward engineering one is executed by the ADW COBOL code automatic generator; we are committed to convert the first tool output into the second one proper objects.

C-TOOL evaluates some structural and dimensional metrics and produces the nesting tree and control graph of the program. In addition, the embedded SQL statements are extracted. At the same time, all information about relations accessed by the SQL statements are exported from the catalog and stored in some supporting tables (IBM Query Manager).

The contents of files and tables are processed, and information relevant for the design and analysis phases are collected and stored in an appropriate format, compatible with the import/export format of the ADW. Such information is also inserted in the SQL supporting tables, making possible the implementation of the presented approach in other development environments (StP, Bachman).

After the creation of an empty encyclopaedia related to the project and the subsequent import phase of the *Charon* output, the user finds the extracted information represented by most of the diagrams of the Analysis and Design workstation.

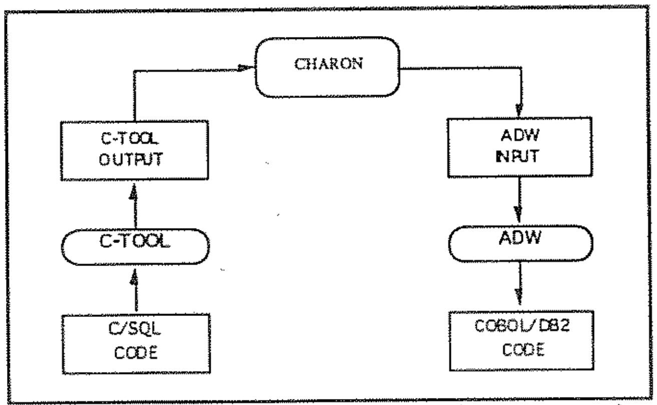


Fig.2 - The re-engineering cycle implemented by Charon

At the present stage, Charon takes into account only the data that are stored as relational tables and the local variables that refer to them, while the traditional program data structures are not processed. In fact, as far as the data architecture is concerned, the user can access Relational DB Design, Data Structures, Data Types, SQL Statements of the design level, and Entity-Relationship Diagram, Entity Attributes Description of the analysis level.

Concerning the processes' architecture, two diagrams of the design level are available: Structure Chart and Action Diagram. They describe respectively the program decomposition in a modules' hierarchy and the logic of a module. Therefore the user can modify, add or delete data, so starting a complete redevelopment of the project, with an obvious improvement of the overall software quality.

The local data structures appear in the Action Diagram, so that the user can have a coherent and complete documentation of the original program. In the Action Diagram construction phase, some instructions of the C programming language are translated into the COBOL equivalent ones. Typical examples are the "switch", the "break" and the "continue", that are translated into "multiway IF" and appropriate "exit" commands.

Starting from the information extracted and imported in the ADW/DWS module, *Charon* also proceed to reconstruct the database conceptual model that belongs to the upper level (AWS).

Charon differs from the similar experiences described in the previous paragraph under several aspects. In fact, it differs from TIBER and Cscope because they are essentially re-documentation tools. On the other hand, as Charon takes into account both the data and processes' aspects, it differs from Bachman/DA, that is concerned only with the data re-engineering, and from AIEWTOOL, that concentrates on the reconstruction of the processes' architecture.

In the subsequent paragraphs we will briefly describe the main features of C-TOOL and of ADW. The reader interested to deeper technical details about ADW, can find in Appendix 1 a list of the diagrams provided by the tool and a more detailed description of their functionality.

5. Reverse engineering through C-TOOL

C-TOOL involves two procedures, subsequently executed, which can analyse a syntactically corrected C program:

- **ANADIM**, which records on the SQL_STAT file all the EXEC SQL commands being in the source C code and their positions;
- **ANASTRU**, which produces the program Call Graph, the Nesting Tree and the Control Flow Graph.
  The first graph represents the program decomposition into modules (procedures and functions) also including the main program. An edge exists between two nodes N_1 and N_2 if N_1 calls N_2 in the program.
  The Nesting Tree describes the program as a tree in which terminal nodes correspond to the simple instructions (assignment, read, write, call, return) and the intermediate nodes correspond to the control structures: sequence, alternative, repetition. An edge exists between two nodes N_1 and N_2 if N_1 includes N_2 in the program.
  The last representation diagram shows the control flow between all the program components. The nodes are of the same type of the Nesting Tree ones. An edge exists between two nodes N_1 and N_2 if N_2 can follow N_1 in a program execution.
  These diagrams are presented in a table form in the following files: OUT2, GRAPH and NESTING.

6. The ADW I-CASE workbench

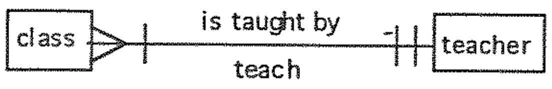
ADW, as most of the CASE tool supporting systems development, is based on the software life cycle Waterfall Model concepts and methodologies. It is modular and includes four workstations, i.e. Planning, Analysis, Design, Construction, which reflect the upper four levels of that model. The user can manage the information, stored in a repository, the "Encyclopaedia", by using some structured analysis and design techniques as SADT, DFD, Modular Decomposition, PDL, etc.. In addition, other important characteristics of the ADW information engineering CASE tool consist in the possibility of keeping continuously and automatically the consistency even between a large number of diagrams and of easily navigating through such diagrams.

We'll import in the ADW/Encyclopaedia the information included in a set of ASCII files that can be divided into three groups: ".EXP" files, ".ENC" files and "MASC" files.

6.1 - ".EXP" files

They contain records representing all the program information entities as objects (OI.EXP), associations between objects (AI.EXP) and properties of objects and associations (PI.EXP for the short properties and TI.EXP for the long textual properties).

In the following, the ADW internal representation of a relationship between two classes is depicted.



```
Objects
1000000003,10007,"teacher                    "
1000000004,10007,"class                      "

Associations
2000000022,20044,1000000004,1000000003

Short properties
2000000022,30034,00000,"is taught by    .              "
2000000022,30037,00000,"teach                          "
2000000022,30035,00000,"1                              "
2000000022,30036,00000,"1                              "
2000000022,30035,00000,"M                              "
2000000022,30036,00000,"M                              "

Long textual properties
1000000004,30076,00001,"A_group_of_students_which_study_"
1000000004,30076,00002,"the_same_arguments    "
```

The records of the OI.EXP file have three fields: the first one is for a token (*Instance token*) which uniquely identifies an object in the project encyclopaedia, the second one is for the type of the object (*Type code*) and the third one is for the name (*Instance name*) given to the object by the user.

The records of the AI.EXP file have four fields: the first one is for the instance token of the association, the second one is for the type code, the third one is for the instance token of the source object of the association and the last one is for the instance token of the target object.

The properties' files include records with the same structure, but the last field in TI.EXP file is longer than the related one in the PI.EXP.

The first field of the records is for the instance token of the object or the association which the property is related to, the second one is for the type code of the property, the third one is for the row number of the textual properties (*Repetition number*) and the last one is for the value of the property (*Property value*).

6.2 - ".ENC" files

They include the description of the program modules Action Diagrams. In the following, the ADW internal representation for an Action Diagram is showed:

```
$ADTEXT$3.00ENGLISH
T   0000000*SECTION P1-READ-CUSTOMER - WE READ THE CUSTOMER RECORD
    0000100**USING EITHER THE CUSTOMER-NO OR THE CUSTOMER-NAME
B   0000000If CNAME-CUSTOMER-NO NOT = 0
    0000000AND CNAME-CUSTOMER-NO NOT = SPACES
    0000000MOVE CNAME-CUSTOMER-NO TO OPG01-CUSTOMER-NO
D   0000004Customer&oi0003D9%FIND KEY OPG01 USING CUSTOMER-NO
C   0000000Else
    0000000MOVE CNAME-CUSTOMER-NAME TO OPG01-CUSTOMER-NAME
D   0000004Customer&oi0003DA%FIND KEY OPG01 USING CUSTOMER-NAME
E   0000000ENDIF
B   0000000If OPG01-STATUS = SPACES
    0000000MOVE CORRESPONDING OPG01 TO CNAME
X2  0000000EXIT
C   0000000Else
    0000000MOVE 'CUSTOMER NOT FOUND' TO CNAME-ERROR-MESSAGE
X2  0000000EXIT
E   0000000ENDIF
G   0000000
```

Each row in a ".ENC" file contains a record having nine fields:

- the first one identifies the type of diagram simple instruction or control structure,
- the subsequent seven fields concern the diagram layout;
- the last field is for the description of the particular action executed.

6.3 - "MASC" files

These files are used when transferring information from the workbench construction level to the extern, but in our application they hold the masks of the tables SQL DML commands.

These templates are called by the modules through an option named "Using call" and include two groups of records:

- the first group takes into account the general information of the accessed table;
- the second one specifies the SQL command.

In the following, the mask of a SELECT command is presented:

```
67,OPG01CUSTOMER
                    <table creator>     <table creation date>
83,MLOPG01A100000CUSTOMER
83,MLOPG01B100000OPS DB.CUSTOMER
83,MLOPG01C100000DB200TSTOPCUSTSP
83,MLOPG01P100000TV
87,MLOPG01R100000CUSTOMER-NO                              *SELECT
31,MLOPG01R100001B0000001EXEC SQL
54,MLOPG01R100002B0000000SELECT                  CUSTOMER_NO
53,MLOPG01R100003  0000000               ,    CUSTOMER_NAME
54,MLOPG01R100004  0000000               ,    STREET_ADDRESS
28,MLOPG01R100005E  0000000
61,MLOPG01R100006B0000000INTO                   :OPG01-CUSTOMER_NO
60,MLOPG01R100007  0000000               ,    :OPG01-CUSTOMER_NAME
61,MLOPG01R100008  0000000               ,    :OPG01-STREET_ADDRESS
28,MLOPG01R100009E  0000000
55,MLOPG01R100010  0000000FROM                  OPCUST_TABLE
57,MLOPG01R100011B0000000WHERE              CUSTOMER_NO =
68,MLOPG01R100012  0000000                    OPG01-CUSTOMER_NO
28,MLOPG01R100013E  0000000
31,MLOPG01R100014E 0000001END-EXEC
```

6.4 - The import process

The *Charon* output information must be transferred in the ADW environment according to three ways of importing (Fig. 3).

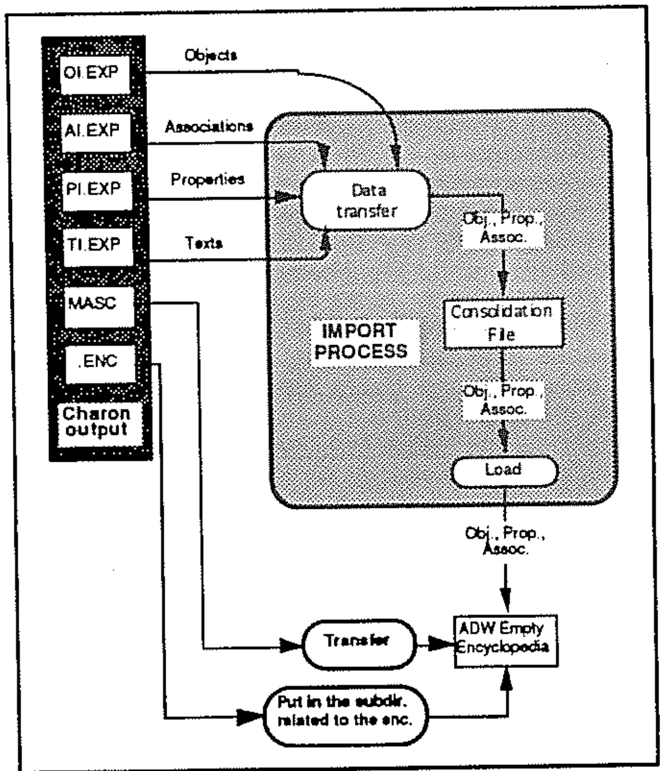


Fig. 3 - The data flow of the ADW empty encyclopaedia populating.

The ".EXP" files are imported by running the *Encyclopaedia Data Transfer* option of the *File* menu of the *Encyclopaedia Services* task in any workstation.

The ".ENC" files must be copied in the ADW subdirectory associated to the empty encyclopaedia.

The "MASC" files are transferred by starting the *Transfer* option of the *File* menu of the *Code Generator* task in the Construction workstation (CWS).

7. - *Charon*: the conversion tool

The conversion from the source program into the ADW tool occurs in several steps, as may be seen from Fig. 4.

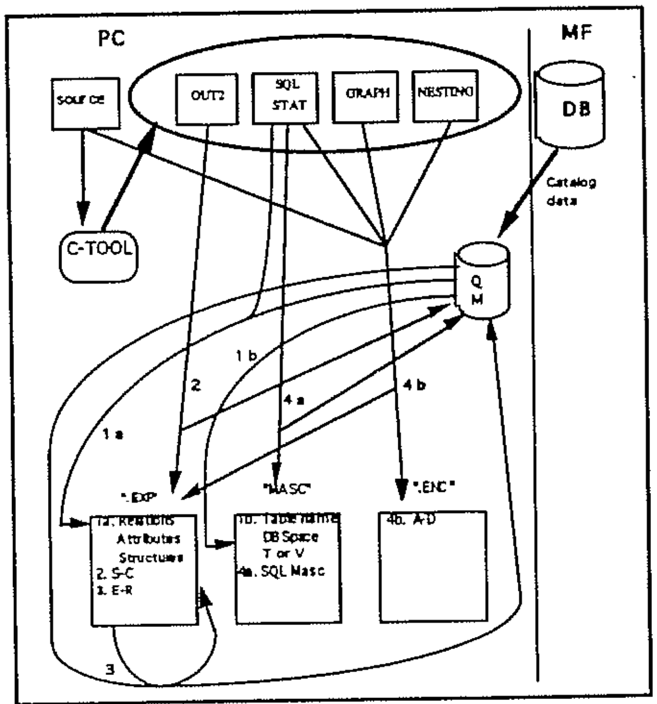


Fig. 4 - The four steps in *Charon*

In more detail, the sequence of the actions is the following:

- *Step 1*
  1a) Create the ".EXP" files' records to represent the logical relational model and the data structure of the relational tables (keys, data types, formats).
  1b) Create and open "MASC" files for coding the general features of the tables (physical name, SQL type, space name).
- *Step 2*
  Reconstruct the Structure Chart.
- *Step 3*
  Generate the E-R model by observing the contents of the ".EXP" files and QM tables.
- *Step 4*
  4a) Represent the modules procedural logic, including the accesses to the database and the calls to other modules.
  4b) Transform the EXEC SQL statements into masks to be inserted into "MASC" files. The statements that are not supported by ADW are inserted as comments.

Conclusions and future developments

In many cases, software quality improvement may require a redevelopment of the application system. When a software development methodology and CASE tools have been adopted as an enterprise standard, recovering the existing software, and documenting it according to these standards may constitute a consistent improvement. It is obvious that understanding the semantics of the original programs is a key point. As a matter of fact, the re-engineering and the adoption of CASE tools, especially in large scale projects, may produce relevant advantages, namely consistency, easy maintenance and clean documentation ([9]). The consistency with the enterprise standards may in fact assure the complete integration of the various subsystems, and reduce the maintenance effort. In fact, the maintenance personnel will no more be forced to operate maintenance interventions on the source code, with the consequence of being tied to a particular set of programs whose code they are aware of, but can operate on higher level specifications, leaving to the CASE tool the burden of the generation of the code. Needless to say, this style of work assures that the documentation will be kept up to date.

In this paper, *Charon*, a tool for the re-documentation and re-engineering of code, has been discussed. It converts the information extracted from a C program by the static code analyser C-TOOL in ASCII records and inserts them in three different typed sets of files.

The files are subsequently imported in the ADW CASE tool environment in order to populate a repository related to the project, thus the user can find the extracted information represented by most of the diagrams of the Analysis and Design level, that is:

- for the representation of general information:
  - *Object List*
  - *Object Details Window*
- for the modelling of data architecture:
  - *Database Relational Diagram*
  - *Data Structure Diagram*
  - *Data Type Window*
  - *DB2 General Information Window*
  - *Entity-Relationship Diagram*
  - *Entity Type Description*
  - *SQL Action Diagram*
- for what concerns processes:
  - *Structure Chart*
  - *Module Action Diagram*

A great advantage offered by the automatic translation operated by *Charon* is that, after the reverse engineering phase, the user can rely on all of the ADW and supporting database manager report writers in order to produce a textual documentation. The integration of the analysed code with the enterprise wide standards can be considered as a benefit too.

Even more important is the fact that the user can operate directly on the high level specifications of the software, getting all the benefits claimed by the CASE tools. In fact, with the rebuilding of E-R model, we create a system knowledge at a as high as possible level so that all the ADW capabilities will be exploited. In this way, if we wish to make some relevant changes to the database structure, we shall easily act on the E-R schema and, then, obtain a normalised relational form by running the Relational Translator task. At that point, according with a bottom-up approach, the user will be able to generate the remaining documentation of the workbench upper levels.

If, on one side, *Charon* can be considered as a test case explaining and making real software engineering capabilities, on the other one, it can be considered as a reference for those who are going to modify software application packages, perform environment conversions, generate code from prototypes.

Moreover, *Charon* completes the C-TOOL reverse results by providing, in some tables, information about database relations, fields, relationships and module calls of the source code, and, in ASCII files, the procedural logic description of each module.

However, it should be stressed that the approach followed in *Charon* requires that the user will conform to some specific design methodology (e.g. the Yourdon Constantine Structured Analysis).

In addition, some code characteristics can't be recovered (SQL dynamic commands, modules formal parameters and input/output conversations) because their conversion from a C environment to a COBOL one is very difficult to make in a completely automatic way.

Perhaps, it would need a human intervention, even if the manual approach, more flexible and friendly, could be too much heavy in the re-engineering of high complexity systems. This considerations suggest that it would be worthwhile to consider the possibility of integrating the positive aspects of the two different approaches.

In order to improve Charon, thus enhancing the set of convertible information and the collaboration with the final user, we have to face the following problems:

- testing the existence of the relationships between relations when examining the SQL commands;
- creating another ADW diagram, Screen Layout, and producing CICS code for the activation of video maps which correspond to the I/O commands in the C code;
- enhancing the Structure Chart diagram by considering the recursivity;
- representing modules formal parameters and converting their C types in COBOL ones in order to allow the generation of the Data Flow Diagram in the Analysis level (AWS);
- implementing an enriched user interface.

Acknowledgements

C-TOOL is a tool developed by the CRIAI (Consorzio Campano di Ricerca per l' Informatica e l' Automazione Industriale), and has been modified for this work in order to intercept the SQL statements. We acknowledge our colleagues from CRIAI for their kind support.

We have also to thank the anonymous referees, which helped in proving clarity of the paper.

References

[1] *Software Re-engineering Symposium*, organised by SYSTECH Systems Technology Institute (Rome, 12-14 February 1990)

[2] Martin J.: *Recommended diagramming standards for analysts and programmers. A basis for automation*, Prentice-Hall Inc., Englewood Cliffs (1987)

[3] Martin F.M.: *Second-Generation CASE Tools: A Challenge to Vendors*, IEEE Software (March 1988)

[4] Martin J.: *CASE & I-CASE*, Informatica 70 n.167 and n.168

[5] Lewis T.G.: *CASE: Computer-Aided Software Engineering*, Informa Tex Press (1989)

[6] Chikofsky E.J., Cross II J.H.: *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software (January 1990)

[7] De Carlini U., Cimitile A.: *Il reverse engineering nella analisi, documentazione, manutenzione e validazione del software*, Sistemi Informatici e Calcolo Parallelo: progetto finalizzato CNR: risultati, stato delle ricerche e prospettive, Angeli (1991)

[8] Lanubile F., Maresca P., Visaggio G.: *An Environment for the reengineering of Pascal Programs*, Proceedings of IEEE Conf. on Software Maintenance, Sorrento, August 1991, pp.23-30

[9] Signore O., Celiano F.: *From a "well designed" database to AD/Cycle tools: a reengineering experience*, Proceedings of CASE and Applications Development in Practice, SHARE Europe (SEAS) Spring Meeting 1991, Lausanne, Switzerland, April 8-12, 1991, pp. 1-8, ISSN 0255-6464

Appendix 1.

A description of the diagrams the user can manipulate after the import process is presented in the following.

The *Object List* lists all the encyclopaedia objects that meet some requirements (name, type, related level in the life cycle model). It combines the report and diagram fashions.

The *Object Details Window* displays general information about an object (name, type, creation date, last update date, definition, comment).

The *Entity-Relationship Diagram* is the well-known Chen diagram including the entities (fundamental, associative or attributive) and the relationships between them.

The *Entity Type Description* describes for each entity the attributes, with their types and cardinality constraints, the relationships the entity is involved in and the keys.

The *Database Relational Diagram* models the database according to the relational logic model. References between relations display cardinalities and show which relations have foreign keys that refer to the primary keys in other relations.

The *Data Structure Diagram* highlights the properties of a stored data. A data structure can be composed of any combination of data elements or data groups. A data element is an atomic unit containing no other structure; on the contrary, data groups can include data elements and/or other data groups. A data structure can describe the structure of a relation, a file record, a segment and screen layout variable. In the case we deal with, data structure describes relations properties like primary and foreign keys, SQL data type, mandatory clause, indexes, etc.

The *Data Type Window* defines the peculiar characteristics of a global or local data type (SQL data type, COBOL format, external and internal length, definition, comment).

The *Structure Chart* offers a graphic depiction of the external, modular structure of a program. It details the hierarchy and organisation of logic units (modules), the distribution of functionality among modules, and the data communication between them.

Module

The *Module Action Diagram* details, in a graphic format, the logic and the structure for a program module. It also establishes the encyclopaedia relationships between modules and other design object types, such as screen layouts, relations, segments, or other modules.

The *DB2 General Information Window* presents the environmental parameters that are bind to a DB2 physical representation of a relation such as DB2 database name, DB2 table space, table name, DDL subject type (table or view).

D M L

The *SQL Action Diagram* enables the generation of a generic template for a SQL DML command.

ADW/Construction generates five different sets of generic DML statements, on the basis of the type you select: CURSOR, DELETE, INSERT, SELECT and UPDATE. A sixth choice, EXPERT, allows to write other DML statements.

Deriving Path Expressions Recursively

Antonia Bertolino[*], Martina Marré[§]

[*] Istituto di Elaborazione della Informazione, CNR, Pisa, Italy.

[§] Dipartimento di Informatica, Università di Pisa, Italy.

Abstract

Program representation plays an important role in software engineering, because it is used by the tools supporting software life cycle activities.

To represent a program's control structure, the dominator tree and the implied tree, derived from the program's ddgraph, can be profitably used. In fact, thanks to their recursive structure, these trees are especially suitable for designing very simple and efficient algorithms for program path analysis, which is widely used in measurement and testing activities.

In particular, this paper presents a recursive algorithm PE for computing path expressions from the dominator and the implied trees.

The algorithm proposed is of interest to program comprehension for two reasons: representation of programs by path expressions is widely applied, e.g., to testing, data flow analysis and development of complexity metrics. More in general, an algorithm as PE, which computes path expressions from flowgraphs, can be used to solve many kinds of path problems.

1. Introduction

The language of *regular expressions* provides a formal setting in which the structural properties of programs can be represented. More precisely, the *path expression* subset is used; this is obtained by restricting the regular expressions to that set of strings which denotes paths in the program flowgraph.

In general, path expressions can be used for program documentation, measurement and testing (e.g. [5], [7]). In particular, path expressions can play an important role in program comprehension as: i) they supply a unified and efficient approach to the solution of path problems [8], [9]; ii) they provide a language-independent model for the representation of program code and low-level design and iii) they can be easily processed by automated tools.

Traditionally path expressions are derived using node elimination techniques[5][9]. In this paper, we propose a new approach, based on the *dominance* and *implication* relationships between the arcs of a *ddgraph* (decision-to-decision flowgraph) $G=(V,E)$, with a unique entry node e_0 and a unique exit node e_k. Dominance has been known for a long time in graph theory [6], but so far has not been fully exploited in program comprehension. In fact, dominance and its symmetric relationship, implication, form two trees of the arcs in a ddgraph G, DT(G) and IT(G) respectively, on which very simple, recursive procedures can be designed.

In the paper, an algorithm, called PE, is described that visits in preorder DT(G) and recursively derives the simple form of the path expression which represents all possible entry-exit paths in G. The fact that PE uses only the dominator tree and the implied tree of a ddgraph to construct the path expression demonstrates that these two are fully sufficient to represent a program's structure.

The paper is organised as follows: Section 2 provides the background necessary for the understanding of the algorithm; Section 3 describes the algorithm giving several examples; some concluding remarks are made in the final Section.

2. Basic concepts

2.1 Graph terminology

Flowgraphs [4] are the most widely used model to represent program structure. A flowgraph is a *directed graph* (or *digraph*) $G=(V,E)$, where V is a set of *nodes* or *vertices* and E is a set of *directed edges* or *arcs*. A directed edge $e=(\mathrm{T}(e),\mathrm{H}(e))\in V$ is an ordered pair of *adjacent* nodes, called *Tail* and *Head* of e. We say that e *leaves* T(e) and *enters* H(e). If $\mathrm{H}(e)\equiv\mathrm{T}(e')$, then e and e' are called *adjacent* arcs. For a node $n\in V$, *indegree*(n) is the number of arcs entering it and *outdegree*(n) is the number of arcs leaving it.

A program's control flow may be mapped onto a flowgraph model in different ways. Henceforth, we shall use the *ddgraph* model [1], which is particularly suitable for the purposes of program path analysis [3]. Ddgraph's arcs are associated to program's blocks (thus ddgraphs revert the more typical usage in flowgraphs of associating blocks to nodes), where a program block, or branch, is a maximal sequence of program statements such that control flow can be transferred only to the first statement

0-8186-4042-1/93 $03.00 © 1993 IEEE

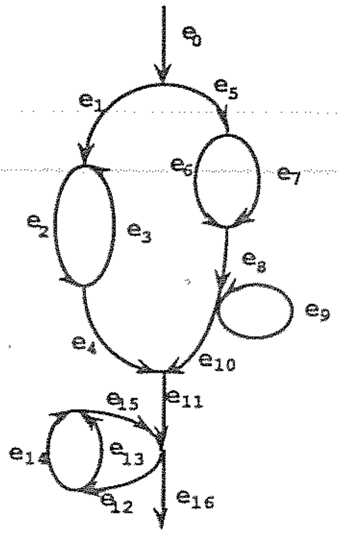


Figure 1: a ddgraph G_1

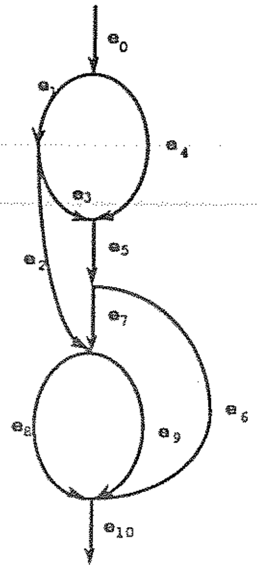


Figure 2: a ddgraph G_2

of the block and, once this first statement is executed, all the statements in the block are executed sequentially. Ddgraph's nodes either correspond to program's predicates, where a branching in the program's control flow occurs, or represent the joining of separate control flow streams. Formally, we give the following definition.

Definition 1: *ddgraph*
A *ddgraph* is a digraph $G=(V,E)$ with two distinguished arcs e_0 and e_k (which are the unique entry arc and exit arc, respectively), such that any other arc in G is reached by e_0 and reaches e_k, and such that for each node $n \in V$, $n \neq T(e_0)$, $n \neq H(e_k)$, (indegree(n) + outdegree(n)) > 2 (while indegree(T(e_0))=0 and outdegree(T(e_0))=1, indegree(H(e_k))=1 and outdegree(H(e_k))=0).
Figure 1 and figure 2 above show two simple ddgraphs G_1 and G_2. G_1 has distinguished arcs e_0 and e_{16} and G_2 has distinguished arcs e_0 and e_{10}.

Obviously, for a strictly sequential program (i.e. a program only consisting of "sequence" control structures), the ddgraph $G=(V,E)$ will consist of just one arc, i.e. $E=\{e_0 \equiv e_k\}$ and $V=\{H(e_0), T(e_0)\}$. This is called the *trivial* ddgraph.

A *path p* of length q in a ddgraph G is a sequence $p=n_{j_0}, e_{i_1}, n_{j_1}, \ldots n_{j_{q-1}}, e_{i_q}, n_{j_q}$, where $T(e_{i_k})=n_{j_{(k-1)}}$ and $H(e_{i_k})=n_{j_k}$, $k=1,\ldots,q$. We will also write $p=e_{i_1},\ldots,e_{i_q}$. An *entry-exit path* $p= e_0, \ldots, e_k$ is a path starting from the entry edge and finishing with the exit edge. A path p is *simple* if all its nodes are distinct. A path $p=n_{j_0}, e_{i_1}, n_{j_1}, \ldots n_{j_{q-1}}, e_{i_q}, n_{j_q}$, is a *cycle* if $n_{j_q}=n_{j_0}$. A *simple cycle* is a cycle in which all the internal nodes are distinct. An *acyclic* ddgraph is a ddgraph that has no cycles.

A (*rooted*) *tree* $T=(V,E)$ is a digraph, in which one distinguished node, called the *root*, is the Head of no arcs; every node except the root is the Head of exactly one arc and there exists a (unique) path from the root to each node. If there is an arc $e=(n_i,n_j)$ in T, n_i is said the *parent* of n_j and n_j is said a *child* of n_i. Tree nodes of zero outdegree are said *leaves*.

2.2 Dominance and implication

An important concept in the graph literature is the *dominance* relation, which imposes a partial ordering on the nodes of a flowgraph. Since in ddgraphs program branches are associated to arcs, we here are interested in applying the dominance relationship to the arcs instead of to the nodes. Therefore, we give a definition of dominance between arcs in a ddgraph.

Definition 2: *Dominance*
Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k. An arc e_i *dominates* an arc e_j if every path p from the entry arc e_0 to e_j contains e_i.

Several algorithms have appeared in the literature to find the dominators in a digraph [6]. These can be adapted very easily to find dominators in a ddgraph.

By applying the dominance relationship between all the arcs of a ddgraph G, we can obtain a tree, called *Dominator Tree* (*DT*(G)). The nodes in DT represent the ddgraph arcs and the root corresponds to the entry arc e_0. For each pair (e_i, e_j) of adjacent nodes in DT,

e_i=Parent(e_j) is the *immediate dominator* of e_j. The immediate dominator e_i of an arc e_j is a dominator of e_j with the property that any other dominator of e_j also dominates e_i. Notice that each arc (different of e_0) has exactly one immediate dominator. In figure 3, DT(G_1) is shown.

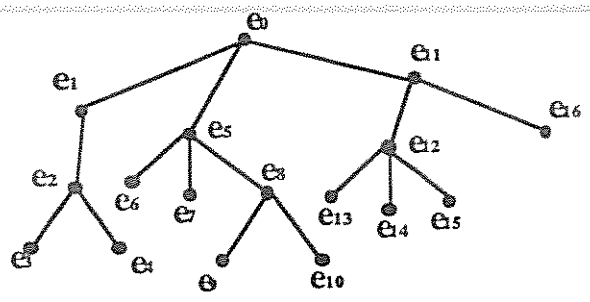


Figure 3: the dominator tree DT(G_1)

Following, we introduce the "symmetric" relation of *implication* between arcs in a ddgraph.

Definition 3: *Implication*
Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k. An arc e_i *implies* an arc e_j if every path P from e_i to the exit arc e_k contains e_j.

The implied arcs in a ddgraph G with distinguished arcs e_0 and e_k can be found as the dominators of the ddgraph G' having distinguished arcs e'_0 and e'_k, in which every arc e' is obtained by reverting a corresponding arc e in G (i.e. H(e')=T(e) and T(e')=H(e)), e'_0 corresponds to the reverse arc of e_k and e'_k corresponds to the reverse arc of e_0.

By applying the implication relationship between the arcs of a ddgraph, we can obtain a tree rooted at e_k, called *Implied Tree* (*IT*(G)). For each pair (e_i, e_j) of adjacent nodes in IT(G), e_i=Parent(e_j) is *immediately implied* by e_j. An arc e_i is immediately implied by an arc e_j if e_i is implied by e_j and any other arc which is implied by e_j is also implied by e_i. Notice that each arc (different of e_k) is immediately implied by exactly one arc. In figure 4, IT(G_1) is shown.

2.3 Structured ddgraphs

Structured programs are often defined informally in terms of GOTO-less programs. Recently, a wide, rigorous theory has been introduced [11], in which different classes of structuredness are defined. Particularly, the class of S_1-structured flowgraphs corresponds to the programs normally referred as structured in the literature, namely those only based on the control structures *sequence*, *if-then-else*, *while-do* and *repeat-until*. In this subsection we present a constructive definition of S_1-structured ddgraphs, or, briefly, S_1-ddgraphs, based on this theory.

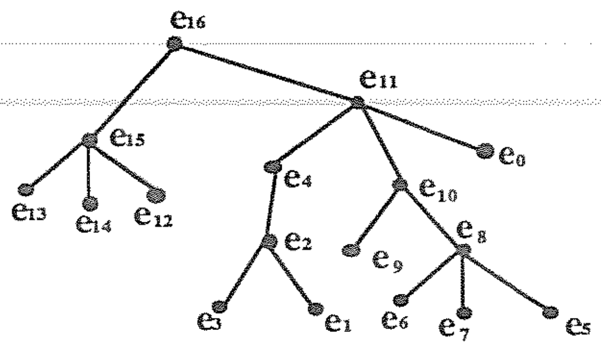


Figure 4: the implied tree IT(G_1)

A strictly sequential program is represented by the trivial ddgraph; then the trivial ddgraph is a basic S_1-ddgraph. Similarly, the other basic S_1-ddgraphs can be immediately identified as those in figure 5 below.

The class of S_1-structured ddgraphs is the class of those ddgraphs that can be constructed, starting from the basic S_1-ddgraphs, by the repeated application of a composition operation, as formally stated by the two following definitions.

Definition 4: *Composition of ddgraphs*
Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k, $G'=(V',E')$ be a ddgraph with distinguished arcs e_0' and e_k'. Let e be an arc in E. The ddgraph $G(G'$ in $e)$ is defined as the ddgraph $G''=(V'',E'')$ with distinguished arcs e_0'' and e_k'', which is obtained substituting the arc e in G by the graph G'. G'' is called the *composition* of G with G' in e.

Let us observe that, by construction, G'' is a ddgraph.

Definition 5: *S_1-ddgraph*
The class of *S_1-ddgraphs* is the smallest class of ddgraphs that satisfies the following conditions:
i) every basic ddgraph in figure 5 is a S_1-ddgraph;
ii) if G and G' are S_1-ddgraphs, and e is in G, then $G(G'$ in $e)$ is a S_1-ddgraph.

Let us note that the ddgraph G_1 in fig. 1 is S_1-structured, while the ddgraph G_2 in fig. 2 is not.

It can be easily verified that, for a S_1-ddgraph G, every node in DT(G) or in IT(G) can have at most three children (see also figure 6). This property will be exploited later in the PE algorithm (Section 3).

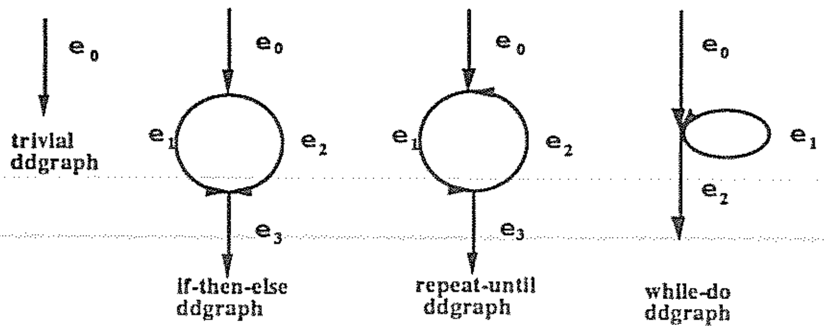


Figure 5: the basic S_1-ddgraphs

2.4 Path expressions

In this subsection path expressions are introduced. Path expressions fully represent the structure of a program by means of strings of labels and operators from the set [+, $\cdot$, *], as flowgraphs do graphically by nodes and arcs.

Let L be a finite alphabet disjoint from $\{\Lambda, \emptyset\}$; we introduce first the set of *regular expressions* over L by the following recursive definition:

i) the empty string Λ and the empty set $\emptyset$ are (atomic) regular expressions;

ii) for any symbol $x \in L$, x is an (atomic) regular expression;

iii) if R_1 and R_2 are regular expressions, then the *concatenation* $(R_1 \cdot R_2)$, the *union* (R_1+R_2) and the reflexive, transitive *closure* $(R_1)^*$ are (compound) regular expressions.

Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k. The set of *path expressions* over E is a subset of the regular expressions over E, obtained by restricting the latter to those strings which denote sets of paths in G.

To assign a meaning to path expressions, we introduce(1) the notion of the *type* of a path expression P. P is said of type (n_1,n_2), with $n_1,n_2 \in V$, if P denotes a set of paths in G from n_1 to n_2, and only such paths. Then, in the context of path expressions, we can interpret the above recursive definition as follows:

i) Λ denotes an empty path in G; $\emptyset$ denotes an empty set of paths in G; for any edge $e \in E$, $P=e$ is of type $(T(e),H(e))$, and P denotes the path formed by the only arc e;

ii) if P_1 is a path expression of type (n_1,n_2) and P_2 is a path expression of type (n_2,n_3), then $P=(P_1 \cdot P_2)$ is of type (n_1,n_3) and P denotes the set of paths obtained by concatenation of the set of paths denoted by P_1 with the set of paths denoted by P_2;

iii) if P_1 and P_2 are both path expressions of the same type (n_1,n_2), then $P=(P_1+P_2)$ is also of type (n_1,n_2) and P denotes the union of the set of paths denoted by P_1 with the set of paths denoted by P_2;

iv) if P_1 is a path expression of type (n, n), then $P=(P_1)^*$ is of type (n, n) and P denotes all paths in G obtained by repeating any number of times (possibly none at all) the paths denoted by P.

For example, the path expression P representing all possible entry-exit paths in G_1, i.e. of type $(T(e_0), H(e_{16}))$, is:

$$P=e_0 \cdot ((e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4) + (e_5 \cdot (e_6+e_7) \cdot e_8 \cdot (e_9)^* \cdot e_{10})) \cdot e_{11} \cdot (e_{12} \cdot (e_{13}+e_{14}) \cdot e_{15})^* \cdot e_{16}$$

(1) For brevity, we have omitted the intermediate definition of an interpretation function $s(R)$ of a regular expression R. A more complete definition can be found in [9].

3. The PE algorithm

In this Section, we describe, in Pascal-like form, our recursive algorithm PE which computes path expressions. PE works on S_1-structured ddgraphs. Actually, the algorithm described forward in figure 7 only works assuming that the composition operation between ddgraphs is not performed on the arc corresponding to the back arc of a *repeat-until* structure. The subset of S_1-ddgraphs obtained applying this restriction is said *S_1-proper* and is defined below. Actually, a version of the algorithm working for the whole class of S_1-structured ddgraphs could also be easily derived. Yet, for simplicity, in this paper we preferred to give this version, since, in fact, for ddgraphs representing programs only constructed by means of the control structures *sequence*, *if-then-else*, *while-do* and *repeat-until* the composition which is forbidden can never occur.

Definition 6: *back arc*

Let $G=(V,E)$ be a S_1-ddgraph, and let $e \in E$ be an arc in G. If there exists a simple path p in G from e_0 to e_k, which has the following form:

$p= e_0,p_0,H(e),p_1,T(e),p_2,e_k$
where p_1 is a not empty path from H(e) to T(e), then e is called a *back arc*.

Definition 7: *S_1-proper ddgraph*
The class of *S_1-proper ddgraphs* is the subset of S_1-ddgraphs that satisfies the following conditions:
i) every basic S_1-ddgraph is also a S_1-proper ddgraph;
ii) if G and G' are S_1-proper ddgraphs, and e is not a back arc in G, then $G(G'$ in $e)$ is a S_1-proper ddgraph.

The PE algorithm performs a visit of the DT and uses also the IT in case a cycle is encountered. We suppose that the DT is constructed in such a way that the child of an arc e "nearest" to the exit arc is always the rightmost among its children; for example, the DTs corresponding to the not trivial S_1-basic ddgraphs (figure 5) are shown in figure 6 below. No order is required for the IT.

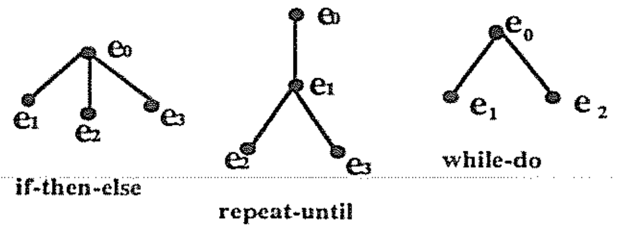


Figure 6: the DTs for the basic S_1-ddgraphs

Given a S_1-proper ddgraph $G=(V,E)$, with distinguished arcs e_0 and e_k, the PE algorithm computes a regular expression P of type (T(e_0), H(e_k)), which represents the set of all entry-exit paths in G.
We shall use the following operations:
- *ch-1(e)*, *ch-2(e)*, *ch-3(e)*, which return, respectively, counting from left to right, the first, the second and the third child of the arc e in the DT.
- *i-imp(e)*, which returns the arc immediately implying e.

```
Function PE(ea, eb): integer;
begin
if (number of children of ea=0) or (ea = eb) then return ( ea )                                        {1}
elsif (number of children of ea=1) then return ( ea•PE(ch-1(ea), eb) )                                 {2}
elsif (number of children of ea=2) then
      if (i-imp(ch-1(ea)) dominates ch-1(ea))
      then return ( ea• (ch-1(ea) •PE(i-imp(ch-1(ea)), ea))* • PE(ch-2(ea), eb) )                      {3}
      else return ( ea• (PE(ch-1(ea), eb))* • PE(ch-2(ea), eb) )                                       {4}
else {i.e. number of children of ea=3 }
      return ( ea• (PE(ch-1(ea),eb) + PE(ch-2(ea),eb)) • PE(ch-3(ea),eb) )                             {5}
end function;
```

Figure 7: the PE algorithm

Let us illustrate by some examples how the PE algorithm works:

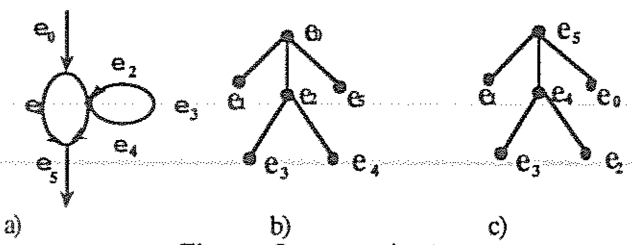


Figure 8: example 1.

Let us apply PE to the simple ddgraph in fig. 8a above between the arcs e_0 and e_5. Since e_0 in DT (fig. 8b) has 3 children, which are from left to right e_1, e_2 and e_5, we have:

$PE(e_0, e_5) = e_0 \cdot (PE(e_1, e_5) + PE(e_2, e_5)) \cdot PE(e_5, e_5)$ (1)

Solving the recursive calls to PE, e_1 and e_5 have no children, therefore:

$PE(e_1, e_5) = e_1$ (2)

$PE(e_5, e_5) = e_5$ (3)

Instead, e_2 has the 2 children e_3 and e_4; we must find in IT (fig. 8c) which is if the arc immediately implying e_3=ch-1(e_2), it is e_4, and then we must check if e_4 dominates e_3. Since the answer is no, (according to statement {4} of the algorithm) we have:

$PE(e_2, e_5) = e_2 \cdot (PE(e_3, e_5))^* \cdot PE(e_4, e_5) =$

$= e_2 \cdot (e_3)^* \cdot e_4$ (4)

Finally, substituting (2), (3) and (4) in (1) we obtain:

$PE(e_0, e_5) = e_0 \cdot (e_1 + e_2 \cdot (e_3)^* \cdot e_4) \cdot e_5$

which is, already in its simple form, the path expression representing all entry-exit paths in the ddgraph of fig. 8a.

It should be clear now that the IT is used just to discriminate if an arc having two children in the DT precedes a *while-do* cycle (as the arc e_0 in the basic *while-do* ddgraph) or is within a *repeat-until* cycle (as the arc e_1 in the basic *repeat-until* ddgraph). The following example should help to further clarify this concept. The two ddgraphs in fig. 9a and 9b have the same DT, which is shown in fig. 9c, and they can only be distinguished by the PE algorithm looking at their ITs, respectively in fig. 9d and 9e. For example, the arc e_1 in DT has two children, the first of which from left to right is e_2; in the ddgraph of fig. 9a, e_2 returns back to a *repeat-until* cycle, and in fact the immediate implier of e_2, i.e., in fig. 9d, e_1, dominates the same e_2; instead, in the ddgraph of fig. 9b, e_2 enters a *while-do* cycle, and in fact its immediate implier, i.e., in fig. 9e, e_3, does not dominates it.

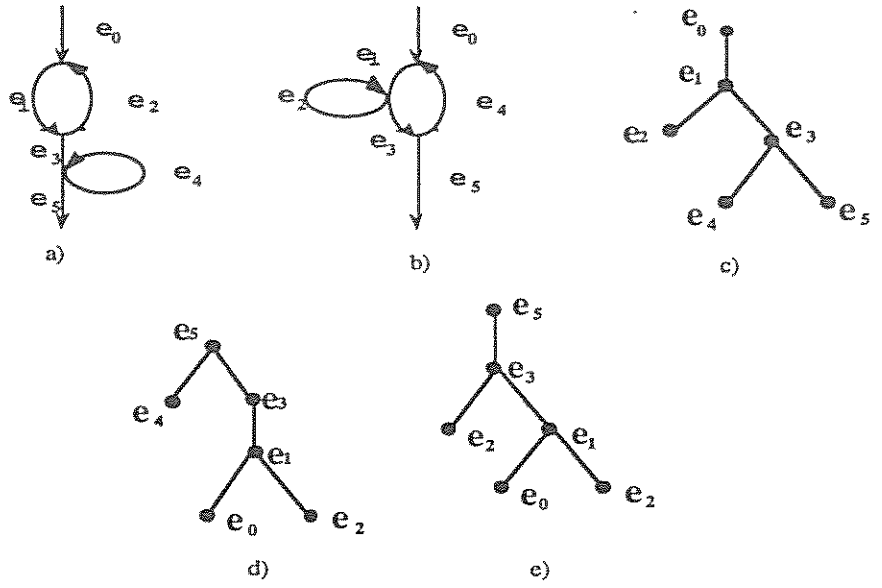


Figure 9: example 2.

Finally, the following example (fig. 10) illustrates the functioning of the algorithm in the case of two nested *repeat-until* cycles:

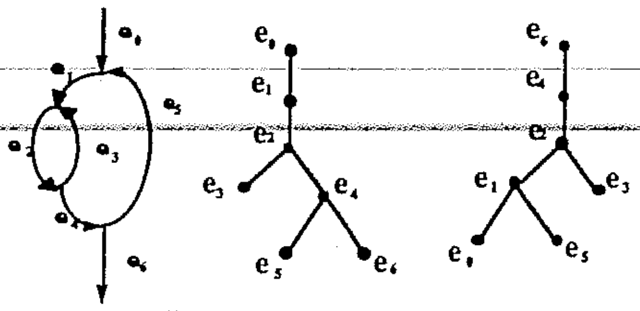


Figure 10: example 3.

$PE(e_0,e_6)= e_0 \cdot PE(e_1, e_6) =$
$= e_0 \cdot e_1 \cdot PE(e_2, e_6) =$
$= e_0 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot PE(e_2, e_2))^* \cdot PE(e_4, e_6) =$
$= e_0 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4 \cdot (e_5 \cdot PE(e_1,e_4))^* \cdot PE(e_6,e_6) =$
$= e_0 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4 \cdot (e_5 \cdot e_1 \cdot PE(e_2,e_4))^* \cdot e_6 =$
$= e_0 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4 \cdot (e_5 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot PE(e_2, e_2))^* \cdot$
$PE(e_4, e_4))^* \cdot e_6 =$
$= e_0 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4 \cdot (e_5 \cdot e_1 \cdot e_2 \cdot (e_3 \cdot e_2)^* \cdot e_4)^* \cdot e_6$

In summary, PE performs a preorder traversal of DT(G) visiting each node once; besides, in case a *repeat-until* structure is encountered, PE must also iterate the visit to a subtree of DT representing the body of the *repeat-until* cycle (recursive call to PE(i-imp(ch-1(e_a)), e_a) in statement {3} of the algorithm). Therefore, it is understood that the performance of the algorithm varies widely with the structure of the program under examination, and particularly in relation with the presence of complex ddgraphs composed with a *repeat-until* ddgraph in the arc e_1 (fig. 5). Roughly, if R is the depth of composition between *repeat-until* ddgraphs, i.e. the number of nested *repeat-until* cycles, PE would perform in $O(|E|^R)$ time. However, with no *repeat-until* cycle, PE would work in $O(|E|)$ time.

Let us observe that the PE algorithm derives the path expression already in its simple form, without no Λ and $\emptyset$ labels and without redundant parentheses.

Theorem 1: *Termination and Correctness of the PE Algorithm*
Let DT(G) and IT(G) be the dominator tree and the implied tree, respectively, of a S_1-proper ddgraph $G=(V, E)$ with distinguished arcs e_0 and e_k. Then the PE algorithm called with parameters e_0 and e_k terminates and returns a path expression which denotes all entry-exit paths in G.

Proof
Termination:
PE called on e_0 and e_k performs a traversal of DT(G). At each node e_a visited, two kinds of recursive call to PE may be made (see fig. 11): either i) PE is called to visit a subtree of DT(G) rooted at a child of e_a, or ii) PE is called to visit a subtree of DT(G) rooted at (i-imp(ch-1(e_a))) which is an ancestor of e_a or is e_a itself. In the first case, the subtree is smaller than (because contained within) the subtree rooted at e_a currently being visited by PE. In the second case, the subtree is limited in the lower part by the arc e_a, at which the traversal will be stopped.

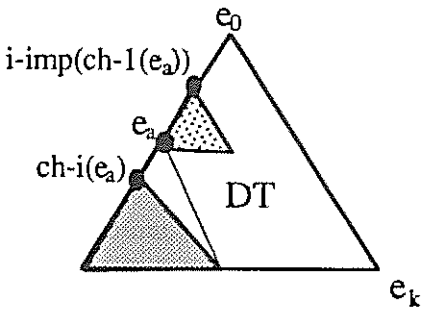


Figure 11: the subtrees visited by PE

Correctness:
The proof is inductive on the construction of the S_1-proper ddgraph G.

If the DT has just one node e_0, there exists a unique entry-exit path, which is $p=e_0$, and the algorithm returns this same arc (statement {1} of the algorithm). If the arc e_0 is not a leaf in DT, then let us consider separately the cases of e_0 having one, two or three children.

i) e_0 has one child e_1:

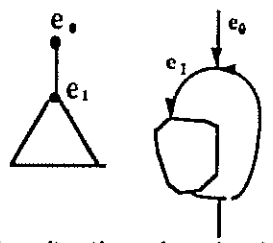


Figure 12: the situation of e_0 having one child

the situation is depicted in figure 12 above, i.e., by construction, e_1 enters the body of a *repeat-until* cycle. In this case, every path from e_0 to e_k is denoted by the path expression obtained concatenating e_0 to the path expression which denotes every path from e_1 to e_k, which justifies statement {2} of the algorithm;

ii) e_0 has two children e_1 and e_2:

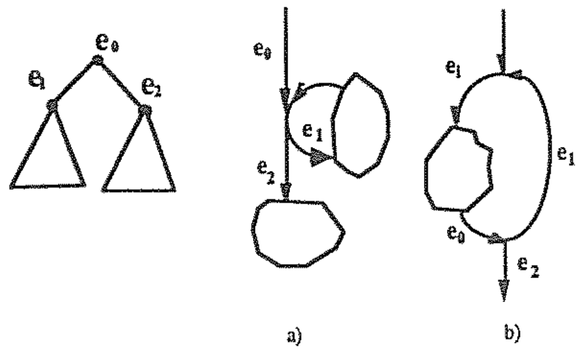


Figure 13: the situation of e_0 having two children

the situation is depicted in figure 13 above, i.e. two cases are possible.

Case a) e_1 enters a *while* cycle and e_2 is the arc just after the exit of the cycle. In this case, every path from e_0 to e_k is denoted by the path expression obtained concatenating e_0 to an arbitrary number (possibly none at all) of repetitions of the path expression which denotes the paths within the *while* cycle, concatenated to the path expression which denotes every path from e_2 to e_k, which justifies statement {4} of the algorithm;

Case b) let us observe that this situation can occur only after at least a recursive call to PE (see where e_0 is in fig. 13 b). In this case, e_1 returns back within a *repeat-until* cycle and e_2 is the arc just after the exit of the cycle. In this case, every path from e_0 to e_k is denoted by the path expression obtained concatenating e_0 to an arbitrary number (possibly none at all) of repetitions of the path expression formed by e_1 concatenated to the path expression which denotes the paths within the *repeat-until* cycle, concatenated to the path expression which denotes every path from e_2 to e_k. In turn, the path expression which denotes every path within the *repeat-until* cycle is the path expression derived by the PE algorithm between the immediate-implier of e_1, say e_i, and e_0, which justifies statement {3} of the algorithm;

iii) e_0 has three children e_1,e_2 and e_3:

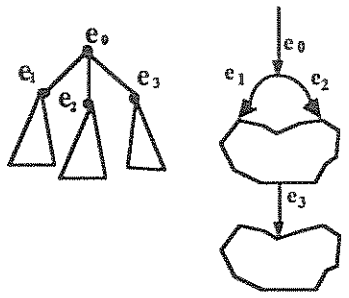


Figure 14: the situation of e_0 having three children

the situation is depicted in figure 14 above, i.e., by construction, e_1 and e_2 enter the *then* and the *else* part of an *if-then-else* statement. In this case the path expression which denotes every path from e_0 to e_k is given by the path expression obtained by concatenating e_0 to the union of the path expressions denoting respectively the *then* and the *else* part of the *if-then-else* statement, concatenated to the path expression which denotes every path from e_3 to e_k, which justifies statement {5} of the algorithm. ♦

4. Concluding remarks

Program representation plays an important role in software engineering, because it is used by the tools supporting software life cycle activities.

Flowgraphs are widely used to represent program structure. A program's control flow may be mapped onto a flowgraph model in different ways. In this paper, ddgraphs have been defined, in which program's blocks are associated to arcs. Then, the symmetric relationships of dominance and implication between ddgraph's arcs have been used to derive the two trees DT and IT, respectively. Dominance has been known for a long time in graph theory [6], but, so far, it has not been fully exploited in program comprehension. In fact, we argue that the DT and IT couple can be profitably used to represents the structure of a program, since, thanks to their inherently recursive structure, they permit very simple and efficient algorithms to be derived for solving many kinds of path problems.

In particular, we have described a recursive algorithm, PE, which can be employed to derive, in a very simple fashion, the path expression denoting all the entry-exit paths of a ddgraph *G*. The PE algorithm derives the path expression in its simple form, without any Λ and $\emptyset$ labels and without redundant parentheses. More importantly, it works almost linearly for real-world programs, if we could assume that real-world programs contain a limited number of nested *repeat-until* cycles. Actually, a number of authors have found that simple programming structures are much more used than complex ones (see, for example, Sect. IV in [10]).

To derive path expressions, node elimination techniques have been traditionally used; an efficient and general-purpose algorithm has already been derived elsewhere [9]. In decribing here the PE algorithm, our aim is not to propose a more efficacious solution; on the contrary, our method is less general than that described in [9].

Instead, our aim was to show how DT and IT can be used by tools performing maintenance, reverse engineering and structural testing, in the design of simple and recursive algorithms for solving path analysis problems. In fact, the method underlying the algorithm can be exploited to solve many other path problems (as for example in [3]). Indeed, it has been previously shown [9] that the construction of path expressions from flowgraphs is in some sense the most general path problem.

Our approach has been already experimented by the implementation of some algorithms within a tool prototype, called BAT, which performs the static analysis of C programs.

Acknowledgements

This work was supported in part by the project "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council (CNR).

References

[1] A. Bertolino, "Unconstrained Edges and their Application to Branch Analysis and Testing of Programs", to appear on *The Journal of Systems and Software*, 1993.

[2] A. Bertolino, M. Giromini, "Easy Branch Testing", *Proc. of the Int. Conf. on Achieving Quality in Software (AQuIS '91)*, pp. 251-258, Pisa, Italy, April 22-24,1991.

[3] A. Bertolino, and M. Marré, "Automatic Generation of Path Covers", IEI-CNR Internal Report B4-59, November 1992.

[4] M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, 1977.

[5] J. Laski, Path Expressions in Data Flow Program Testing, *Proc. COMPSAC*, pp. 570-576, Chicago, Oct. 29-Nov. 2, 1990

[6] T. Lengauer, and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph", *ACM Trans. on Programming Languages and Systems*, vol. 1, pp.121-141, 1979.

[7] K. Magel, Regular Expressions in a Program Complexity Metric, *ACM SIGPLAN Notices*, vol. 17, No. 7, pp.61-65, 1981.

[8] R. E. Tarjan, "A Unified Approach to Paths Problems", *ACM Journal*, vol. 28, No. 3, pp.577-593, 1981.

[9] R. E. Tarjan, "Fast Algorithms for Solving Paths Problems", *ACM Journal*, vol. 28, No. 3, pp.594-614, 1981.

[10] L. J. White, and E. I. Cohen, "A Domain Strategy for Computer program Testing", *IEEE Tr. on Software Engineering*, vol. SE-6, No. 3, pp.247-257, May 1980.

[11] R.W. Whitty, N.E. Fenton and A.A. Kaposi, "A rigorous approach to structural analysis and metrication of software", *Software and Microsystems*, vol. 4, No. 1, pp.2-16, February 1985.

Extracting Application Domain Functions from Old Code: a Real Experience

F.Cutillo(*), F.Lanubile (§), and G.Visaggio (§)

(§) Dipartimento di Informatica
University of Bari
fax: +39-80-243196 email: giuvis@vm.csata.it
(*) StarService s.p.a., Bari, Italy

Abstract[1]

This work deals with the problem of locating domain dependent functions into old application systems and drawing out them for reengineering and reuse. Our approach is based on a particular form of program slicing which makes it possible to recover user functionalities although they are spread over the code. Supported by a commercial tool, the approach has been experimented with a banking application system, whose maintenance problems were increasingly serious. Lessons learned suggest that a successful application of program slicing needs the correct identification of data used as operands and results of the domain function. Moreover some preliminary form of code segmentation may be required to enable program slicing to focus on the expected functionalities.

1: Introduction

Many reverse engineering tools which have been proposed in the past are source code analyzers which make redocumentation [6], i.e. they produce a representation at the same abstraction level but focused on particular views of the program such as control flow [3], data flow [5], program structure [16], and data structure [20].

Although these tools improve the understanding degree of a maintenance programmer, they don't reach an abstraction level such that meaningful concepts can be recognized. In fact, due to poor structuring of the programs, application domain concepts like business functions or entity types are scattered over the code [17] and consequently the recovered design artifacts have a low cohesion. Locating and extracting high cohesion components from old code is a complex task because it requires a radical restructuring of the program structure.

We distinguish between two classes of components which can be located in a program: the former concerns the application domain while the latter depends on the technological environment which holds a system. Application domain components are made up of functions and data which typically characterize a class of problems. Environment-dependent components come from design decisions which specialize a solution with respect to a specific organization and the existing hardware/software platform. The ability of correctly discerning these two classes of components is advantageous for dealing with adaptive maintenance or platform migration which regards only those parts of a program which are affected by the technology progress. On the other hand, the flexibility of a system is increased if those components which are influenced by the domain characteristics can be easily isolated.

While [10] addresses the problem of extracting environment-dependent components from large programs, in this paper we face the problem of the identification of domain-dependent components.

The extraction of high level components from source code, both domain and environment dependent, has been attempted with knowledge-based tools. Knowledge-based tools hold programming concepts at a higher level of abstraction and recognize these concepts into a foreign source code by means of pattern matching techniques [11], [13], [19], [22]. A common assumption is that programs are composed of stereotypical computational patterns, called plans [21]. A plan is an abstract representation of a computation which ignores the syntactic details which depend from the programming language. So, software comprehension is successful if program plans are recognized into the source code. However, our experience with large and old programs brings us to believe that plans are difficult to find

[1]This work has been partially supported by the Finalized Project on "Information Systems and Parallel Computation" of Italian National Research Council (CNR) under grant no.91.00930PF69.

because each time a programmer is assigned a development task, he starts from scratch and implements the required functions in a new form. So, years of maintenance and enhancement by different programmers with different skills and programming techniques cause programs to become overly large and complex and original plans to get lost.

In [14], the recovery of function abstraction was based on a structured approach, called stepwise abstraction, which iteratively rewrites source code into a program design language, starting from prime programs (single-entry, single-exit pieces of programs). The analysis is completed when a full specification, explaining the program behavior, has been obtained. However, this technique has some drawback due to the lack of automatic support for driving the abstraction and the complexity when dealing with loops.

Program reading by stepwise abstraction is coherent with the bottom-up theory of program understanding [8], 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. Top-down 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 [2].

In the case of function abstraction, expectations can derive from the application domain knowledge, especially that derived from the conceptual data model [1]. In fact, the attributes of the entity types suggest the presence of functionalities which create, check and update them. For example, if the "internal check code" attribute is stored in the database then the function which compute them could be hidden inside the program, or if a file records the amount of banking transactions then reverse engineers may look for the function which computes the account balance.

In [9], program slicing has been proposed as a segmentation technique to be used in a top-down way. The goal is to extract application domain functions from an unknown program, by using a partial knowledge of the application derived from both the maintainer expertise and the reverse engineering of data.

In this paper an exploratory experimentation is described. In order to verify soon the approach, a commercial tool which supports the large-scale application of program slicing is used. The extraction procedures have been adapted to the slicing tool. The object of study has been an old Cobol program of a banking application system.

The following section briefly defines the context where function recovery is executed, the third section summarizes the program slicing and the fourth section outlines the extraction criteria. The fifth section describes the experimentation environment and the results achieved. Finally, conclusions present the lessons learned and point out the future work.

2: The context

Figure 1 shows a model of the reverse engineering process [7] which we use to recover data-oriented application, i.e. application systems where most of the tasks involve manipulating and retrieving data from a database. Function recovery is based on the following assumptions.

- There is a working program which is part of the application portfolio of a given organization.
- There is a knowledge about problem and application domain which makes it possible to make hypotheses on the existence of functions inside the program.
- 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. Information which results from the data analysis phase may allow a novice programmer to acquire enough knowledge to become expert for that application because the recovered data model contain all the information for describing the input and output of the business functions. In this way program slicing is used after an hypothesis about the existence of a function has been made on the basis of clues which are provided both by the maintainer expertise and by the presence of derived data into the recovered data model.

3: Program slicing

Program slicing is a decomposition method introduced by Weiser in [23]. Initially proposed for program debugging [24] and parallel processing [25], later on program slicing has been applied to other activities such as module integration [15], cohesion evaluation [18], testing [12] and modification [12].

Program slicing is based on the observation that we are often interested only to 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 [24] an experiment was conducted whose results confirm the hypothesis that programmers implicitly use slices when debugging unknown programs. Program understanding is difficult because slices are often scattered through the entire program, making difficult locating bugs into programs.

The concept of program slicing has been extended in [12] to capture all the computation which is relevant for a given variable. While a program slice depends on a variable and a statement number, its extension, the decomposition slice, depends only on a variable.

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. For each decomposition slice there is a complement collecting all the program instructions which are not affected by modifications in the related slice. 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.

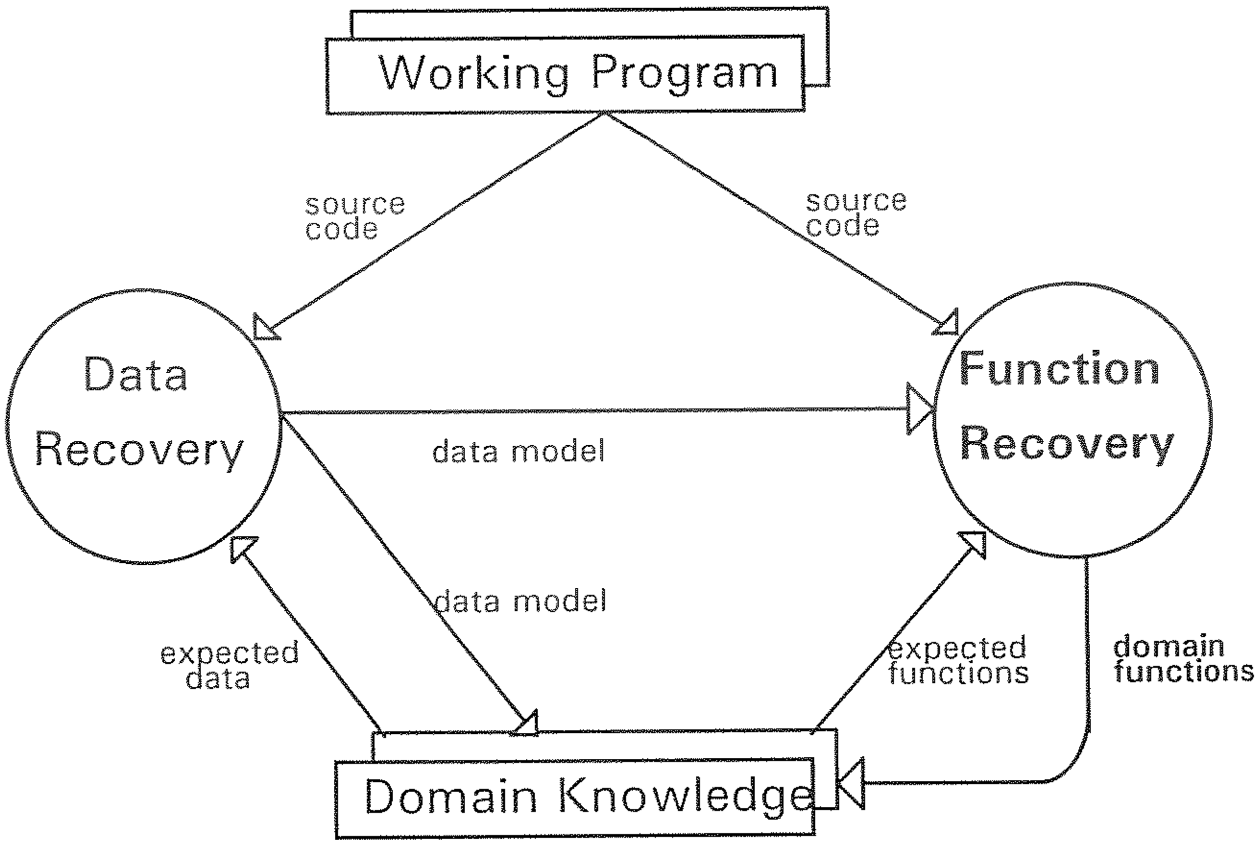


Figure 1. The reverse engineering process

4: Extraction criteria

By using different information sources, a reference data model may be associated to an application system [1]. The reference data model provides a set of expected functions, $F=\{f_1,f_2,\ldots,f_n\}$.

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 to expect 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 yield 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 may be obtained by deleting from the output slice all the statements which also belong to the input slice: $f_i = S(OUT_i) - S(IN_i)$.
6) Package the slice as a separate module.
7) Derive the complement of the extracted slice and package it as a main module which calls the module corresponding to the extracted slice. The interface between the caller and the called modules is made up of the slice input data IN_i and output data OUT_i, as shown in figure 2.

The extracted slice could be too much complex for a single module. Module cohesion should be measured [18], to evaluate if the extracted slice can be refined. For example, if cohesion is sequential [26], the function will contain other subfunctions bound by a composition relation: $f_i = f_{i1}\, f_{i2}\, \ldots\, f_{in}$.

The extraction technique for subfunctions is analogous to that defined above. The subfunction is looked for into the main function which will become its complement.

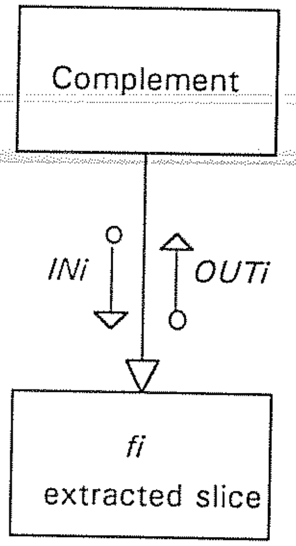


Figure 2. Structure chart after the extraction steps

5: Experimentation

The experimentation was performed on a banking information system. The information system, more than 15 years old, is made up of Cobol programs which run on a Siemens mainframe with BS2000 operating system. The information system has evolved in the years both as functionalities provided and subject areas covered. The numerous maintenance interventions and the frequent turnover of programmers have produced a critical degradation of the initial architecture and a disalignment of the existing documentation. Consequently the change requests are expensive and sometime cannot be satisfied, the backlog increases and the training of new programmers cannot be planned in advance.

The nature of the experimentation was only exploratory in the intentions. The goal was to analyze the application of program slicing to the extraction of domain components for the purpose of characterization with respect to feasibility from the point of view of the reverse engineer.

5.1: The sample program

A program was chosen which manages all the information regarding the relationships with the bank's customers. The program is data-strong, transaction-oriented and poorly modularized. Table 1 summarizes some measurable characteristic.

program AA0000	22034 LOC
Data Division	16834 LOC
Procedure Division	5200 LOC
Dead data	657 LOC
Dead code	221 LOC
Transactions	19
Files	10

Table 1. Program Measures

5.2: The reverse engineer

A junior programmer was selected to conduct the experimentation. He knew the program slicing technique but he had never seen the information system neither he had a previous knowledge of banking problems. His training consisted of:
- 20 days to read a Cobol manual
- 1 day to read a user manual
- 2 half days of interviews with the EDP manager and the maintenance programmer.

5.3: The slicing tool

A commercial tool was chosen, VIA/Renaissance from Viasoft, which is part of a family of maintenance tools called Existing Software Workbench (ESW), running on a MVS-TSO/ISPF platform. For this reason the program was moved from its production environment to an IBM mainframe and some modifications were made to enable a successful recompilation. The tool is made up of three main functional components:
1. Program analysis; you can see the hierarchical relationships inside the program, or navigate along the control and data flow, or follow the execution paths.
2. Code extraction; you can isolate portions of code according to five different criteria, including program slicing.
3. Module generation; you can create modules from the extracted components which can be edited, saved, invoked as Cobol subprograms or independent programs. Complements are automatically treated.

5.4: Results

12 domain functions were successfully located:
- 9 domain functions were expected because the data were attributes of the conceptual data model. Data were cheked before to be stored, or physically transformed and next recorded into the database (derived attributes).
- 3 domain functions were suggested by the maintenance programmer.

For example the "Generate CIN" (Internal Code) function was suggested by the data item NCIN which was present into one of the record layouts. The "Search Customer's Links" function was more difficult to isolate because input and output data were not clearly defined and the resulting slice covered a large piece of code. The "Generate CIC" (Internal Check Code) function was refined by applying the slicing criterion on each of its three inputs corresponding to different legal forms. The "Check Date" function was found three times during different refinements. Two occurences were the same, while the third was a different algorithm which existed in the same program. The programmer who had written the function did not reuse the old one but neither replaced it with a unique copy.

In general, the results improved when i/o data were defined with accuracy (it was not so easy because many variables had not meaningful names) and the source code was restricted. The extracted modules are composed of line numbers which are not always contiguous and the resulting structure provides a more meaningful view of the program structure than usual call graphs can offer. Traceability with the old code is not lost because the extracted components keep track of the source code file and line numbers.

6: Conclusions and future activities

Because the experiment was only exploratory and then it was not strictly controlled and neither conducted on multiple projects, we cannot provide definitive conclusions. However some useful lessons have been learned which can guide future investigation.

L1. Domain-dependent functions explain a small part of the behavior of a data-oriented application. The small number of retrieved functions is partly due to the characteristics of the application system whose greatest effort is devoted to retrieve and store data into the database (MOVE statements were the 41% of the total statements while arithmetic instructions were 2%). So, environment-dependent components must be recovered too for understanding a foreign program.

L2. Programs with a complex control flow due to flags which guide the selection of operations must be previously segmented with some other criterion. When applying program slicing to Cobol sections with logical cohesion, for example, data items which were contemporarily present in the alternative branches caused the resulting slice to cover more than one transaction. It could be useful to identify those flags which drive the control flow and slice the code after having isolated the transactions.

L3. Sometime it is better to apply program slicing in a bottom-up way. Because our approach is top-down, modules at the bottom with a fan-in greater than one are replicated in each extracted slice which depends from their execution. To prevent code duplication, program slicing could first be applied to bottom-level modules with a high fan-in.

L4. When the program is poorly modularized the effect of program slicing is to increase the number of modules by decomposition. However, resulting modules have high internal cohesion and low coupling with other modules.

Future experimentation is necessary to confirm the application of program slicing to function recovery.

- Experiments will be designed using different subjects on different programs, but always using data-oriented applications.
- The extraction criteria will be more automated by customizing the commercial tool or building a specialized prototype.
- Refinements criteria, based on complexity metrics, will be provided to reverse engineers for stopping the slice decomposition.

Acknowledgements

We would like to thank Basica s.p.a. for having provided the tool and the experimentation sample. We are also grateful to Dr. Gregorio Fatone of StarService s.p.a. for the precious collaboration in the experiment.

References

[1] F.Abbattista, F.Lanubile, and G.Visaggio, "Recovering conceptual data models is human-intensive", *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, 1993.

[2] B.Adelson, "Problem solving and the development of abstract categories in programming languages", *Memory & Cognition*, vol.9, 1981, pp.422-433.

[3] P.Antonini, P.Benedusi, G.Cantone, and A.Cimitile, "Maintenance and reverse engineering: low level design documents production and improvement", *Proceedings of IEEE Conference on Software Maintenance*, Austin, Texas, IEEE Computer Society Press, 1987.

[4] R.Brooks, "Towards a theory of the comprehension of computer programs" *International Journal of ManMachine Studies*, vol.18, 1983.

[5] G.Canfora, A.Cimitile, and U.De Carlini, "A logic based approach to reverse engineering tools production", *Proceedings of IEEE Conference on Software Maintenance*, Sorrento, Italy, IEEE Computer Society Press, 1991.

[6] E.J.Chikofsky, and J.H.Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, January 1990.

[7] G.Como, F.Lanubile, and G.Visaggio, "Design recovery of a data-strong application", *Proceedings of the Third International Conference on Software Engineering and Knowledge Engineering*, Skokie, Illinois, 1991, pp.205-212.

[8] 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.

[9] F.Cutillo, F.Lanubile, and G.Visaggio, "Using program slicing for software comprehension", *IEEE Workshop Notes on Program Comprehension*, Orlando, Florida, 1992.

[10] F.Cutillo, P.Fiore, and G.Visaggio, "Identification and extraction of domain independent components in large programs", *Proceedings of the Working Conference on Reverse Engineering*, IEEE Computer Society Press, Baltimora, 1993.

[11] A.Engberts, W.Kozaczynski, and J.Ning, "Concept recognition-based program transformation", Proceedings of IEEE Conference on Software Maintenance, Sorrento, Italy, IEEE Computer Society Press, 1991.

[12] K.B.Gallagher and J.R.Lyle, "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering*, vol.17, no.8, August 1991.

[13] M.T.Harandi, and J.Q.Ning, "Knowledge based program analysis", *IEEE Software*, January 1990.

[14] 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.

[15] 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.

[16] F.Lanubile, P.Maresca, and G.Visaggio, "An environment for the reengineering of Pascal programs", *Proceedings of IEEE Conference on Software*

Maintenance, Sorrento, Italy, IEEE Computer Society Press, 1991.

[17] S.Letovski, and E.Soloway, "Delocalized plans and program comprehension", *IEEE Software*, May 1986.

[18] 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.

[19] C.Rich, and L.M.Wills, "Recognizing a program's design: a graph parsing approach", *IEEE Software*, January 1990.

[20] H.M.Sneed, and G.Jandrasic, "Inverse transformation of software from code to specification", *Proceedings of IEEE Conference on Software Maintenance*, Phoenix, Arizona, IEEE Computer Society Press, 1988.

[21] R.C.Waters, "The Programmer's Apprentice: a session with KBEmacs", *IEEE Transactions on Software Engineering*, November 1985.

[22] R.C.Waters, "Program translation via abstraction and reimplementation", *IEEE Transactions on Software Engineering*, August 1988.

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

[24] M.Weiser, "Programmers uses slices when debugging", *Communications of ACM*, vol.25, no.27, July 1982, pp.446-452.

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

[26] E.Yourdon, and L.L.Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, NY: Yourdon Press, 1978.

Author Index

NOTES

IEEE Computer Society

IEEE Computer Society Press Publications

Monographs: A monograph is an authored book consisting of 100-percent original material.
Tutorials: A tutorial is a collection of original materials prepared by the editors, and reprints of the best articles published in a subject area. Tutorials must contain at least five percent of original material (although we recommend 15 to 20 percent of original material).
Reprint collections: A reprint collection contains reprints (divided into sections) with a preface, table of contents, and section introductions discussing the reprints and why they were selected. Collections contain less than five percent of original material.
Technology series: Each technology series is a brief reprint collection — approximately 126-136 pages and containing 12 to 13 papers, each paper focusing on a subset of a specific discipline, such as networks, architecture, software, or robotics.
Submission of proposals: For guidelines on preparing CS Press books, write the Editorial Director, IEEE Computer Society Press, PO Box 3014, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1264, or telephone (714) 821-8380.

Purpose

The IEEE Computer Society advances the theory and practice of computer science and engineering, promotes the exchange of technical information among 100,000 members worldwide, and provides a wide range of services to members and nonmembers.

Membership

All members receive the acclaimed monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others seriously interested in the computer field.

Publications and Activities

***Computer* magazine:** An authoritative, easy-to-read magazine containing tutorials and in-depth articles on topics across the computer field, plus news, conference reports, book reviews, calendars, calls for papers, interviews, and new products.
Periodicals: The society publishes six magazines and five research transactions. For more details, refer to our membership application or request information as noted above.
Conference proceedings, tutorial texts, and standards documents: The IEEE Computer Society Press publishes more than 100 titles every year.
Standards working groups: Over 100 of these groups produce IEEE standards used throughout the industrial world.
Technical committees: Over 30 TCs publish newsletters, provide interaction with peers in specialty areas, and directly influence standards, conferences, and education.
Conferences/Education: The society holds about 100 conferences each year and sponsors many educational activities, including computing science accreditation.
Chapters: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.