JYVÄSKYLÄ STUDIES IN COMPUTING 4

Jussi Koskinen

Automated Transient Hypertext Support for Software Maintenance

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2000

Editors
Seppo Puuronen
Department of Computer Science and Information Systems, University of Jyväskylä
Kaarina Nieminen
Publishing Unit, University Library of Jyväskylä

ISBN 951-39-0673-6 (nid.), 978-951-39-5482-6 (PDF)
ISSN 1456-5390

Copyright © 2000, by University of Jyväskylä

Jyväskylä University Printing House, Jyväskylä and
ER-Paino Ky, Lievestuore 2000

ABSTRACT

Koskinen, Jussi
Automated Transient Hypertext Support for Software Maintenance
Jyväskylä: University of Jyväskylä, 2000, 98 p. (+included articles)
(Jyväskylä Studies in Computing
ISSN 1456-5390; 4)
ISBN 951-39-0673-6 (nid.), 978-951-39-5482-6 (PDF)
Finnish summary
Diss.

The purpose of the study is to develop and evaluate a hypertext-based approach for legacy software maintenance support. Program text is viewed as transient hypertext, consisting of program parts connected by links enabling fast nonlinear browsing. Transient hypertextual access structures (THASs) are formed automatically to satisfy the situation-dependent information needs of software maintainers. We develop a layered model called HyperSoft for this purpose, implement the approach, and evaluate its hypothesized usefulness empirically. The formation of THASs is based on applying program analysis techniques. The approach is implemented in the HyperSoft system, which is an experimental software maintenance support tool. The implementation of the system is guided by representatives from our industrial partner enterprises. The target language (C) and the implemented THAS set is selected according to the needs of the enterprises. The supported THAS set includes definition references, occurrence lists, call graphs, and program slices. The usefulness of the approach, the system, and the implemented THAS types is evaluated in three different ways: first, by small-scale testing in the partner companies; second, by comparing the HyperSoft's capability to the information needs of software maintainers revealed in a series of earlier empirical studies; and third, by two independent test series. The test series compare information retrieval task performance effects of using HyperSoft and Borland C/C++. The results clearly support our hypothesis regarding the usefulness of the approach.

Keywords: hypertext, software maintenance, CASE (Computer Assisted/ Aided Software Engineering), reverse engineering, program analysis, program comprehension, program slicing

ACM Computing Review Categories

D.2.2. Software Engineering: Tools and Techniques, *Computer-aided software engineering (CASE), User interfaces*
D.2.5. Software Engineering: Testing and Debugging, *Tracing*
D.2.7. Software Engineering: Distribution and Maintenance, *Corrections, Enhancement*
D.3.4. Software: Programming Languages, *Parsing*
E.1. Data structures, *Graphs*
F.3.3. Theory of Computation, *Studies of program constructs*
H.3.3. Information Search and Retrieval, *Retrieval models*
H.5.1. Multimedia Information Systems, *Hypertext navigation and maps*

Author Assistant professor Jussi Koskinen,
Department of Computer Science and Information Systems,
University of Jyväskylä,
P.O. Box 35, FIN-40351, Jyväskylä, Finland.
Email: koskinen@cs.jyu.fi

Supervisors Professor Airi Salminen,
Department of Computer Science and Information Systems,
University of Jyväskylä, Jyväskylä, Finland.

Professor Jukka Paakki,
Department of Computer Science,
University of Helsinki, Helsinki, Finland.

Reviewers Professor Kaisa Sere,
Department of Computer Science,
Åbo Akademi University, Turku, Finland.

Professor Carolyn Watters,
Faculty of Computer Science,
DalTech, Dalhousie University, Halifax, Nova Scotia, Canada.

Opponent Professor Kai Koskimies,
Department of Software Engineering,
University of Tampere, Tampere, Finland.

ACKNOWLEDGMENTS

This research originated out of discussions with Professor Airi Salminen (University of Jyväskylä, Dept. of Computer Science and Information Systems; DCSIS). It was clear from the beginning that transient hypertext support could be useful. Of the decisive importance was the insight of applying transient hypertext to software maintenance. Professor Jukka Paakki (currently at the University of Helsinki, Dept. of Computer Science and at Nokia Research Center) joined the team, and was the central figure in leading the HyperSoft project. Airi and Jukka were my mentors and coauthors. Mika Nieminen MSc. (currently managing director of SupraSoft) was my colleague in the course of implementing the HyperSoft system. Together, we formed the HyperSoft team. Thus I especially wish to thank Airi, Jukka, and Mika. This earlier work has since continued as my PhD project.

I thank the reviewers of the thesis, Professor Kaisa Sere (Åbo Akademi University) and Professor Carolyn Watters (Dalhousie University, Canada) as well as Professor Kai Koskimies (University of Tampere) for their comments and remarks. Series editor, Dr Seppo Puuronen (DCSIS), provided many useful remarks related to the summary part of the thesis. The steering group of the HyperSoft project provided helpful comments during the implementation of the HyperSoft system. The group consisted of the representatives of our partner enterprises: Dr Antero Taivalsaari (senior researcher, Nokia Research Center), Paavo Holopainen (system chief, Novo Group), Jyrki Saarivaara (project manager, Tieto Corporation), and Marita Tolvanen (customer service chief, Tieto Corporation).

The work done on parsers related to the AnaGram™ parser generator by Jerome Holland (Parsifal Software, MA, USA) provided a good basis for the construction of the C parser part of the HyperSoft system. Dr Annaliisa Kankainen and Lecturer Anna-Liisa Lyyra (both from the Univ. of Jyväskylä, Dept. of Statistics) provided helpful support related to the use of statistical methods. I remember the useful discussions I had on program slicing with Mr. Veli-Matti Risku and on ESQL with Mr. Timo Suominen while they were writing their master's theses. Professor Jari Veijalainen (DCSIS) and Professor Markku Sakkinen (DCSIS) both took a positive attitude towards the empirical testing of the HyperSoft system at their courses on software engineering. Dr Steven Kelly (DCSIS) and Lecturer Michael Freeman (Univ. of Jyväskylä, English Dept.) proof-read some of the critical parts of the text. These language revisions have helped to improve the style of the thesis. I also wish to extend my thanks to my long-time friend, Matti Kukkonen Doctor of Law, Lic. of Econ. (Helsinki School of Economics and Business Administration) for innumerable general discussions on science and life during the years leading to this dissertation.

Katriina Byström L.Soc.Sc. (University of Tampere), Professor Erkki Mäkinen (University of Tampere), Professor Norman Wilde (University of West Florida, FL, USA), Professor Anneliese von Mayrhauser (Colorado State University, CO, USA), Dr Terence Parr (MageLang Institute, USA), Dr Harri Oinas-Kukkonen (University of Oulu), Professor Frank Wm. Tompa (University of

Waterloo, Canada), and Dr Eila Kuikka (University of Kuopio) provided useful material and/or comments related to reverse engineering, for which I thank them. I'm grateful to those who participated in the evaluation of the HyperSoft system: 8 persons from the partner enterprises during summer 1995 and summer 1996 and 70 students of computer science at the University of Jyväskylä during fall 1998 and spring 1999. Last, but not least, I thank all my friends, my mother, and my sister, who provided support for the fulfillment of this project.

This work has been funded by the Jyväskylän Kauppalaisseuran Säätiö (1994) and by COMAS (Jyväskylä Graduate School in Computing and Mathematical Sciences, at the University of Jyväskylä, 1997-1999). The HyperSoft project (1994-1996) was funded by TEKES (National Technology Agency of Finland) together with the University of Jyväskylä and our partner enterprises: Nokia Research Center, Novo Group (formerly: KT-Tietokeskus) and TietoEnator (formerly: Tieto Corporation, TT-Kuntajärjestelmät and VTKK). The research related to the HyperSoft project was mainly carried out within the Information Technology Research Institute (TITU/ University of Jyväskylä).

Jyväskylä, Finland. April 2000.

Jussi Koskinen

CONTENTS

1 INTRODUCTION 13

2 SOFTWARE MAINTENANCE: CHARACTERISTICS, PROBLEMS, AND SOLUTIONS 16

2.1 Software and program text 16

2.2 Software maintenance and program comprehension 17

2.2.1 Classifications of maintenance tasks 18

2.2.2 Program comprehension 18

2.2.3 Economic significance 20

2.3 Solutions 20

2.3.1 Algorithmic solutions for program analysis 21

2.3.2 Hypertext and software hypertext systems 23

3 TRANSIENT HYPERTEXT SUPPORT FOR SOFTWARE MAINTENANCE 26

3.1 Research objectives and problems 26

3.2 Principles 28

3.3 The HyperSoft system 28

3.3.1 Static program analyzer 30

3.3.2 Program database 31

3.3.3 THAS generator 31

3.3.4 Generic user interface 32

3.4 Example HyperSoft sessions 33

3.4.1 Call graph example 34

3.4.2 Backward slicing example 39

3.4.3 Forward slicing example 40

3.5 Evaluation of the approach 42

3.5.1 Proposed benefits and probable drawbacks 42

3.5.2 Solutions related to HyperSoft 45

4 OVERVIEW OF THE ARTICLES 50

4.1 "Program Text as Hypertext: Using Program Dependences for Transient Linking" 51

4.2 "HyperSoft: An Environment for Hypertextual Software Maintenance" 52

4.3 "Creating Transient Hypertextual Access Structures for C Programs" 53

4.4 "Automated Hypertext Support for Software Maintenance" 54
4.5 "From Relational Program Dependencies to Hypertextual Access Structures" 55
4.6 "Hypertext Support for Information Needs of Software Maintainers" 57
4.7 "Evaluations of Hypertext Access from C Programs" 58
4.8 About the joint articles and other publications 59
5 DISCUSSION ON RESEARCH DIRECTIONS 60
5.1 Model extensions 61
5.2 Query mechanisms 62
5.3 Technical optimizations 62
5.4 New access structures 63
5.5 Language extensions 64
5.6 View enhancements 64
5.7 Empirical studies 65
CONCLUSION 66
REFERENCES 68
APPENDIX 1 ALGORITHMIC SOLUTIONS FOR SOFTWARE ANALYSIS 91
APPENDIX 2 SURVEYED SOURCES 95
FINNISH SUMMARY 98

LIST OF INCLUDED ARTICLES

I Koskinen, J., Paakki, J. & Salminen, A. 1994a. Program text as hypertext - using program dependences for transient linking. In *Proc. 6th Int. Conf. on Software Engineering and Knowledge Engineering* (*SEKE'94*). Skokie, IL: Knowledge Systems Institute, 209-216.

II Salminen, A., Koskinen, J. & Paakki, J. 1994a. HyperSoft: an environment for hypertextual software maintenance. In B. Magnusson, G. Hedin & S. Minör (Eds.) *Proc. Nordic Workshop on Programming Environment Research* (*NWPER'94*). LU-CS-TR: 94-127. Lund, Sweden: Lund Univ., 25-37.

III Koskinen, J. 1996c. Creating transient hypertextual access structures for C programs. In *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering* (*ICCSSE'96*). Los Alamitos, CA: IEEE Computer Soc., 56-65.

IV Paakki, J., Salminen, A. & Koskinen, J. 1996. Automated hypertext support for software maintenance. *The Computer Journal* 39 (7), 577-597.

V Paakki, J., Koskinen, J. & Salminen, A. 1997. From relational program dependencies to hypertextual access structures. *Nordic Journal of Computing* 4 (1), 3-36.

VI Koskinen, J., Salminen, A. & Paakki, J. 1999. Hypertext support for information needs of software maintainers. Univ. of Jyväskylä, Jyväskylä, Finland. *Computer Science and Information Systems Reports, Working paper* WP-37. Submitted (Dec. 1999) for publication to *IEEE Transactions on Software Engineering*.

VII Koskinen, J. 1999c. Evaluations of hypertext access from C programs. Submitted (July 1999) and conditionally accepted (Jan. 2000) to be published in *Journal of Software Maintenance: Research and Practice*.
(VII') An earlier version of the paper has been published as: Koskinen, J. 1999b. *Empirical Evaluations of Hypertextual Information Access from Program Text*. University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Working paper WP-36.
(VII") The results of the 1st experiment has been published as: Koskinen, J. 1999a. Empirical evaluation of hypertextual information access from program text. In *Proc. 7th Int. Workshop on Program Comprehension* (*IWPC'99*). IEEE Computer Soc., 162-169.

INTRODUCTION AND OVERVIEW

1 INTRODUCTION

The problems of software maintenance are prominent and well-known. Surprising amounts of resources are needed in order to obtain adequate comprehension of the structure and behavior of software systems. Program comprehension is needed in finding information in programs and in modifying them without introducing undesired side-effects. Reverse engineering techniques attack these problems. Reverse engineering comprises the process of identifying the components of a software system and their relations and of creating alternative (often abstracted) representations for the system. Reverse engineering tools are an important research area owing to the considerable costs involved in software maintenance.

This dissertation studies one way of supporting software maintenance via reverse engineering techniques. The name of our approach, model and corresponding implementation is **HyperSoft**. HyperSoft combines the techniques of **hyper**textual information representation and retrieval with those of automated **soft**ware and program analysis (see Figure 1). HyperSoft can be motivated by the possibilities to form hypertext based on program text, by congruence with the central issues of program comprehension theories, by focused support for information needs of practical importance, and by task performance effects. These issues will later be discussed in detail.

Program text is viewed as hypertext in which the division into relevant fragments and dependencies between the fragments is made explicit. Until the present the hypertextual representation scheme and program analysis techniques have been employed in *software hypertext systems* and *reverse-engineering tools*, respectively, and usually separately. The HyperSoft approach is based on the idea that hypertext is formed *automatically*. The hypertextual nodes correspond to the syntactical fragments of the program text and the links to the *program dependencies*[1] characteristic of the programming language.

---

[1] The terms *dependence* (plural: *dependenc(i)es*) and *dependency* (plural: *dependencies*) have the same meaning and both are used in the literature - we mainly utilize the latter form.

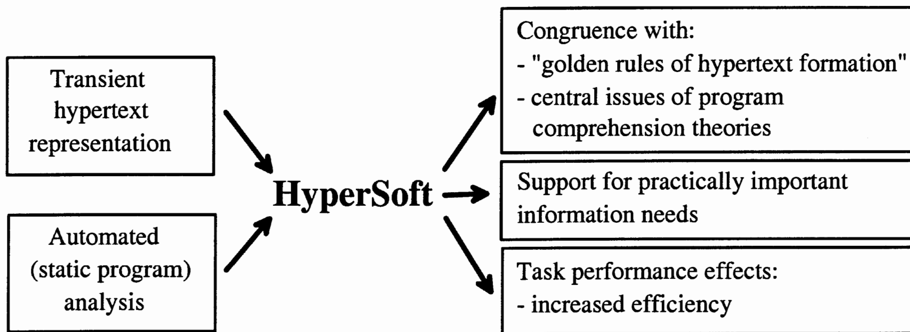


FIGURE 1 The HyperSoft approach

The HyperSoft model is a specialization and extension of a general, abstract, language-independent model for textual databases (Salminen & Watters, 1992). The HyperSoft model separates the syntactic and access structure layers from each other. The *transient hypertextual access structures* (THASs) are composed of those program parts which are linked together on the basis of the existing program dependencies. THASs are *transient*, which means that they are not stored permanently, but are instead generated on user request for the duration of the tool usage session. THASs help software maintainers to focus their attention on those parts of the program which are relevant, *cf.* for example (Mizzaro, 1997; JASIS, 1994), to the current software maintenance situation and to browse through the THASs by following the hypertextual links formed by the support tool. THASs are represented to the user as textual, hypertextual, and graphical views.

The work contains theoretical, constructive, and empirical elements. The implementability and practical applicability of the ideas presented is demonstrated with an experimental software maintenance support tool, the HyperSoft system (Paakki *et al.*, 1996; Koskinen *et al.*, 1997), which was constructed during the HyperSoft project. Thus, the work includes a relatively large constructive part (Koskinen, 1997; Koskinen *et al.*, 1997). HyperSoft was a TEKES (National Technology Agency of Finland) project during 1994-97. Since then, HyperSoft has continued as my PhD project. The HyperSoft project has been guided by an industrial steering group, consisting of representatives of three large Finnish software houses: Nokia, Novo Group and Tieto Corporation. The steering group has participated in the guidance of the project by reviewing the proposed functionality of the HyperSoft system. The implemented THAS set and the supported programming language, C (Kernighan & Ritchie, 1988), were chosen by the steering group.

The work done in relation to HyperSoft is also reported in various papers or reports (Koskinen *et al.*, 1994a; 1994b; Salminen *et al.*, 1994a; 1994b; Koskinen, 1995; Koskinen, 1996a; 1996b; 1996c; Paakki *et al.*, 1996; 1997; Koskinen 1999a; 1999b), in submitted papers (Koskinen *et al.*, 1999; Koskinen 1999c) and in form of a bibliography (Koskinen, 1999d). Moreover, there is the HyperSoft system itself (Koskinen *et al.*, 1997). The technical and user documentation of the

HyperSoft system includes a report (Koskinen, 1997), which contains the design documentation of the back-end parts of the system, a Master's thesis (Nieminen, 1996) (which describes the front-end), and the HyperSoft user manual (Nieminen & Koskinen, 1997). Other works related to HyperSoft include Risku (1995), Tuovinen (1995), Suominen (1997) and Sillanpää (1997).

This part contains a general overview and a summary of the thesis. An updated and extended discussion is provided based on the existing literature as an attempt to bind the results obtained in a potentially fruitful way to the related research and possible extensions of the HyperSoft system (v. 1.0) (Koskinen *et al.*, 1997). The survey part includes efforts made to understand the processes of software maintenance and program comprehension and the support mechanisms of those processes.

First, the application area - software maintenance and program comprehension - is characterized and the branches of solutions and concepts which constitute the background to HyperSoft are briefly surveyed in Chapter 2. There exists a lot of research related to the aspects of reverse engineering and hypertext support for software engineering, but only little or no research directly related to all of the aspects relevant for supporting software maintenance via transient hypertext. The approach is potentially very fruitful. Those readers who prefer more details are directed to the material provided in Appendix 1. The literature survey is focused on the sources represented in Appendix 2. The approach is described in Chapter 3, which presents the research objectives and problems, principles, example sessions and evaluation (benefits, drawbacks, and comparisons to other approaches).

Overviews of the articles forming the main part of the dissertation are given in Chapter 4. A discussion on the scope of the approach and further research directions are provided in Chapter 5. Finally, the conclusions are presented. The original articles are included. It should be noted that this overview part extends the discussion, particularly in relation to the literature survey (Section 2.3), the HyperSoft system (Section 3.3), and research directions (Chapter 5), whereas many of the other important aspects are noted only in passing. The articles provide the detailed level information.

2 SOFTWARE MAINTENANCE: CHARACTERISTICS, PROBLEMS, AND SOLUTIONS

The term *software engineering* - see *e.g.* Sommerville (1996); Pressman (1997) - suggests a discipline resembling that of other engineering fields. However, compared to the established fields of engineering, the area in which certain software engineering techniques can be applied is less well-defined. The same techniques can be applied in numerous areas of application (Glass & Vessey, 1995). This diversity of domains of application introduces many of the problems related to software engineering. Compared to other fields of engineering general libraries of ready-made or reusable components (see Weide *et al.*, 1991) are not so widely applied. Thus, old software in particular consists, in the most part, of non-standard functions and elements, whose understanding and interfacing related to their maintenance and reuse (Prieto-Diaz, 1991) is problematic. A general introduction to the main aspects of the area is provided, for instance, by (Sommerville, 1996/ Sections 24.3 Static analysis tools; pp. 493-496, 25.1 CASE classification; pp. 507-511, 26.3 Testing workbenches; pp. 538-539, 32 Software maintenance; pp. 659-674, and 34.4 Reverse engineering; pp. 711-714).

2.1 Software and program text

Software, written as *program text,* consists of the instructions to the computer and is typically stored in an electronic form. A characteristic feature of program text is that in addition to this *linear structure* it also has a *hierarchic structure* defined by a grammar, and that the instructions follow the rules of the *programming language* used. Modern commercial software systems are typically very large and programmed by various people each with their own level of expertise, programming styles (Straker, 1992) and, possibly, using multiple languages. Moreover, because software describes the *abstract relations* between its parts, the dependencies within it are essentially *invisible* (Brooks, F.P. Jr. 1987). Software entities are characterized by the *concepts* related to the programming

(implementation) and problem (application) domains, *representations* through which these concepts are expressed, and by the fact that the operation of the entities depends on the conditions present at the time of execution. Much of the complexity of software is of an arbitrary nature determined by the human institutions and computer systems to which the interfaces have to conform. These issues are discussed in text books, such as Sommerville (1996).

Program text and some pieces of system documentation are structured text, which has been studied in relation to the structured text databases, for example, by Rossiter *et al.* (1990); Kuikka (1996). Program text often has to serve as a *document* for maintainers, but it differs from other technical documentation (Hopkins & Jernow, 1990) in that it is typically not tailored for that purpose. It is however often annotated with comments and supplemented with other documentation including, for example, the requirements, functional, design and architectural specifications, as well as user manuals (Garg, 1989). But, especially for large, old undocumented *legacy systems* (IS, 1995; 1998; Ning *et al.*, 1994; Bennett, 1995), the source code is the only accurate description. So, in dealing with software systems of this kind, the necessary information needs to be extracted from the source code.

As noted *e.g.* by Lakhotia (1993a), the problems of program comprehension are often overwhelming. There is no single unique *fragmentation* or specific type of representation by which it would be possible to grasp all the aspects of software. It is typical of program text that there are a lot of *interdependencies* between parts belonging to different fragmentations. Unlike the text in printed books or journals, the program text is typically not static and never fully reaches its final form, but typically evolves through numerous changes and modifications (Ramalingam & Reps, 1992). The fact that program text is essentially a complex web of invisible interdependencies generates many of the problems of its maintenance.

2.2 Software maintenance and program comprehension

The program text is originally written during the process of programming. The term *software maintenance* (Hagemeister *et al.*, 1992; von Mayrhauser, 1994) is often used only with reference to making changes to programs after they have been delivered to customers. The importance of software maintenance and program comprehension has been recognized more readily during the last few years, partly because of the potentially great effects of the Y2K problems; see *e.g.* (Newcomb & Scott, 1997; Ragland, 1997; Sharon, 1997; Zvegintzov, 1997; Marcoccia, 1998). There is also a shortage of empirical studies on software maintenance, which has been noted by Haworth *et al.* (1992).

Because even new software has to be changed, the entropy (*i.e.* disorder) of aging software is always increasing. This is the so-called *ripple effect*. Changes made to software based on an insufficient understanding of its structure and behavior make it more fragile. Because there is only a small chance of making

changes correctly on the first attempt, changes need to be designed and their possible negative side-effects minimized. These problems are exacerbated by the fact that the sizes of new applications have tended to increase as new technical solutions and methods have made this feasible.

2.2.1 Classifications of maintenance tasks

Software maintenance tasks are often classified, according to the purpose of the needed program changes, into the large sub-categories represented in Table 1. This classification has appeared - for instance - in the software engineering books of Pressman (1997) and will be applied in Article VI. Note that the list of the included articles appears after the table of contents. Classifications and the meanings of the terms slightly differ in different text-books. Sommerville (1996) uses only the three first categories: corrective, adaptive, and perfective. Software maintenance is often most difficult to accomplish while large changes or enhancements (Jones, 1989) of the adaptive maintenance category in case of insufficient, unupdated or non-existent documentation are made to large legacy systems.

Modifying programs on the basis of inadequate comprehension, then, easily leads to errors and, consequently, to tasks of the corrective maintenance category (Regelson & Anderson, 1994; Duncan & Robson, 1996; Eisenstadt, 1997). Other ways of classifying maintenance activities have also been proposed (Arunachalam & Sasso, 1996). If prototyping (Luqi, 1989; Davis, 1995) is used according to the spiral model of software development (Boehm, 1988a), there may be numerous "maintenance phases".

TABLE 1 The main maintenance task categories and their purposes

Maintenance category	Purpose
Corrective	Diagnosis, localization, and correction of errors.
Adaptive	Interfacing software with a changing environment.
Enhancement or perfective	Additions, enhancements, and modifications made based on changing user needs.
Preventive	Enhancement of future maintainability.

2.2.2 Program comprehension

Problems of software maintenance may be reduced via proper program comprehension. At a general level, *program comprehension* may be defined as a process which aims to enhance the level of knowledge about issues which are important to the fulfillment of programming and maintenance tasks. General, important surveys of program comprehension research issues include (Corbi, 1989; Robson *et al.*, 1991; JSS, 1999) and models for program comprehension have been suggested or surveyed by Brooks (1977; 1983), Vessey (1989), von Mayrhauser and Vans (1995a; 1995c), and Tilley *et al.* (1996).

The comprehension model of von Mayrhauser and Vans (1995a) is supported by a series of empirical studies (von Mayrhauser & Vans, 1995b; 1996; 1997a; 1997b; 1998; von Mayrhauser *et al.*, 1997). Comprehension of specific program components has been studied, for example, by Soloway *et al.* (1983), and Iselin (1988). Program comprehension is based on general cognitive processes, which have been studied by Letovsky (1986) and Iio *et al.* (1997).

More precisely, program comprehension is a process in which the programmer or maintainer interacts with the source code and tries to recreate the *design rationales and decisions* (Rugaber *et al.*, 1990) that the original programmers used while they were writing the program. In order to avoid the introduction of side-effects due to the changes made, program comprehension efforts should precede the making of changes. The relation of program comprehension and software maintenance is such that program comprehension is necessary in order to fulfill software maintenance tasks successfully; see *e.g.* (Visaggio, 1997).

Programs are typically read both in sequential and nonlinear fashion. Moreover, program comprehension is not only a text comprehension process, it is more of a *plan recognition process* (Robertson & Yu, 1990) producing increased knowledge about the original intentions. Because of their limited knowledge, software maintainers constantly have *information needs*, which most often are satisfied by examining the source code. Information needs have earlier mainly been studied on a general, application domain independent level related to information science (Dervin & Nilan, 1986; Kulthau, 1991; Wilson, 1994).

The program comprehension theories referred to above suggest that comprehension is affected by the existence of the following kinds of elements:

- *mental models, i.e.* representations within the mind of the programmer about issues which are important to comprehension efforts (Pennington, 1987; Wiedenbeck & Fix, 1993; Corritore & Wiedenbeck, 1999),
- *chunks* (*or cliches*), *i.e.* meaningful program segments (Vessey, 1987; Rich & Waters, 1988; Hartman, 1991; Burnstein & Roberson, 1997),
- *program*(*ming*) *plans, i.e.* stereotypic action sequences (Soloway & Ehrlich, 1984; Letovsky & Soloway, 1986; Davies, 1990; van Deursen *et al.*, 1997), and
- *beacons, i.e.* easily observable program lines which may serve as starting points for comprehension efforts (Wiedenbeck, 1986; 1991; Gellenbeck & Cook, 1991).

Most notably, the comprehension process is difficult if the original programming plans are *delocalized* (Letovsky & Soloway, 1986; Soloway *et al.*, 1988), meaning that program parts which should be read together are dispersed among other, irrelevant parts. "Reading" program text has been compared, *e.g.* to reading a murder mystery or solving a puzzle in which all the central information is not explicitly presented. Maintainers have to generate the "implicit story" in their mind to achieve their objectives. They thus have to extract from the scattered information a "series of events" that will describe various aspects of the program's operation. The process has also been compared to the tasks faced by a historian, a detective, and a clairvoyant (Corbi, 1989).

The program comprehension theories suggest that programmers use some systematic comprehension strategies (Koenemann & Robertson, 1991), such as

top-down and bottom-up strategies. While employing, for example, the top-down strategy, the maintainer first tries to comprehend the main program, or function, and then descends to the lower levels of the calling hierarchy. While employing an "as-needed strategy", the maintainer uses a variety of strategies in combination depending on the current situation. Program comprehension processes and their relation to the information needs of the software maintainer are further analyzed in von Mayrhauser and Vans (1995b) and in other studies conducted by them (von Mayrhauser & Vans, 1997b; 1998; von Mayrhauser *et al.*, 1997).

2.2.3 Economic significance

Software maintenance is the single most expensive software engineering activity. The effort expended on maintenance is between 65-75% of the total effort targeted to information systems development (Sommerville, 1996, p. 660). Moreover, the proportion of maintenance costs of total costs has increased over the years (Edelstein, 1993), even though new improved design methodologies and programming paradigms, such as object-oriented analysis, design, and programming, have been introduced and have had some positive effects on software maintenance (Mancl & Havanas, 1990; Henry & Humphrey, 1993). Localization of the relevant lines of code (*e.g.* those which need to be changed) is an important class of tasks causing costs, since localizations require both time and other resources.

Edelstein (1993) has estimated that a sum of US $70 billion was used for software maintenance in 1993. On the other hand, Jones (1997) has estimated that the Year 2000 (Y2K) problem alone will potentially cause, worldwide, the astonishingly large total of $1600 billion. The strategic importance of the Y2K problem is also noted by Gunter *et al.* (1996). Thus, reverse engineering methods and tools improving the fulfillment of software maintenance tasks (and especially program comprehension and localization of the relevant program lines) have considerable economic significance.

2.3 Solutions

The problems of software maintenance and program comprehension have been attacked in various ways. The development of reverse engineering tools (Rock-Evans & Hales, 1992; CACM, 1994) is motivated by their potential to increase productivity. Since the HyperSoft approach combines the notions of hypertext and reverse engineering (program analysis) techniques, mainly these approaches are briefly surveyed here. Since HyperSoft is based on the idea of automated support, methods, techniques, and tools especially enabling that purpose are noted and references given to the most important sources. A wider bibliography is available (Koskinen, 1999d).

Reverse engineering, program analysis, and software hypertext systems all represent various aspects of support which can be found in (integrated)

CASE (Computer Aided Software Engineering) tools and environments (Misra, 1990; Fuggetta, 1993; Sommerville, 1996/ Section 25, pp. 505-544). Our research can be considered as belonging to CASE research, although the focus is on supporting maintenance (via reverse-engineering; lower CASE tools), instead of supporting actual systems development (forward engineering; upper CASE tools).

2.3.1 Algorithmic solutions for program analysis

Reverse engineering means the process of identifying a system's components and their interrelations and of creating representations of the system in another form or at a higher level of abstraction (Chikofsky & Cross, 1990; Cross *et al.*, 1992; IJSEKE, 1994). Thus, in our context, the representations created are based on automated transformations. Reverse engineering is a reverse process of *forward engineering* (ordinary systems development) in which the activity proceeds from various specifications to analysis and design modelling, and finally to implementation (coding). The software source code is usually available as an input to the reverse engineering process and the process is normally part of software re-engineering (Sommerville/ Section 3.4.4, pp. 711-713). Established methods of *program analysis* (Welsh & McKeag, 1980; Aho *et al.*, 1986) are typically used in producing the higher-level representations related to reverse engineering. Program analysis may be static or dynamic. HyperSoft applies static analysis; *cf.* for example, Wichman *et al.*, (1995) and von Mayrhauser and Lang (1999). Dynamic analysis relies on information which is available only at the run-time of the maintained program.

Since HyperSoft is based on the automatic generation of THASs, it is useful to list here some of the algorithms and solutions to various reverse engineering and software analysis problems (in automatically analysing either the source code or the related documentation, applicable in generating various THAS types. From the view point of HyperSoft, these solutions can be grouped according to whether they mainly support the identification of program components (nodes) or their interrelations (dependencies). These algorithms correspond to the possible content of the 'Algorithm' attribute of dependencies as represented within our classification of program dependencies (see Article V/ Figure 2).

Relevant solutions for component identification include techniques for program decomposition, various metrics and techniques which can be used to identify interesting program parts, and text analysis and concept extraction techniques which may be used as applied to the system documentation or comments. Techniques related mainly to dependencies include general data flow analysis, its variants, and techniques for producing trees and graphs which can be used in storing program information. A classification of algorithmic solutions is provided in Appendix 1. Here we will refer, as examples, to two of the categories - program dependency analysis and program slicing (Appendix 1; parts 5 and 6).

Program dependency analysis

Many of the reverse engineering tools available represent the target program to the user on the basis of program dependencies, which can be automatically recognized from the source code. General models, classifications or characterizations for program dependencies have been proposed (Perry, 1987; Yau & Tsai, 1987; Podgurski & Clarke, 1990; Paakki *et al.*, 1997). Some tools and techniques (Moriconi & Hare, 1986; Gopal, 1991; Livadas & Roy, 1992; Muller *et al.*, 1992; Baratta-Perez *et al.*, 1994; Linos *et al.*, 1994) are based on program dependencies. Control and data flow are the main examples of program dependencies analyzed in reverse-engineering and maintenance tools.

Since the main target language of our implementation is ANSI-C, it is relevant also to view here the existing research done on C program analysis. A formal method for reverse engineering C programs, including formal definition of the semantics of the language is provided by Gannod and Cheng (1996). The side-effect and modification analysis of C programs have been studied by Yur *et al.* (1997). C has many special features affecting maintainability, which are discussed in some detail, for example, by Darnell and Margolis (1991). The hard problems of analysing C programs - including handling of array and pointer variables and unstructured flow of control - have been attacked by Jiang *et al.* (1991). An environment for recognizing architectural cliches for C programs based on abstract syntax trees has been developed by Fiutem *et al.* (1996).

Program slicing

Program slicing, as coined by Weiser (1982; 1984), can be defined as the extraction of relevant statements from the source programs. Some definitions of slicing require that the slice is an executable subset of the program. Empirical experiments (Weiser, 1982; Weiser & Lyle, 1986) suggest that slices are especially useful in debugging. General surveys of program slicing are provided by Binkley and Gallagher (1996) and Harman and Gallagher (1998). Program slicing techniques are surveyed by Tip (1995), compared by Kamkar (1995), and program slicing tools are compared by Hoffner *et al.* (1995). IST (1998) is a recent special issue on program slicing.

Slicing helps the maintainer to focus on program parts which are somehow relevant to a certain maintenance situation. The focus of interest is indicated by the *slicing criterion*, which typically is a variable occurrence within the program text. Slices are formed on the basis of the analysis of the control and data flows of the program. Backward slicing; see *e.g.* (Kamkar, 1993), reveals the program parts which may affect the slicing criterion, whereas forward slicing reveals the parts that may be affected by the slicing criterion. Hence, backward slicing is typically used in debugging (Kamkar, 1998), whereas forward slicing may be used, for example, in impact analysis; see *e.g.* (Turver & Munro, 1994; Queille *et al.*, 1994; Ajila, 1995; Fyson & Boldyreff, 1998). Many articles have been written on impact analysis, of which some of the most important are reproduced by Bohner and Arnold (1996).

Program slicing may also be used as an aid in regression testing (Harman & Danicic, 1995; Gupta *et al.*, 1996; Binkley, 1998). The slices may be formed on the basis either of static (Choi & Ferrante, 1994) or of dynamic analysis (Korel & Rilling, 1998) of the program. Variants which are related to slicing include *parametric slicing* (Field *et al.*, 1995), *dicing* (Chen, T. & Cheung, 1993; Samadzadeh & Wichaipanitch, 1993), *generalized slicing* (Sloane & Holdsworth, 1996), *sliving* (Gupta, 1997), and *chopping* (Reps & Rosay, 1995). These special techniques can be used in avoiding some of the problems related to traditional slicing.

Efficiency problems related to program slicing are discussed, for instance, by Reps *et al.* (1994). The time-efficiency problems of slicing can be alleviated by storing the needed information into program dependency graphs or into some other permanent structures. Program dependency graphs were introduced by Ottenstein and Ottenstein (1984) and currently there exist many variants (Horwitz & Reps, 1992; Harrold & Malloy, 1993; Kinloch & Munro, 1993; 1994). HyperSoft relies on the traditional way of producing the slices via iterative solving of data flow equations, as in (Weiser, 1982).

Slicing tools include C-Debug (Samadzadeh & Wichaipanitch, 1993), SLICE (Venkatesh, 1995) and those reported by Beck and Eichmann (1993) and Hoffner *et al.* (1995). Some of the tools employ static and some dynamic analysis. The intended application areas vary from debugging to program integration. Problematic areas include the analysis of pointers (Fiutem *et al.*, 1999) and the analysis of unstructured programs (Choi & Ferrante, 1994; Harman & Danicic, 1998). Most of these problems of slicing from the view-point of C programs are noted by Samadzadeh and Wichaipanitch (1993).

2.3.2 Hypertext and software hypertext systems

Hypertext is surveyed by Conklin (1987). Nowadays, hypertext is a very widely applied technique. Hypertext is text with nonlinear browsing capabilities, consisting of text fragments called nodes and links connecting these nodes. Within hypermedia systems the nodes may also contain non-textual information.

The usefulness of hypertext is often motivated by asserting that it complements the more traditional information retrieval based on search and querying (see *e.g.* Belkin & Croft, 1992) with local navigation (Nielsen, 1990; Rivlin *et al.*, 1994) based on the linkage that it provides and by which it generates an open, exploratory information space (Nielsen, 1989). One of the central problems in forming hypertext is that it is often not clear what the most useful fragmentation and linkage would be. An example of this is the Oxford English Dictionary project (Raymond & Tompa, 1988).

Empirical studies

A qualitative synthetic review of quantitative experimental studies on the use of hypertext is provided by Chen, C. and Rada (1996). The areas of application of the target hypertext systems are varied. On the basis of an analysis of 23 studies they made the following main observations: generally, hypertext

appears to enhance performance (although there is wide variation among the studies, partly because of the different "benchmarks", different levels of sophistication of the features provided and different kinds of experimental designs); hypertext appears to benefit the users more in the case of open tasks (which demand, for example, browsing and are typically more complex than closed ones, for example, simple searches); the effect of cognitive styles appears to be small; and the effect of spatial abilities appears to be great. Users clearly benefit from the graphical overview maps.

The great importance of providing an overview map over large hypertext structures has also been noted, for example, in the empirical experiments by Monk *et al.* (1988), related to the browsing of literate programs. They compared hypertextual browsing, scrolling (the text of documents is shown sequentially within a single window) and folding (elision; section titles are shown and by clicking them their text is shown). They found tentative evidence that hypertext alone as compared to scrolling is a less efficient way to perform program comprehension tasks. Comparisons of searching and hypertext browsing also exist (Rada & Murphy, 1992; Qiu, 1993), but not with software engineering as an area of hypertext application.

Methods of hypertext formation

Since one of the main principles of HyperSoft is the automatic formation of hypertext, the efforts of forming hypertext automatically are of interest here. In most cases the (document-based) hypertext is constructed manually or semi-automatically; see Carmel *et al.* (1989); Rada *et al.* (1992). The projects of forming hypertext automatically on the basis of information retrieval techniques are surveyed in Agosti *et al.* (1997). Efforts of automatic hypertext formation (Agosti *et al.*, 1996; Agosti & Allan, 1997; Fraisse, 1997; French *et al.*, 1997; Tebbutt, 1999) typically, apply statistical, document analysis or clustering techniques to link parts of the documents together. Automatic formation includes automated link typing (Allan, 1996; Cleary & Bareiss, 1996). Many of the efforts (Rada 1992; Agosti *et al.*, 1995; Allan 1995; 1997) do not specifically address or emphasize the problems of software engineering as a target area of support.

Hypertext models

Frisse and Cousins (1992) have compared three representative models of hypertext: Dexter (Grønbaek *et al.*, 1994), IBIS (Conklin & Begeman, 1989), and Trellis (Stotts & Furuta, 1989). In addition, there is the GHMI model represented by Wan and Bieber (1996), which is based on Dexter and extends its storage layer. These models do not emphasize the idea of automatic hypertext creation; rather, the emphasis is on the notion that hypertext is authored by people. The purpose of developing Dexter has been to capture the important abstractions found in a wide range of current and possible future hypertext systems. Within HyperSoft, we instead, attempt to capture the important abstractions found in program text and combine them with the abstractions concerning dynamic, transient hypertext. Some information models (Shepherd *et al.*, 1990; Watters &

Shepherd, 1990; Tague *et al.*, 1991; Salminen *et al.*, 1995), support the idea of automatically created (dynamic) transient hypertext, which can be used as a basis for our efforts. HyperSoft and Dexter will be compared in Article IV.

Software hypertext systems

Software hypertext systems support program development and maintenance activities by allowing users to create links between the source code, related documentation or their internal components (some cases, the links may by produced automatically), thus *e.g.* enabling redocumentation (Fletton & Munro, 1988; Younger & Bennett, 1993) of legacy systems. The impact of hypertext in CASE environments is discussed by Kerola and Oinas-Kukkonen (1992), and hypertext in software development environments by Ziv and Osterweil (1995). After the links have been formed, they can be used as an aid in program comprehension efforts. These sorts of links are generally considered as important aids in system development, *cf.* (Kerola & Oinas-Kukkonen, 1992).

There are innumerable commercial reverse-engineering tools. Nowadays, some sort of hypertextual navigation has also found its way into some of them. Typical structures are various cross-references. The most important of such systems for C include SHriMP (Storey & Muller, 1995), RIGI (Wong, 1996), HindSight (IntegriSoft, 2000), Cygnus Source Navigator (Red Hat, 2000), Discover (SET, 2000), Imagix (2000), Logiscope (Verilog, 2000) and Sniff+ (TakeFive, 2000). Since these are commercial systems, the modelling aspect of hypertext is not their focus area. Since hypertext features are evidently becoming more frequently provided, the importance of having a well-thought out approach for their application and a well-grounded model for the representation of the hypertext structures is emphasized.

Systems having some important similarities with HyperSoft (and which will be described in more detail in Section 3.5.2) include DynamicDesign (Bigelow & Riley, 1987; Bigelow, 1988), HyperCASE (Cybulski & Reed, 1992), ISHYS (Garg, 1989; Garg & Scacchi, 1989), and DIF (Garg & Scacchi, 1990). There are also many other systems including those reported by Østerbye (1995), Oinas-Kukkonen (1997a; 1997b) and Rajlich and Varadarajan (1999), which provide mechanisms for creating hypertext for the purposes of forward engineering. Although the capabilities of the above-cited systems are clearly useful in providing associations between the source code and the documentation (which especially is an important part of creating program comprehension based on revealing the connections between technical and application domains), their benefits in relation to the maintenance of legacy systems are mainly confined to documenting these systems. This is because most of them heavily rely on manual hypertext formation, and for legacy systems such a pre-defined structure is not available. The systems which have most important similarities with HyperSoft are Whorf (Brade *et al.*, 1994) and HyperPro (Østerbye & Nørmark, 1993; 1994; Nørmark & Østerbye, 1994; 1995). These systems provide support for viewing program text as hypertext.

3 TRANSIENT HYPERTEXT SUPPORT FOR SOFTWARE MAINTENANCE

In this chapter we will describe our approach. First, we will describe the research objectives and research problems (Section 3.1). Then we will represent the principles (Section 3.2), describe the HyperSoft system, which is the implementation of our approach (Section 3.3), and provide example sessions of the HyperSoft system in practice (Section 3.4). Finally, we will evaluate the approach from the theoretical point of view by proposing the potential benefits and probable drawbacks on the basis of our experience, the results of the empirical evaluations, and comparisons with other tools sharing similar features (Section 3.5).

3.1 Research objectives and problems

The objectives of this study consist of the following: investigation of the possibilities of viewing program text as transient hypertext in order to facilitate software maintenance and program comprehension support, development of an approach for these ends, implementation of the approach in the form of a tool, and evaluation of both the tool and the approach.

By transient hypertext, we mean hypertext which is formed automatically and which exists for only a relatively short period of time. The user is provided with transient hypertextual access structures over the subject software system. Although hypertext has been applied in the software engineering context to a relatively large extent (see Section 2.3.2), source code and its internal structures are seldom considered as hypertext. Because program text follows well the so-called "*golden rules of hypertext formation*" (Shneiderman, 1989), it seems to be reasonable and potentially useful to view program text as hypertext. The golden rules assert that it is possible to form hypertext, if:

- there exists a large body of information organized into numerous fragments,

- the fragments relate to each other, and
- the user needs only a small fraction at a time.

Program text clearly satisfies these conditions. Moreover, hypertext seems to be a natural way in which to represent program text, because the programmer typically browses the text in various nonlinear ways while trying to comprehend it. Hypertext as a way of representing information within software development environments has also been motivated by Ziv and Osterweil (1995). Hypertext is usually formed manually, but in the case of program text automatic formation is also quite possible. Because the automatic formation of transient hypertext will eliminate the need for elaborate manual linking and ensures the currency of the hypertextual structures even if the software changes, we have chosen to investigate that particular strategy.

Our research aims to achieve the following objectives.

1) Analyze whether and how program text can be viewed as transient hypertext (Article I).
2) Determine what kind of model is suitable for representing program text as transient hypertext (especially Articles I, IV).
3) Determine the proper form (and the related fragmentation of the program text in case) of some of the important THAS types (Articles III, IV, V, VI).
4) Determine the nature of the program dependencies which can be used as a basis on which to form the different hypertextual link types (Article V).
5) Determine the necessary static information and the convenient form of storing that information in order to support some of the most prominent THAS types (Articles II, III, VI).
6) Discuss how to deal with the possibly large amount of static information needed (Sections 3.5.1, 5.3, Article III).
7) Analyze what kinds of THAS types can be formed automatically (especially Section 2.3.1, Appendix 1, Articles I, V).
8) Discuss how THASs can be utilized in software maintenance (especially Section 3.4, Articles I, V, VI).
9) Determine the nature and technical architecture of a software maintenance support environment (the HyperSoft system) in which program text is represented via THASs (especially Section 3.3, Articles II, III, IV, and the report; Koskinen, 1997).
10) Analyze what kind of THAS types can be used to satisfy the typical information needs of software maintainers (mainly Article VI).
11) Determine the effects of the THAS-based approach on the information retrieval performance of software maintainers (Article VII).
12) Gather information about the subjective notions of users on THAS-based maintenance support (Articles IV, VII).

Because there were no pre-existing models tailored to view program text as automatically formed transient hypertext, we have developed one such general model (Article I). This modelling has been one of the main aims. The model has been refined and extended during the research process. Since our model contains the grammar of the supported programming language, the hypertextual nodes may correspond to any of the syntactical structures of the program. In practice, however, only some of the node types and the structures composed

from them are useful. The implemented HyperSoft system makes it possible to empirically evaluate the relative usefulness of the different THAS types.

3.2 Principles

We provide a model as a basis for the systematic hypertext-based support in software environments. In the HyperSoft model a THAS is modeled as a directed graph, that is, as a pair (N, L) where N is a set of nodes and L is a set of ordered node pairs called links. Typically, the nodes are parts of specific syntactical types and the link types correspond to program dependencies. The links are formed to enable unlinear text browsing. We have described and classified the program dependencies potentially applicable for creating links in Paakki *et al.* (1997). A complete description of the HyperSoft model will be provided in Article IV.

It is characteristic of the HyperSoft approach that the access structures are transient (temporary), they are generated automatically, the source code is the main input, and the maintenance support of legacy systems is the main target area. The approach describes a hypertext support environment by four layers - source code, syntactic structure, access structure, and user interface - which are explained, for example, in Article IV. The source code layer deals with linear text representations in files and related operations (retrieval and modification). The syntactic structure layer deals with parse trees representing the information needed by the support environment and related operations. The access structure layer deals with THASs and THAS operations. And, finally, the interface layer is related to text and THAS representations as conveyed to the user and interaction between the user and the support environment.

The syntactic parts of program text serve as the basis for forming the hypertext nodes, program dependencies serve as the basis for forming the links, and both the nodes and the links are generated by automatic analysis. It is important within the appoach to be able to focus on relevant program parts and important dependencies. At the interface layer, graphical representations and abstract views are used to deal with the disorientation and cognitive overhead problems (Wright, 1991) often associated with hypertext systems.

3.3 The HyperSoft system

The HyperSoft system (Koskinen *et al.*, 1997) is based on our approach. It is described in Articles II, III, and IV and in Koskinen (1997). It can be characterized as a reverse-engineering tool. HyperSoft supports the ANSI-C language (Kernighan & Ritchie, 1988; Ritchie, 1993) and embedded SQL (Date, 1987). HyperSoft runs under Microsoft Windows™ 3.1/95/NT. HyperSoft supports the comprehension and maintenance processes by providing various THAS types

and view types. THASs are formed automatically by the tool (based on static analysis). The academic objectives of the project have been attained. The set main requirement has been the implementation of a tool to support the maintenance of large software systems written in the C language by providing capabilities to view these programs unlinearly via automatically generated hypertext in accordance with typical maintenance situations. The requirements are detailed in Koskinen (1997/ Section 1.2).

The architecture of the system is represented in Figure 2. The architecture corresponds to our layered HyperSoft model. The main components of HyperSoft are: 1) *analyzer* (static program analyzer), 2) *generator* (THAS generator), 3) *program database*, 4) *interface* (generic user interface), and 5) *editor* (HyperSoft is currently integrated with PFE; Programmer's File Editor). The analyzer corresponds to the syntactic structure layer, generator to the access structure layer, and interface to the interface layer. The source program collection contains the source files for which the program database is generated. The program database is a repository component storing the information passed through other components. Components 1), 2) and 3) belong to the back-end of the system and components 4) and 5) to the front-end. Users interact with the system by using the front-end components. The back-end components are needed in generating THASs. Detailed design and abstract implementation descriptions of the back-end components are included in (Koskinen, 1997).

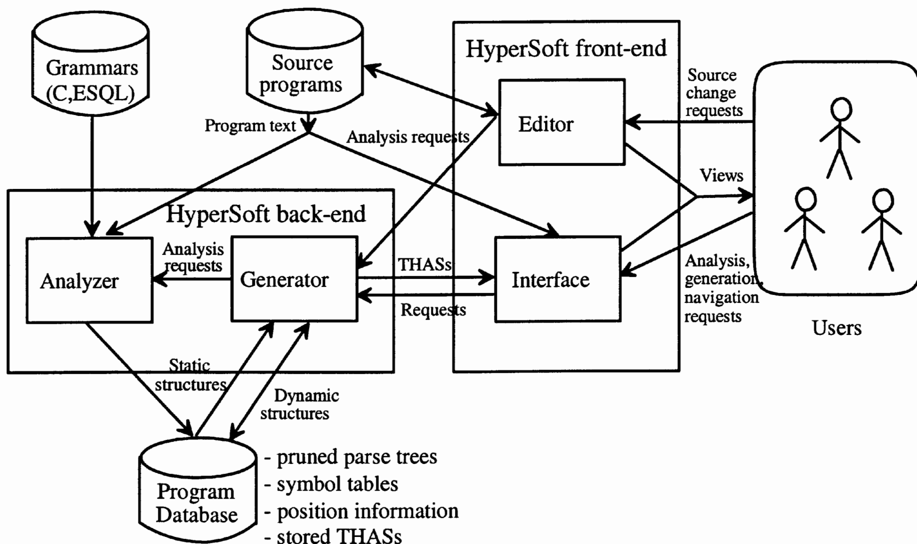


FIGURE 2 The general architecture of the HyperSoft system

HyperSoft consists of two programs: *Analyzer* (component 1) and *HyperGenerator* (components 2 and 4). Database (component 3) is used by both programs. HyperGenerator consists of two parts: the generator is written by the author and the interface by Nieminen (1996). The analyzer analyzes the original source programs and forms the program database, The generator then uses that information to form THASs which are represented to the user as hypertext through the interface-component. The size of the system (excluding PFE) is about 32,000 LOC (lines of code)[2]. The general order of forming and representing the necessary data structures is represented in Table 2. First, preprocessing is performed including macro expansions. Then, parse tree is formed for each module and compressed to spare memory. Next, symbol tables are formed based on parse tree traversals and necessary cross-references are made. THAS generation is initiated by the generator. Finally, a THAS is represented to the user via the interface.

TABLE 2 Overall schemata of the order of the main HyperSoft functionalities

#	Phase
1	Preprocessing, parsing, and parse tree formation.
2	Parse tree abstraction and compression.
3	Creation of symbol tables, based on the preorder traversal of the already formed parse trees.
4	THAS generation, based on the static program database produced in the phases 1, 2, and 3.
5	THAS representation to the user via a generic, graphical user interface providing text and navigation views.

3.3.1 Static program analyzer

The analyzer has two parts: first one (Koskinen, 1997) supports ANSI-C and the other one (Suominen, 1997) supports ESQL (Embedded Structured Query Language). The analyzer creates the static parts of the program database for the source programs that need to be (re)analyzed. This is a preliminary action preceeding the generation of the THASs. The analyzer also provides the necessary parse tree and symbol table representations and operations. The analyzer is built using the AnaGram™ metacompiler (Parsifal, 1993) and the C Macro Preprocessor Package (*CMPP*) delivered with AnaGram. HyperSoft extensions are built on top of the *CMPP*. The formation of the program database (see Section 3.3.2) is based on static analysis (*cf.* Section 2.3.1). The analyzer is implemented as a DOS-program and supports the analysis of syntactically correct ANSI-C programs[3].

---

[2] The analyzer is about 4,000 LOC and the generator about 5,000 LOC.

[3] The analysis is subject to some limitations, which are detailed in (Koskinen, 1997).

AnaGram metacompiler

The AnaGram metacompiler (Parsifal, 1993) has been used to reduce the work needed to code the analyzer. AnaGram is made by Parsifal Software and it contains an *LALR(1)* parser generator; see (Aho & Johnson, 1974), which creates a table-driven parser from a grammar written in a variant of BNF (Backus-Naur Form). Compared to hand-made parsers, parsers generated by AnaGram are more readable and thus more maintainable, yet not significantly slower.

C Macro Preprocessor Package

The C Macro Preprocessor (*CMPP*) delivered with the AnaGram metacompiler includes the syntax descriptions for C parsers in the Kernighan and Ritchie (1988, Section A13, pp. 234-239) form. The syntax (of programming languages) is defined for AnaGram using a notation similar to BNF. The same notation can be used both in the parser and in the lexer components of the system being implemented. Components of the *CMPP* that are used in HyperSoft are described in some detail in Article III. HyperSoft handles macros correctly. In cases where the expanded macro text contains a symbol relevant for the current THAS, the corresponding macro label will be included in the THAS which is currently under construction. HyperSoft also shows the user those lines which are not within the current (conditional) compilation with special highlighting, which is useful feature since conditional compilation is heavily used in the C language.

3.3.2 Program database

The program database (PDB) is implemented as a set of DOS files, and it is created by the analyzer component. There exist a parse tree, a local symbol table, and static occurrence list files for each C (or ESQL) source or header file within the user-defined project. Moreover, there exists a global symbol table which gathers information about the symbols used in multiple files. The PDB files are changed or deleted only when the corresponding source files are modified. In that sense, the program database is permanent in contrast to the THASs. Since all the intermodular information is gathered into the global symbol table, there is no need for the time-consuming process of checking out the local PDB elements in the case of source modifications. This is in contrast with conventional compilers which fetch the object files into main memory during the linking phase.

3.3.3 THAS generator

THASs are formed by the generator component on the basis of various program analysis techniques. The generator uses the static information stored in the program database during the execution of the analyzer and passes information about the THASs so-formed to the interface-component. All of the THASs are transient/dynamic in a sense that the user specifies the criteria for generation during a HyperSoft session. The life-cycle of the THASs is such that they are

created on user request and are removed permanently when the session ends. The generator is a part of the HyperGenerator, which is a Microsoft Windows program.

HyperGenerator supports partial "multiprocessing". Within HyperSoft, it is possible to move inside the text and map windows during THAS generations. This is an especially useful capability when forming large THASs. It is also possible to initiate multiple THAS generations which will then be processed sequentially. HyperSoft is linked with the DBwin - a simple PD program - which is used to show various items of status information and other messages to the user during THAS generation in a separate window. Status information is shown in order to give the maintainer an estimate of the time that the THAS generation will take. The use of HyperSoft, however, does not require DBwin.

The generation of a THAS is initiated from the interface component by sending a request to the generator's main function, which in turn calls the adequate THAS generation functions. THAS formation constitutes of traversals of occurrence lists and parse trees. When forming the occurrence list, it is only necessary to traverse the static occurrence list of the selected symbol. Calling dependency formations require both finding the function occurrences in the occurrence lists and finding the function calls within a certain function body based on the parse tree traversal (in forward call graphs). Slicing requires extensive and complete traversals of the relevant parse (sub) trees.

The HyperSoft system currently supports the following THAS types:

1) Definition references (for variables, functions, and user-defined type names), providing a link to the program part where the relevant component is defined.
2) Occurrence lists (for variables, functions, and user-defined type names), providing a chained list which can be used to check the symbol occurrences.
3) Instance lists (for syntactical constructs), providing a list of components of a specified syntactical type, such as declarations or jump statements.
4) Forward calling dependency structures (complete, "traditional" call graph and calling-level-wise partial variants).
5) Backward calling dependency structures (complete and calling-level-wise), showing the functions (and places of their implementation) which a function is called from.
6) Intraprocedural backward slices, containing information about the statements within a function that may have effect on the value of a variable in a specified program part.
7) Interprocedural forward slices (complete and calling-level-wise), containing information about statements that may be affected by the value of a variable in a specified program part.

3.3.4 Generic user interface

Graphical representation of THASs and interaction with the user is a necessary requirement in HyperSoft. The generator and the (generic user) interface of the HyperSoft system are dependent on each other, since the interface would be

useless without information about the THASs, and the generator would be practically useless without a graphical way of representing the THASs. The interface has been implemented by Nieminen (1996).

The MVC (Model-View-Controller) model (Krasner & Pope, 1988) has been used as the underlying architectural model in HyperSoft. According to Booch *et al.* (1999) the responsibilities of the layers are as follows: the Model layer manages the state of the model (in this case THASs); the View layer renders the model on the screen, manages movement and resizing of the view and intercepts user events; and the Controller synchronizes changes in the model and its views. HyperSoft provides various views for the program text and for the THASs formed. The clear importance of (overview) maps has been reported, for instance, by Chen, C. and Rada (1996), and McDonald and Stevenson (1998). Examples of implemented views are gathered in Figure 8. The views are linked to the program text such that from within them a user can directly move to the appropriate program part. Thus, these links are hypertextual/ hypermedial links between different representations of the same objects. The views include the following.

1) Project file window, listing the modules which are related to the current project and providing a way to move focus to their active element (at first, to the beginning of the file).
2) Structured map view, for hierarchic examination of a THAS, showing the modules, functions and the nodes within a THAS as embedded in the program text. There is also a special structured map view for the integrated editor (PFE), supporting code modifications in an integrated fashion.
3) Function dependency view, showing graphically the functions and the dependencies between them within a THAS.
4) Module dependency view, showing graphically the active modules (which are files in case of C) and relations between them within a THAS.
5) Miniature view, showing the code in a tiny font, for a quick overview of the dispersion of the nodes of a THAS within the program text.

The hypertext nodes are highlighted in different colors. The color of nodes represents the number of originating links (0, 1, more) or their target (whether the dependency exists within or between functions or modules). Hypertext links are optionally shown graphically on top of the program text.

3.4 Example HyperSoft sessions

HyperSoft runs under Microsoft Windows. The tool provides the user with several different views on program text with hypertextual navigation capabilities within the THASs generated by the tool. We will provide three examples of using non-trivial THAS types - call graphs and program slices. The use of the HyperSoft system is described in more detail in the user manual (Nieminen & Koskinen, 1997), and within the documentation distributed with the HyperSoft system (Koskinen *et al.*, 1997).

3.4.1 Call graph example

In this session, the user (unfamiliar with the program) is interested in obtaining a general overview of a chess program and more detailed knowledge of how individual chess moves are handled by it. The example project (Koskinen, 1993) consists of 7 files and contains about 2,700 lines of code. This is the program which has been used as a sample in Article VII, in the empirical evaluation of the HyperSoft system. When starting HyperSoft the user defines the program files from which the program database is to be formed. These files are included in the project file. The program analyzer then forms the static program database for these files. Figure 3 shows the contents of the project file window, a list of the files from which the database is made, and status information on the formation of the static program database. The name of the file currently being analyzed is given in the window. The figure shows the situation when all 7 files of the chess program have been analyzed.

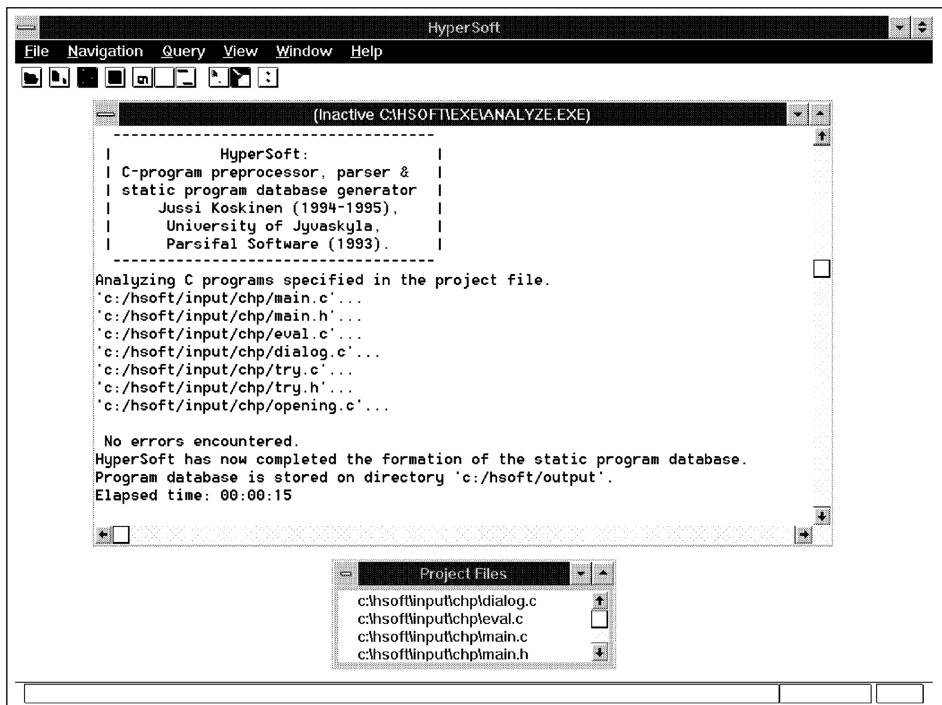


FIGURE 3 The project file window and the formation of the static program database

The user may open multiple program files, views, and dependency diagrams simultaneously on screen within a single HyperSoft session. The system can also be used "simultaneously" (in batch mode) with other Microsoft Windows applications (*e.g.* with an editor or a compiler). In Figure 4 two of the files, `main.c` and `main.h`, have been opened, the program text being shown on the windows. The user may browse the source files and select the sizes of the windows and the fonts used. Within C, a program starts from the function called

main, which typically is located at the beginning of the module main.c. The left pane of Figure 4 shows the beginning of the function main, which is supplemented with general comments, serving as a general overview of the purpose of the program and as a starting point (beacon) for further comprehension efforts.

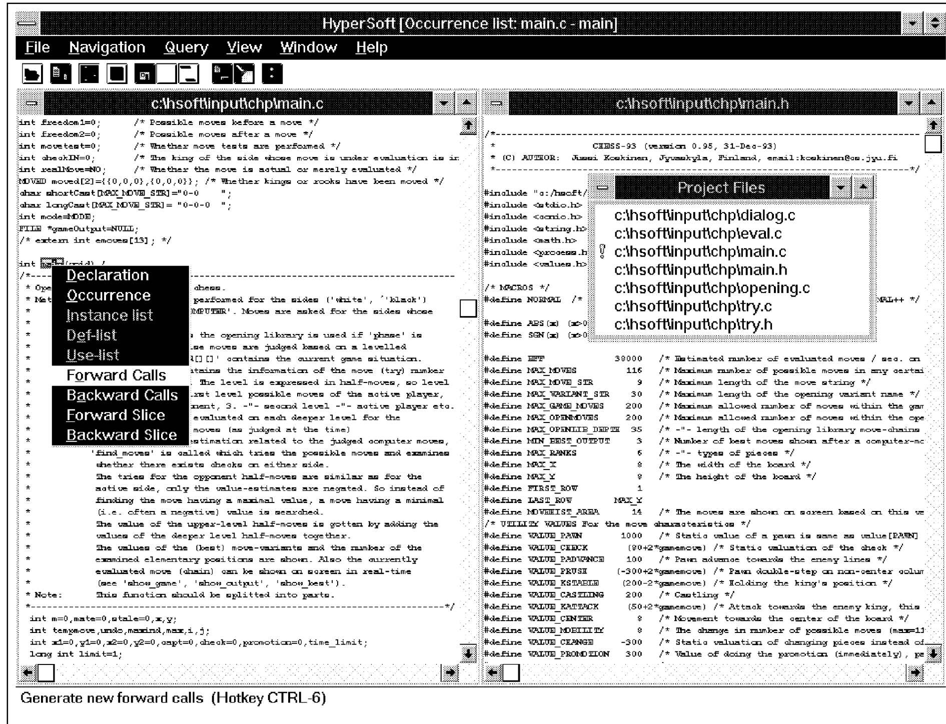


FIGURE 4 THAS specification

The user may give THAS generation commands over the source files by selecting (pointing) the relevant program elements and the desired operations. Figure 4 shows the way that THAS generation is initiated by the user. In the example the user has decided to generate a forward calling THAS (complete, traditional "call graph") initiated from the top of the calling hierarchy, the main function (at the corner of the pop-up window). The THAS type is selected from a pop-up menu. Examining forward calls is a typical way to follow the systematic top-down comprehension strategy. The THAS type shows the calling relations between the functions, and it is useful in gaining an overall understanding of the functionality of the program, or in finding all the functions whose behaviour is potentially affected by an intended change in the program.

Figure 5 shows part of the THAS created (the THAS extends to several files). Status information about the generation of the THAS is shown in the "Debug Messages" window. The hypertextual nodes within the hypertextual views are shown in reverse color. The figure shows, for example, a function call node find_moves within the module main.c (at the center of the left pane) from

which there is a link to another module, `eval.c`, where the function is implemented. If the user clicks the `find_moves` function call area (anchor), the cursor moves to the destination of that hypertext link. In this way the user may browse back and forth, by following the links set by the system (or via backtracking along them). Since the links may (and typically do) indicate the program dependencies which connect program parts somehow belonging together, comprehension of "delocalized program plans" is supported. The user may limit the levels to which THAS generation will expand. In the example the user has specified that the analysis should extend to the fifth calling level (see the "Debug Messages" window; there are various ways to limit the expansion of the analysis within the system). The program files into which the THAS has expanded are annotated with '!' in the project file window. This feature helps to reduce time and cognitive complexity in relation to the maintenance situation.

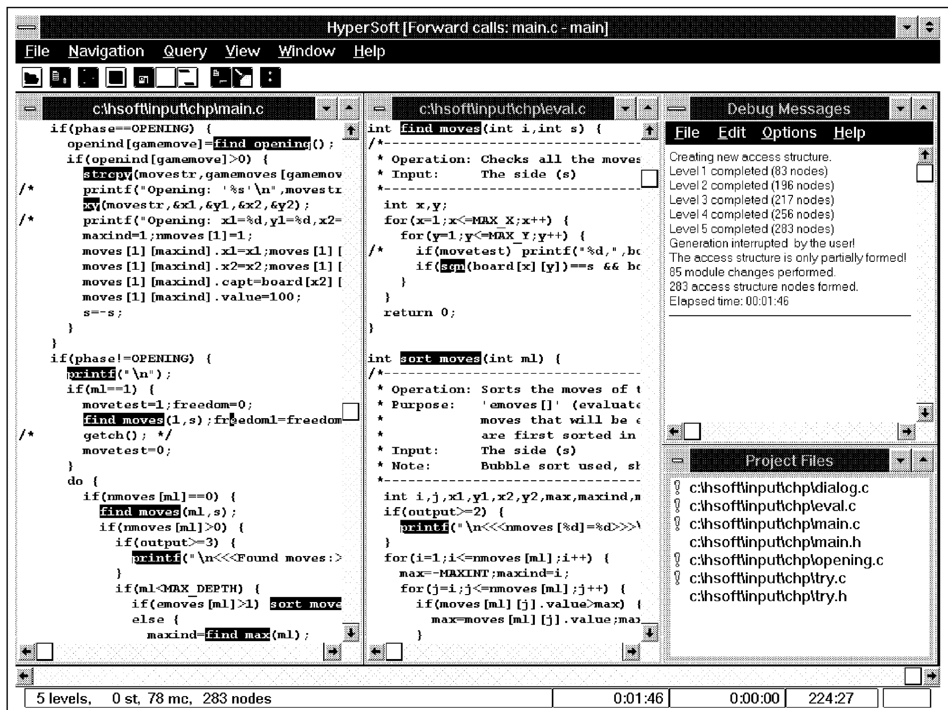


FIGURE 5 THAS formation

Figure 6 shows some of the links which exist between the THAS nodes. HyperSoft supports different levels of link viewing, namely: 1) links are not highlighted, only the nodes, 2) if a cursor is set on top of a node, the links originating from it are shown, 3) selected links are shown, 4) interprocedural links are shown, or 5) all the links are shown. In the figure all the defined links are shown. These options also help to reduce the amount of information represented and thus to overcome the cognitive bottleneck related to focusing attention.

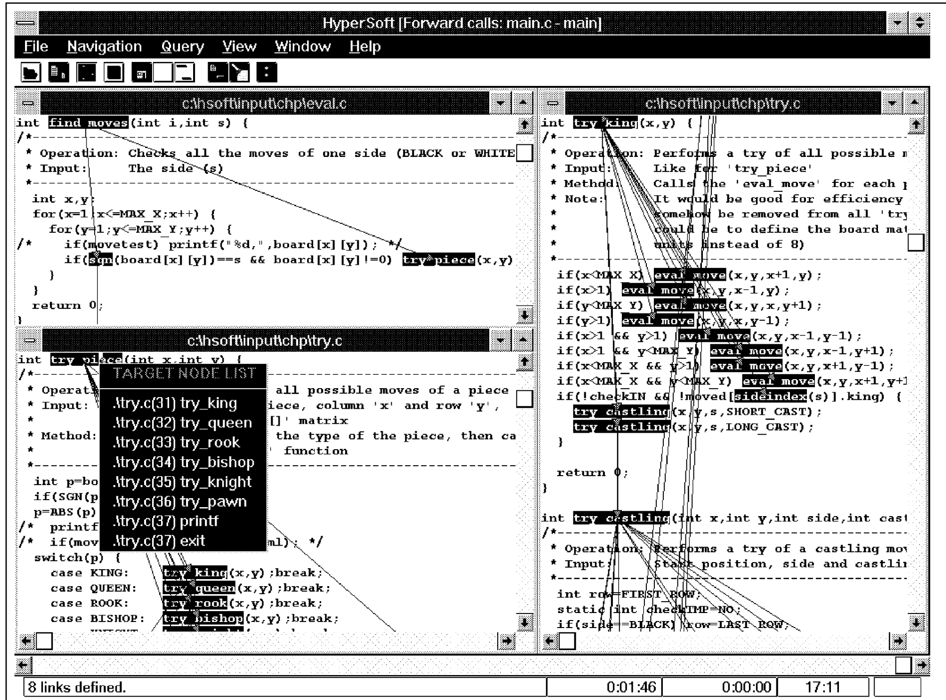


FIGURE 6 Links and navigation within the THAS

In the example the user has followed the links from `main` to `find_moves`, and further to the function `try_piece`, which calls the proper handler of each category of chess piece. These in turn call the move evaluator - `eval_move` - with proper square parameters. The function `eval_move` is the most central function within the example application, determining the relative goodness of individual chess moves. Thus, at this stage the user has arrived at the most central component.

If there is only a single link originating from the active node, that link is obviously the default. If there are multiple links, a pop-up menu is shown from which the user may select the link to follow. As usual in hypertext systems, the user may backtrack along the active link-chain one step at a time or go back to the first ("home") node simply by pressing the corresponding icons (the icons are located on the third row down from the top of the screen).

Graphical views and maps can be used to manage large THASs and to provide direct access to the source files on an abstract level. The importance of this was mentioned in relation to the empirical studies of hypertext in Section 2.3.2. It is also possible to view many THASs simultaneously within the HyperSoft system. As noted earlier, in the case of program text, a single fragmentation or linkage is not sufficient for describing all the relevant aspects.

Figure 7 shows two THASs using different methods: on the left the forward calling THAS described above is represented as a dependency diagram, and on the right a backward calling THAS from the function `eval_move` is represented as a structured map view. The backward calling structure shows the

functions which the initial function is called from. By following these links the user may combine the applied top-down comprehension strategy with the bottom-up strategy. Moreover, since the different kind of THAS types can be used in various combinations, the "as-needed comprehension strategies" are also supported. The backward calling information is useful, for example, when trying to understand the purpose of a function. The user may decide to let the THAS generation continue in the background whilst continuing to navigate through already generated THASs or program files, or using other applications.

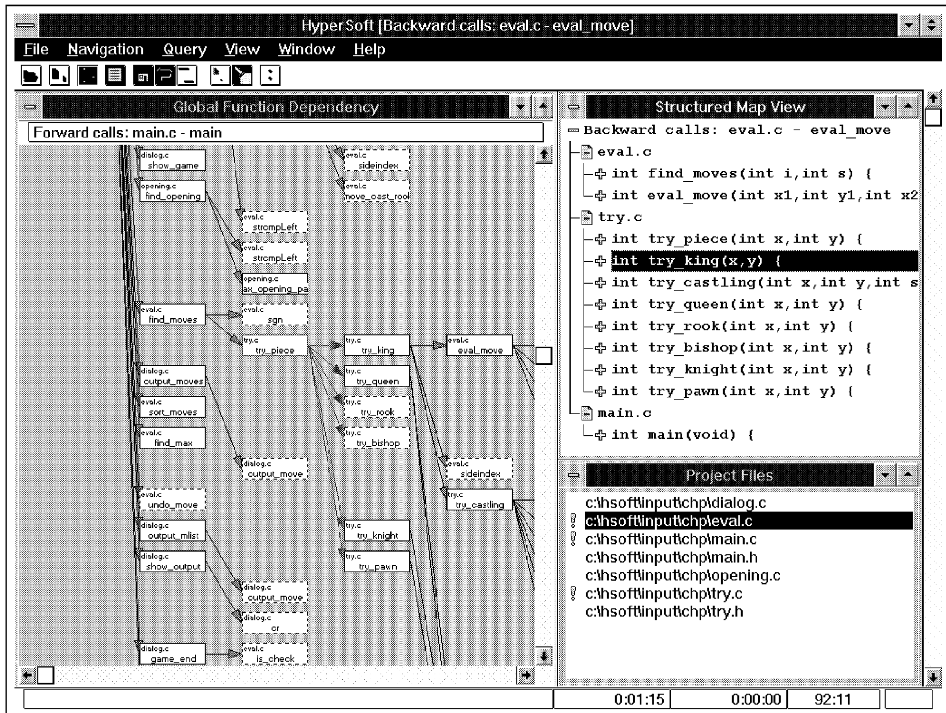


FIGURE 7 Multiple THASs and multiple representations

Figure 8 summarizes various ways of viewing the program text through HyperSoft. Starting from the left, the figure shows the so-called miniature view, hypertextual views, the structured map view, the project file window, and the graphical function dependency view. The general purposes of these views were described in Section 3.3.4. The views are used to manage the systematic examination of the created THASs. The views are linked to each other so that the user can move, for example, from the structured map view or the function dependency view to the hypertextual view. Within the hypertextual view the user can move along the transient links. This sort of view integration is important in order to support cognitively smooth operation.

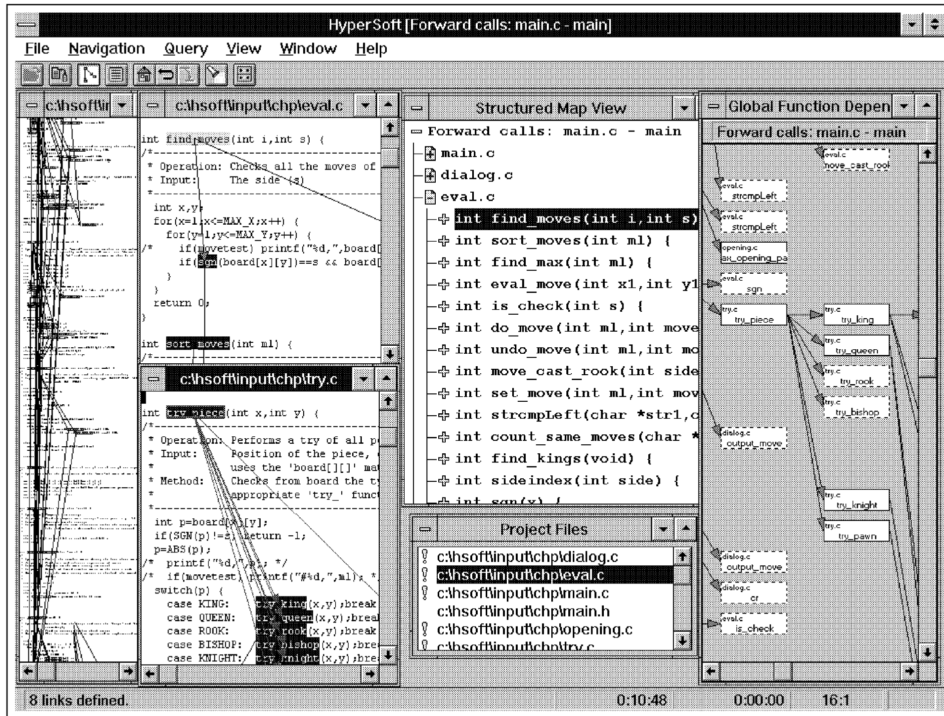


FIGURE 8 Various THASs and representations within the HyperSoft system

The above example session dealt with the forward calling THAS type. Another important THAS category within the HyperSoft system are program slices (Weiser, 1982). Program slices are more specific THASs than the above-described call graphs. When the user has first fixed the specific focus of interest (on a certain situation, for example, via browsing through the source code using other THAS types), slices may be used to restrict the subsequent area of interest. The importance of program slices as THASs is discussed in Article VI.

3.4.2 Backward slicing example

Within this session, the user aims to understand a function that produces erroneous results. Here we assume that the static program database has been formed properly, as in case of the previous example. Program slices can be formed according the same principles as in the previous example. Two examples of intraprocedural backward slices as THASs are represented in Figure 9.

In this case the user is interested in examining the program parts which may have an effect on some program part (slicing criterion). The example also exemplifies that multiple THASs can fluently be viewed (and thus compared) fluently simultaneously on screen. The slice in the left pane is started from the occurrence of variable `d` (in statement `++d`), and the slice at the right pane from the occurrence of variable `b` (in statement `--b`). Statements that might have a data flow effect on the slicing criterion variables are included in the THASs. The analysis proceeds against the direction of the control flow. For example, in the

slice on the left, there is a link from the home node (statement `++d`) to the place where `d` immediately received its value (statement `d=d+x`). From there, accordingly run links to the statements where `d` (statement `d=e`) and `x` (statements `x=x-1, x=a+b+d, ...`) received their values. In cases where the cursor is set on top of some of the anchors, a pop-up window (as in case of all THAS types) is shown which the user can use to systematically traverse paths of interest.

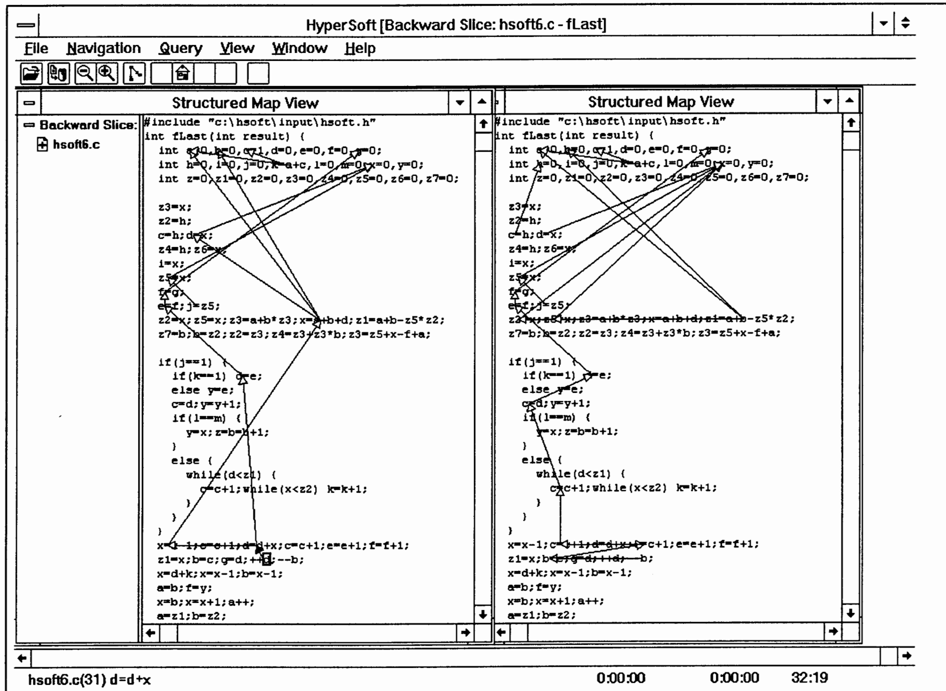


FIGURE 9 Intraprocedural backward slices as THASs

The slices can be used, *e.g.* to find the statements from which the possibly incorrect values of the slicing variables originate. Thus the user is provided with a view of the factors relevant to the current corrective maintenance or debugging situation. Since the formation of intra-procedural backward slices is fast, the user may constantly further specify the focus of interest (based on the existing understanding at each moment). The process of forming program slices in the HyperSoft system is described in (Koskinen, 1997).

3.4.3 Forward slicing example

In this session, the user aims to evaluate the potential effects of making a change in a program. The user is interested in finding out which program parts may be affected by the specific program part (to be changed). This sort of investigation is supported in HyperSoft via forward slicing. Figure 10 shows an example of interprocedural forward slicing. The slicing criterion is variable `g1` (second `g1++` statement, function `f21`). Downward slicing consists of the

analysis of the functions called from the initial function, whereas upward slicing consists of the analysis of the functions which the initial function may have been called from. The names of the functions which are upward calling contexts are always appended to the THAS since downward slicing must also be applied to them. Downward slicing is performed for each function within the calling trajectory (for those parts which appear after the call in a particular situation). What functions will actually be analyzed is determined by the status of the analysis. The process and terminology of slicing is described in more detail in Article III.

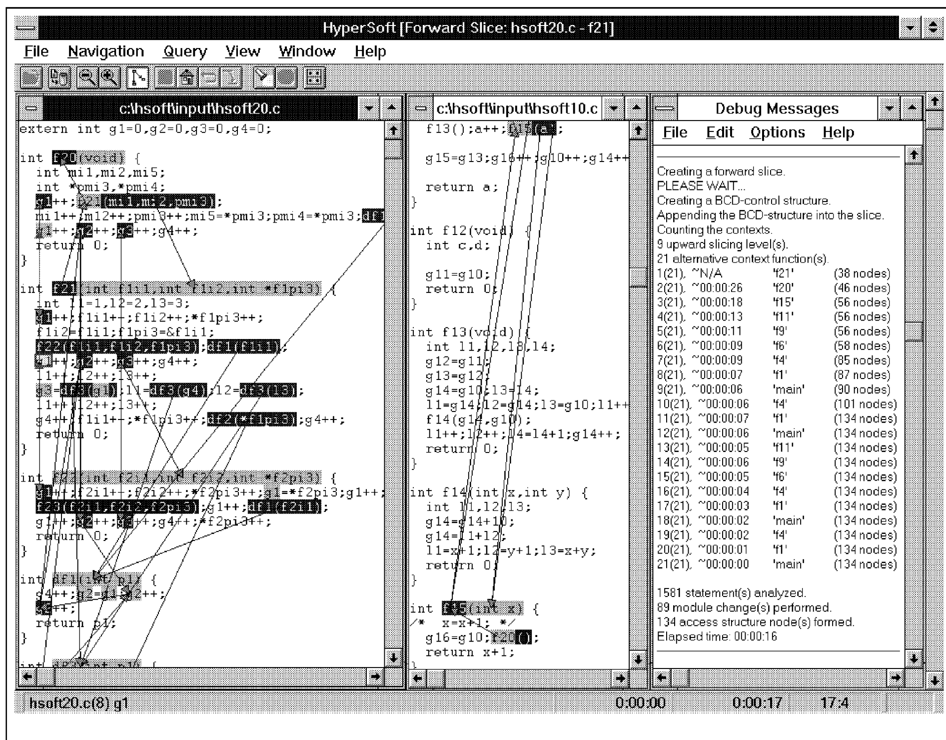


FIGURE 10 Interprocedural forward slice as a THAS

In the example, the slicing criterion is within the function `f21`. Thus, in this case, the initial function is `f21`. Thus, upward calling contexts are the functions which `f21` may be called from. Note that the reason for appending a certain node into the THAS or forming a link between two nodes may be indirect. The reasons may be related to the downward and/or upward slicing analysis and/or the usages of global variables. The example points out to the user that the global variable `g1` at the indicated program point cannot be changed without wide effects on the rest of the program. The example also demonstrates the form of the slices as THASs in cases of a very complex situation. The examination of large THASs is supported via the various views, which were shown in Figure 8.

Those views help the user, *e.g.* to view the large interprocedural slices at an abstract, aggregated level. In case of realistic projects, complete slices often are very large and their formation may take much time. Thus, in some cases, it is useful to restrict the slicing analysis and form partial slices instead. In HyperSoft this is supported such that the user may specify the number of upward and downward calling levels into which the slicing analysis will expand. This reduces radically the time needed to form slices in the case of interprocedural static slicing, which is described in Article IV. Thus, there exists a tradeoff between whether to form slices quickly or precisely. More examples of slicing are given in Article V, and the subject is discussed in more detail in (Koskinen, 1997).

3.5 Evaluation of the approach

The general idea of THASs (as well as many other reverse engineering techniques) is to provide support mechanisms for enhancing productivity by speeding up the work flow and decreasing errors and effort. Many software productivity models exist, including those represented by Boehm (1988b) and Banker *et al.* (1991), which discuss programmer productivity in detail. HyperSoft as a reverse engineering technique aims at increasing productivity, especially in the case of maintenance and comprehension tasks of legacy systems. We will first evaluate the approach on the basis of our empirical results and theoretical background (Section 3.5.1) and then compare it with other related approaches (Section 3.5.2).

3.5.1 Proposed benefits and probable drawbacks

The issues discussed include empirical evaluations within partner enterprises, studies conducted based on earlier empirical studies as reported in the literature, series of laboratory experiments conducted, interaction and representation aspects, technical considerations and limitations, target area of the approach, and congruence with the program comprehension theories.

Usefulness of HyperSoft

We have empirically evaluated the effects of using HyperSoft. The usefulness of the approach, the system, and the selected THAS set has been evaluated in three different ways: 1) by small-scale testing in the partner companies, 2) by comparing the capabilities offered by HyperSoft to the information needs of software maintainers as revealed in a series of earlier empirical studies, and 3) by two test series. The results support our hypothesis regarding the usefulness of the approach.

The results of the small-scale testing in the partner companies are presented in Article IV and the results of the analysis of the earlier empirical studies in Article VI. The appropriateness of the implemented THAS set has been

evaluated on the basis of comments received from the partner companies (Article IV) and on analysis of four empirical studies on the information needs of professional software maintainers (Article VI). The results suggest that the THAS set which is currently implemented in HyperSoft satisfies well the typical information needs of software maintainers for which static analysis is applicable.

In relation to the test series, we have measured the number of correct answers, wrong answers and time needed for the completion of given tasks. We have modelled usefulness via task performance (efficiency, accuracy, completeness, error rate, and time needed) and subjectively felt effort as well as subjective opinion of the usefulness of the maintenance support provided (Article VII). We have obtained clear support for our hypothesis regarding the usefulness of our approach, as reported in Article VII. The efficiency of task performance is clearly enhanced as compared to conventional program text browsing and information seeking. Completeness and accuracy of searches and the localization of the needed information are enhanced. Statistically almost significant results have also been obtained regarding the reduction in the amount of time needed to complete tasks.

These positive results can partly be explained by the way that interaction and hypertext representation are organized in HyperSoft. Mental resources can obviously be saved by using HyperSoft as compared to more elaborate ways of interacting with the support environment, including querying, which requires more elementary user operations. Thus, with HyperSoft, the comprehension process is interrupted only to a minimal degree by the technical details of searching for new information, owing to the simplicity of user interaction. Hypertext and graphical views are integrated in HyperSoft, as suggested by Brade *et al.* (1994), such that the graphical views contain links to the original program text, which should be useful in program comprehension. Because many different THAS types can be formed in HyperSoft, the text can be viewed from many different points of view within a single paradigm of information representation. Within HyperSoft, THASs are represented on top of the original text (as embedded components) which probably helps in understanding the context and surroundings of the nodes *cf.* Utting and Yankelovich (1989) as compared to isolated views. HyperSoft includes standard features of hypertext, *e.g.* for backtracking, which probably is of use in relation to program browsing since programmers typically browse programs back and forth while trying to comprehend them. Due to the automated analysis, relevant program components can be found completely, and thus traversed and handled appropriately. This is especially important with safety-critical applications, as well as, *e.g.* with respect to the Y2K problems.

Technical considerations and limitations

Since hypertext structures are formed automatically, the problems related to laborious manual hypertext formation and maintenance (Kaplan & Maarek, 1990; Østerbye, 1992) are avoided. This is essential, especially when changing source

programs. Also, for large programs the formation of many of the hypertextual structures manually would be impossible or at least impractical.

Owing to the applied general model, the structure of the HyperSoft system is modular, and thus the introduction of new THAS types is very straightforward (and the introduction of new programming languages is relatively uncomplicated). Since THAS types can be tailored to meet the requirements of specific maintenance tasks and related information needs, they can effectively aid in focusing the attention of the user within a given situation. The method of storing all of the global information in a global symbol table supports efficient program analysis in cases of modifying sources, which is a good first step towards incrementality.

In the case of realistically large source programs, the generation of some of the THAS types (especially complete static forward slices) in HyperSoft is slow. This is partly a matter of optimization. On the other hand, most of the THAS types can be formed reasonably efficiently. In any case, the decision to use certain THAS types naturally results tradeoffs between usage of memory and time and accuracy of results. An analysis of HyperSoft's space and time requirements and consumption is provided in Article IV. The topic is further discussed in Section 5.3 and the technical limitations of the implemented HyperSoft system are detailed in Koskinen (1997).

The application of HyperSoft in industrial settings in its present form would necessitate some changes in working habits. For example, there would be a need to use multiple tool environments, the compiler and the HyperSoft system, "simultaneously", and, possibly, the specific editor integrated with HyperSoft to make source program modifications. Another, better, possibility would be to use the HyperSoft ideas and to integrate the HyperSoft approach with some CASE or compiler-environment. These requirements could give rise to a certain amount of change-resistance among users or would necessitate studying the way that integration is achieved effectively.

Congruence with program comprehension theories

Software maintenance requires program comprehension. The HyperSoft approach and the way that program text is represented is congruent with the central issues of the main program comprehension theories. Program text is our main focus, which is also always the main focus of software maintenance and program comprehension. Support for delocalized program plans is achieved by enabling smooth navigation based on linkage between various dispersed and yet related program components. Support for systematic program comprehension strategies is supported by providing a way in which to browse through the relevant program components, which helps in focusing attention. Support for the top-down strategy is enabled via THASs that help to follow the order of function calls and to associate program components and the documentation related to them. Support for the bottom-up strategy is achieved by making explicit the low-level meaningful structural components of the program text. The composition of lower-lever elements into higher-level entities is, in particular, supported by the abstract, graphical views. Support for identifying beacons

(which serve as starting points for further browsing) is achieved by making them explicit as hypertextual nodes. Several different THASs and THAS types can then be generated by using these as starting points.

3.5.2 Solutions related to HyperSoft

Related research has been surveyed in Section 2.3.1 for reverse engineering and slicing and in Section 2.3.2 for software hypertext systems. Here we will provide a more detailed discussion of the similarities to and differences from the most closely related solutions and implementations. We will first consider general reverse engineering solutions and then software hypertext systems. We will make some comparisons at the end of the section.

Reverse engineering tools

Surgeon's Assistant (Gallagher, 1997) is a tool for visualizing decomposition slices of ANSI C programs. The tool is integrated with the Emacs text editor which allows the changes to be made only to the chosen slice, thus eliminating undesired change side-effects. The visualization component provides capabilities for collapsing regions of the resultant graph and to marking nodes. Gallagher (1997) notes the importance of providing complementary graphical views to support the investigation of large program slices. Venkatesh (1995) notes the need for showing that the program slices formed in realistic situations are "thin" enough, meaning that to be practical they should not be too large. Venkatesh has built a slicer for C programs (SLICE) and determined average worst-case metrics for the size of the dynamic slices. The conclusion is that dynamic slices are generally thin enough.

EDATS (Extensible Dependency Analysis Tool Set) (Wilde *et al.*, 1994) is a PROLOG-based reverse engineering tool for heterogenous software environments. The tool includes a simple query-by-example type of query language via which information about source code objects can be retrieved. The system has been tested in form of a case study with a 25,000 LOC C program. The focus in the EDATS project has been on back-end features. The data model behind the tool consists of 12 entity subclasses and 22 dependency subclasses (and classes for the inverse dependencies). The authors conclude that an important feature is the possibility to retrieve information from chains of dependencies.

CARE (Linos *et al.*, 1993b) is a re-engineering tool for C programs, maintaining a repository of entities and relations (control flow and data flow dependencies). CARE focuses on visualization and incremental modifications of programs. Linos *et al.* (1993b) emphasize the importance of an open architecture, meaning that it should be possible for users to use their favorite tools (editor, debugger etc.) in an integrated fashion. They use so-called colonnade graphs to represent data flow information. Colonnade graphs represent the related variables, types, parameters, functions, and constants organized so that each category forms a column. The study of colonnade graphs is ongoing (Linos *et al.*, 1999). Control and data flow dependencies can be viewed either through the so-called monolithic views (entailing complete code or control flow or

colonnade graph) or through so-called multiple-slice views (enabling variant graphical representations of the dependencies). The features include the so-called call graph and colonnade editors. The tool includes program slicing. The features of the tool have been empirically evaluated using 40 senior computer science students and a 2,000 LOC program. The results suggest that the most useful features include having access to conventional, textual code representation from within the graphical representations, search, undo, zoom, highlighting, program slices, and the ability to move graphical entities on display.

CIA (C Information Abstraction system) (Chen, Y.-F. *et al.*, 1990) automatically extracts relational information from C programs and stores it into a program database. The created database can be processed and retrieved by any relational database system. The conceptual data model of the system contains 5 data types (functions, global variables, types, files, and macros) and 11 relationships between these component types (including file inclusion, function references to other data types, variable references to other data types, references among types, and type references to macros). The system applies relational, textual, and graphical views. The authors mention that with a proper interface the described information could be represented as hypertext.

The approach of Heisler *et al.* (1993) provides so-called structural and functionality views of C programs. The structural view shows relations among structural elements such as code blocks and variables, whereas the functionality view provides a hierarchical outline of the functionalities of the program. The implemented tool supports ripple-effect analysis, program slicing and redocumentation. The tool runs under UNIX and is implemented using Yacc and Lex.

Storey *et al.* (1997) have empirically compared three representative reverse engineering tools which also contain some hypertext capabilities and which were mentioned at the end of Section 2.3.2. These tools are Rigi (Muller & Klashinsky, 1988), SHriMP (Storey & Muller, 1995), and Sniff+ (TakeFive, 1998). The comparison comprised 30 subjects and a 1,700 LOC C program to be comprehended. Storey *et al.* conclude that the features that these reverse engineering tools possess may affect the program comprehension strategies applied, the dependency relationships provided by all three tools were used by most of the subjects, that searching features are needed, and that seamless integration between the higher-level views and the source code is desirable.

Nine representative static call graph extractors for C language, including CIA (Chen, Y.-F. *et al.*, 1990), Imagix (1998), Rigiparse/Cparse (Muller & Klashinsky, 1988), and Cflow (which is distributed with the Unix operating system), are compared in detail by Murphy *et al.* (1998). The main conclusion of the paper is that there exists a significant difference between the content and form of the call graphs produced by the different tools owing to the design choices made.

As mentioned in Section 2.3.1, there are many intermediate program representations. Those representations can also be used as a basis for the formation of hypertextual structures. These representations include: program dependency graphs (Ottenstein & Ottenstein, 1984), system dependency graphs (Horwitz *et al.*, 1990), unified interprocedural graphs (Harrold & Malloy, 1993), and combined C graphs (CCGs) (Kinloch & Munro, 1994). These graphs typically

contain explicit representations of a program's control and data dependencies. CCG is a fine-grained intermediate program representation, which can be used as a basis for forming *e.g.* program slices, call graphs, data flow information, definition-use information, and control dependency views. CCG is composed of Function CCGs (FCCGs), each representing an individual function of the C program. Each FCCG is a directed graph containing several different types of edges connecting its vertices. The content of the vertices typically corresponds to elementary statements or their parts. The approach is implemented as a tool named PERPLEX which produces a generic PROLOG fact base. Kinloch and Munro (1994) state that one of the advantages of CCG, as compared to other graphs, is that it eliminates redundant information.

Software hypertext systems

Whorf (Brade *et al.*, 1994) is a hypertext tool for the maintenance of C programs, targeted at supporting the recognition of delocalized program plans on the basis of an as-needed strategy via multiple, concurrent views of software. The views provide capabilities for source code editing, and for representing call graphs and variable and function cross-references. The supported structures include identifiers, calling dependencies, and containment. The usefulness of the tool is motivated by stating that it focuses the search process, provides quick access to the desired information and access to additional information (related to functions and variables). The system has been evaluated with 12 subjects (professional programmers and graduate students) with a 250 LOC program. The evaluation of using the tool as compared to paper documentation suggests that the applied approach is useful for accessing information related to software.

In the approach developed by Østerbye and Nørmark (1994) the key principle is the separation between internal hypertext representation and external screen representation. This approach is termed (semantically) rich hypertext, and is implemented in the HyperPro system. HyperPro provides an interaction engine governed by rules for representation, interpretation of events, and menu setup, relative to the type hierarchy of nodes and links. One of the focus areas has been the typing of nodes and links and the internal structures of hypertext nodes.

The HyperCASE environment (Cybulski & Reed, 1992) is an architectural framework for integrating a collection of tools. The system provides a visual, integrated and customizable software engineering environment consisting of loosely coupled tools for presentations involving both text and diagrams. HyperCASE combines a hypertext-based user interface with a common knowledge-based document repository. The tools include managers for reuse, integrity, specification, documentation, and configuration, as well as abstract trackers and analyzers.

Similarly, DynamicDesign (Bigelow, 1988) is a CASE environment containing hypertext capabilities. Nodes contain project components and links depict the relationships between components. The approach covers requirements specifications, system and user documentation, and source code. Although the system explicates such concepts as object code and symbol tables, the

relationships between source code elements are not emphasized nor discussed. The work addresses the importance of integrated CASE as well as the automatic generation of sequential and relational links, storage of fine-grained (intra source code) information in a relational database and the formation of a relationally complete query language.

The focus of ISHYS (Garg, 1989; Garg & Scacchi, 1989) and DIF (Garg & Scacchi, 1990) is on managing software documents. Through the development process, DIF (Documents Integration Facility) stores the relevant information about the target system (related to its design, development, use, and maintenance) in textual objects as nodes of hypertext.

Comparisons

The possibility of representing C program information in the form of hypertext is first suggested in Chen, Y.-F. *et al.* (1990). Since then, the most notable and comparable tool to HyperSoft in this regard has been Whorf (Brade *et al.*, 1994). The aims of HyperSoft and Whorf are similar in the following regards: both support the C language, apply hypertext explicitly, aim at supporting the as-needed comprehension strategy and comprehension of delocalized plans, and apply multiple representations which are linked to each other. The motivation given by the developers of Whorf applies also to HyperSoft. The graphical representation of call graphs is different, which is quite typical of the different reverse engineering tools, as noted by Murphy *et al.* (1998). Whorf does not include program slicing. As in the approach of Østerbye and Nørmark (1994), we have aimed at separating internal hypertext representation and external screen representation. In addition we have aimed at providing a versatile set of THAS types based on our model of hypertext representation.

Our approach differs from other software hypertext systems in the sense that we focus solely on the source code. The HyperCASE environment (Cybulski & Reed, 1992) does not specifically address the problems of source code analysis and representation. The emphasis is rather on forward engineering throughout the whole life-cycle of software. DynamicDesign (Bigelow, 1988) is very similar to HyperCASE in regards relevant to HyperSoft.

Within the HyperSoft system openness, as suggested by the developers of CARE (Linos *et al.*, 1993b), has been approached by making it possible to integrate the desired text editor with the system (Programmer's File Editor; PFE is the current editor). The way that program slices are represented in CARE is different from that in HyperSoft. The possible differences in their relative usability are unknown. Both represent slicing information in a sensible way. The functionality view of the approach of Heisler *et al.* (1993) is analogous to the structured map view of the HyperSoft system. The slicing features of HyperSoft and other slicing tools are compared in Article III. Like HyperSoft, HyperPro (Østerbye & Nørmark, 1994) also aims at the separation of hypertext representation and screen representations. The HyperPro project, however, is more focused on the issues of the storage layer than HyperSoft. The suggestions and observations related to program data storage (Kinloch & Munro, 1994), dynamic program slicing (Venkatesh, 1995), and program slice representation (Gallagher,

1997) are also relevant to HyperSoft. It should be noted that these and other techniques introduced after the architecture of the HyperSoft system was designed (1994) could not necessarily have been taken into account when the system was implemented.

Empirical studies on hypertext effects were listed in Section 2.3.2. When comparing the evaluation of the HyperSoft system (see Section 4.8, Article VII) to the evaluations of the related tools, the following observations can be made. Whorf (Brade *et al.*, 1994) is in many regards similar to HyperSoft. Thus, the results received from the evaluation of HyperSoft are relevant to the development of tools like Whorf as well. Since Whorf has been evaluated with only a small program (250 LOC) and only as compared to using paper documentation, the results with HyperSoft complement the results received from the use of Whorf.

On the other hand, EDATS (Wilde *et al.*, 1994) has been evaluated with a larger program (25,000 LOC), but only as a case study. The observations of Wilde *et al.* (1994) and Storey *et al.* (1997) support the usefulness of dependency-based tool features. The related tool which is best evaluated is CARE (Linos *et al.*, 1993b) (2,000 LOC program, 40 subjects comparing the features of the tool). The observations of the most important features are largely taken into account within the implementation of the HyperSoft system: there are links from the graphical views to the program text in HyperSoft. Storey *et al.* (1997) have also drawn attention to the importance of this sort of integration. Hypertextual nodes are represented as highlighted elements within the program text, and program slices (which are considered as useful) have been implemented. The importance of a search function (complementing browsing) has been noted by Halasz (1988); Linos *et al.* (1993b); Storey *et al.* (1997). HyperSoft currently does not include a search function, which, obviously, would be a simple but important additional feature. A search function would clearly be a first step to the direction of supplementing the approach with querying capabilities, an option to be discussed in Section 5.2.

4 OVERVIEW OF THE ARTICLES

In this chapter we will summarize the main parts of the study. We will also briefly characterize the purposes of the research and the applied research methods. The main part of the work is reported in the included articles. The Articles I and IV-VI mostly deal with the HyperSoft model and approach, and Articles II and III with the HyperSoft system. The HyperSoft system is also described at some length in Section 3.3 and in Articles IV, and V. The empirical evaluations presented in Article VII are relevant to both the system and the approach. Owing to the fact that the dissertation includes published articles, the treatment of some issues is repeated in them. The HyperSoft model and system have been developed gradually during the research process, which means that the most detailed descriptions are to be found in the later articles. Since the approach is potentially very versatile, much effort has been made to gather together references to the applicable methods and algorithms as well as to the potential areas of application, see Section 2.3. Since studies of program dependencies, their representations, and their automatic extraction from the program text have been extensively reported in the literature, we have focused on the problem of forming hypertext based on program code (instead of inventing new program analysis algorithms).

The methodological characterization is given in terms used by Haworth *et al.* (1992) and, especially, Nunamaker *et al.* (1991), who have focused in their paper on describing systems development as a research methodology. Systems development research includes constructing a conceptual framework (theory formation), developing a system architecture, systems analysis and design, building the (prototype) system, and observing and evaluating the system. Theories can suggest research hypotheses, as well as guide and enable research. Systems development may take the role of 'proof-by-demonstration'. Haworth *et al.* (1992) have represented a framework for classifying software maintenance research applying three central targets of research: programmer, code, and requirements. In our research, the emphasis has clearly been on code and on the support of the interaction between code and programmer. With regard to programmers we need to chart their main attributes while empirically evaluating

our tool. Requirements mostly correspond to information needs. The information needs of the programmers must be satisfied in order for them to be able to fulfill the requirements set to the software.

4.1 "Program Text as Hypertext: Using Program Dependences for Transient Linking"

Koskinen, J., Paakki, J. & Salminen, A. 1994a. Program text as hypertext - using program dependences for transient linking. In *Proc. 6th Int. Conf. Software Engineering and Knowledge Engineering* (*SEKE'94*). Skokie, IL: Knowledge Systems Institute, 209-216.

Research problems and methods

The paper investigates the possibilities of viewing program text as transient hypertext. The literature has been reviewed for that purpose and for identifying meaningful research questions in the area. The formation of a conceptual framework has been commenced. This formation entails the development of the first version of the HyperSoft model for viewing program text as hypertext.

Content and results

The paper introduces a model for viewing program text as hypertext and explores the possibilities of creating hypertext automatically on the basis of well-known program dependencies. The model is a specialization of a generic domain-independent model for text databases (Salminen & Watters 1992). The paper also presents examples of some of the possible access structures. The intended application domain of the model is software maintenance.

One of the problems of representing program text as hypertext is that there does not exist a unique fragmentation, nor a unique set of links that would be suitable in all situations. Moreover, because program text is typically not static, but frequently changes, the formation of all of the potentially needed hypertextual structures manually is not practical. The idea of *transient hypertext* is introduced. The user is provided with the possibilities for the dynamic specification of structures to support hypertext access. The syntactic structure is a parse tree for the program with respect to the grammar, and the access structure is a graph or hypergraph consisting of a set of program parts and a set of links connecting the parts.

Because program parts and their relationships have been extensively studied in programming language research, the automatic creation of hypertext is possible. Many of the important program dependencies are typically represented in the form of different dependency graphs which are surveyed, and their potential as a basis for hypertext generation is discussed. The structures traditionally used to contain program dependency information include parse

trees, module dependency graphs, call graphs, data flow graphs, control flow graphs, and program dependency graphs. The dependencies which exist between the elements of these structures include, for example, structural dependencies, cohesion, coupling, calling dependencies, data flow dependencies and control flow dependencies. The possibilities of using the common properties of program parts as a basis for this purpose are also discussed. The possible properties include the text type of a part, the value of the part, containment, interesting level, and the complexity of the program part.

4.2 "HyperSoft: An Environment for Hypertextual Software Maintenance"

Salminen, A., Koskinen, J. & Paakki, J. 1994a. HyperSoft: an environment for hypertextual software maintenance. In B. Magnusson, G. Hedin & S. Minör (Eds.) *Proc. Nordic Workshop on Programming Environment Research* (*NWPER'94*). LU-CS-TR: 94-127. Lund, Sweden: Lund Univ., 25-37.

Research problems and methods

This paper tackles the problem of how to construct a support environment in which program text can be viewed as hypertext. The HyperSoft model is used as a basis for designing the architecture of the HyperSoft system. In this stage, only system design exists, not actual implementation. System extensibility and modularity are stressed as central goals.

Content and results

The paper presents the general architecture of the HyperSoft system (*cf.* Figure 2), which combines the techniques of static program analysis and dynamic hypertext access. The architecture separates on one hand the back-end and the front-end components, and on the other hand the components generating static (permanent) and dynamic (transient) data.

There are static links between the components of the program database. Links from the parse trees to the symbol tables enable fast retrieval of symbolic information whereas the links from the symbol tables to the parse trees enable the retrieval of the position and contextual information of the symbol occurrences. The THASs are formed by the generator component and are, like the static structures, stored in the program database. When the generation of a THAS has been completed it is passed to the interface component, whose architecture is described. Ways of dealing with the disorientation and cognitive overhead problems, often related to hypertext systems, are discussed, as well as the general ideas of navigation through THASs.

The main interconnections between the program database components with some examples of link and node types are given. The aspects of source

code change and incremental updates of the database are discussed. The possible THAS set to be implemented is discussed. Examples of architectural THASs, structures based on data flow, calling structures, and program slices are given.

4.3 "Creating Transient Hypertextual Access Structures for C Programs"

Koskinen, J. 1996c. Creating transient hypertextual access structures for C programs. In M. Kavanaugh (Editorial production) *Proc. 7th Israeli Conf. Computer Systems and Software Engineering* (*ICCSSE'96*). IEEE Computer Soc., 56-65.

Research problems and methods

This paper investigates the lessons learned from the HyperSoft prototype (v. 0.7), which was constructed during the first phase of the HyperSoft project. System requirements (including implemented THAS types) are mainly determined on the basis of the information received from the private-sector partner companies. Experiences and observations using the prototype have further been taken into account when refining the model and designing the system.

Content and results

The paper describes the implementation of the back-end components of the HyperSoft system. The detailed architecture of the system, the contents of the program database, and the process of creating the static and transient structures are presented. The transient access structures (THASs) are formed on the basis of the HyperSoft model and method. THASs are classified as singletons (references), lists, trees, and general graphs.

Examples of occurrence lists, call graphs and program slices as THAS types are given. Since the HyperSoft system is not only a slicing system, a versatile set of THASs have to be supported and thus a more general approach than is typical in pure slicers have to be taken. HyperSoft's slicing capabilities are compared to the features of other slicing tools. The way of forming the slicing THAS types is described. The most complex of the implemented THAS types are static forward slices. In the longer version of the paper, published as part of Koskinen (1996b), they are used as an in-depth example of building THASs. In the included paper, the process of interprocedural static slicing is described in detail. Some possibilities for improving its current implementation are outlined in Koskinen (1997). Program parts, dependencies, and linkages related to slicing, as well as the formation and representation of the slices as THASs are briefly discussed.

The main problem areas of the implementation of HyperSoft are discussed. These include the efficient execution of static interprocedural program slicing and the reduction of the amount of static information needed. A much

wider discussion on these topics can be found in Koskinen (1997). In static interprocedural slicing there is a clear tradeoff between whether to produce precise and complete slices slowly or to obtain partial results quickly.

4.4 "Automated Hypertext Support for Software Maintenance"

Paakki, J., Salminen, A. & Koskinen, J. 1996. Automated hypertext support for software maintenance. *The Computer J.* 39 (7), 577-597.

Research problems and methods

An extended discussion of the HyperSoft model and its relation to the developed HyperSoft system is provided. More observations are made with respect to HyperSoft's usability. The usability evaluation has been performed by obtaining feedback from the representatives of the partner enterprises (the number of participating subjects was 4) related to the suggested and implemented HyperSoft functionalities and via questionnairies distributed to the test users (3), who were professional software maintainers. Moreover, a more detailed examination of the system (v. 0.8) is performed with regard to the space and time needed by using simulation with example programs. The lessons learned from the development of the system and the feedback received from the partner companies have served in improving the design of the later versions of the system and the planning of its statistical empirical evaluations (Article VII).

Content and results

The model is extended to include four layers: the source code layer, the syntactic structure layer, the access structure layer, and the user-interface layer. The layered structure of the model makes the extension of the sets of supported programming languages, THASs, and graphical user interface environments easier. The original source code is retrieved from the source code layer to the syntactic structure layer in order to create the static program database, and to the interface layer in order to display the code. The syntactic structure layer determines the possible THAS node types. The THASs are created at the access structure layer on the basis of the information determined by the syntactic structure layer. Finally, the visual representation of these structures is reached at the interface layer. The use of the model is illustrated with an example program. The model is also compared to Dexter, which is another hypertext model. See (Halasz & Schwartz, 1994; Leggett & Schnase, 1994; Grønbaek *et al.*, 1994; Grønbaek & Trigg, 1994) for information about Dexter.

The questions of source code storage and retrieval need to be answered at the source code layer. The questions which need to be answered at the syntactic structure layer include the determination of the information which is to be stored statically and the method(s) of creating the program database and its storage form. Issues which need to be resolved at the access structure layer

include the determination of useful THASs, the THAS seed and the method of specifying it, the mixture of different node and link types within a single THAS, the need to combine different THASs, and THAS storage mechanisms. The interface layer gathers decisions concerning the necessary textual and graphical windows and views, node and link visualizations, and the interaction between the user and the system during navigation and THAS specification.

A relational characterization and classification of program dependencies is presented to serve as a basis for concrete THAS formulations. This abstract classification includes, for example, matching, subordination, control, and data dependencies. The dependencies are characterized with respect to their reflexivity, symmetry, transitivity, and arity. Elementary THASs contain program elements bound together on the basis of the relations subsisting between the above types. Dependencies are characterized as navigation structures.

The paper introduces the implementation of the HyperSoft prototype and characterizes its typical application area: adaptive maintenance of legacy systems. Preliminary THASs are characterized according to the dependency classification and are presented as they are shown to the user through the interface component. The implemented THAS types include occurrence lists, calling structures and program slices. An occurrence list reveals the points in the program where a certain symbol is defined or used. Calling structures reveal the forward or backward calling dependencies between function calls and function implementations. A backward slice reveals the statements which may have an effect on the value of a specified symbol occurrence(s). A forward slice shows the statements which would be affected if the value of a specified symbol occurrence(s) were changed. The interface layer allows the display of the source code in windows, the specification and manipulation of THASs, and navigation within them.

4.5 "From Relational Program Dependencies to Hypertextual Access Structures"

Paakki, J., Koskinen, J. & Salminen, A. 1997. From relational program dependencies to hypertextual access structures. *Nordic Journal of Computing* 4 (1), 3-36.

Research problems and methods

The focus area of the paper is on further theory building. A model of program dependencies is developed, and the THAS types implemented in the HyperSoft system (v. 0.85) are characterized using the terminology of the model.

Content and results

Program dependencies can be used as a basis on which to determine the hypertextual links to be formed between nodes (program parts) by binding together

related node pairs. The paper provides a relational characterization and classification of program dependencies. The classification is represented as a lattice using the OMT notation (Rumbaugh *et al.*, 1991). Subtypes within the class hierarchy inherit the properties of their superclasses. Dependency classes have the following attributes: *start* and *destination types, arity*, an *algorithm* for their extraction/determination, and *relation*. The three relational properties, namely: *reflexivity, symmetry*, and *transitivity* (and their inverses) are used as a basis for characterizing program dependencies. The *relation* property specifies whether all, some, or none of the dependencies within a category are reflexive, symmetric, or transitive, respectively. The classification can be used, *e.g.* as a basis for the systematic development and evaluation of the dependency features provided in the HyperSoft system and in other similar reverse-engineering tools.

Some of the classes are clearly more important than others in the sense that conventional programming languages apply mechanisms which produce dependencies belonging to these classes. These classes, termed as *Essential, Incidental, Symmetrical, Subordinative, Matching, Extrovert, Structural*, and *Imperative*, are discussed in detail in the paper. The most important dependency categories are Matching and Subordinative. The subtypes of Matching are *Lexical* (program parts share a similar textual representation), *Syntactic* (sub parse trees for parts are isomorphic), *Semantic* (parts share some computational, run-time value), and *Qualitative* (parts share some quality factor of software engineering). Within the category of Subordinative, the clearly most important subtypes are *Control* (part *a* is executed after *b* on condition *c*), and *Data* (the value of data on part *b* is dependent on the value of data on part *a*). The Matching dependencies can be supported via THAS types, which are lists, whereas Control and Data dependencies may be supported via program slice THAS types.

The process of forming THASs is represented using an example. Hypertext links are not needed for reflexive relations. For symmetric relations a unidirectional linking is sufficient (because of the available backtracking features). Indirect transitive relations should not usually be supported via linking, since the linkage tends to become too dense. On the other hand, direct transitive relations require hypertextual linkage. There exist multiple combinations of these three basic properties in the case of individual program dependencies. Elementary dependencies can be used as a basis for compound dependencies. These, in turn, can be used as a basis for THAS types using set operations (union, intersection, difference, and restriction). Since THASs are graphs, a section is devoted to characterizing them from the graph-terminological view-point. The HyperSoft system is used as an example environment whose THAS types are characterized using the terminology of the dependency model. The dependencies manifested within the currently implemented THAS types are described in terms of the classification.

4.6 "Hypertext Support for Information Needs of Software Maintainers"

Koskinen, J., Salminen, A. & Paakki, J. 1999. Hypertext support for information needs of software maintainers. Univ. of Jyväskylä, Jyväskylä, Finland. *Computer Science and Information Systems Reports, Working paper* WP-37. Submitted for publication to *IEEE Transactions on Software Engineering.*

Research problems and methods

The research problem of the paper concerns the information needs of software maintainers and their relationship to HyperSoft solutions. The paper analyzes and classifies the information needs of software maintainers as revealed in a series of earlier empirical studies. The HyperSoft access structures (v. 1.0) are then evaluated with respect to those information needs.

Content and results

In this paper information needs have been analyzed based on the data gathered in the series of empirical studies conducted by von Mayrhauser, Vans and Howe (von Mayrhauser & Vans, 1995b; 1997b; 1998; von Mayrhauser *et al.*, 1997). Those studies provide data on the information needs of professional software maintainers at the most detailed and comprehensive level available. These information needs have been ranked, grouped, characterized, and analyzed from the view point of supporting them via hypertext techniques. The nature and the interpretation of the information needs are discussed. Certain kinds of information are needed, especially in the case of certain kinds of maintenance tasks: general, corrective, preventive, and adaptive. The information sought is grouped on the basis of source from which it can be obtained and the type of analysis required. The alternatives are static analysis of source code, dynamic analysis/code execution, analysis of documentation and other textual material, and recording of user operations and session history.

The way in which information needs relate to the formation of THASs is described. Five large THAS categories, *i.e.*, references, lists, sets, trees, and general graphs are proposed as a way of gathering the data satisfying the most prominent information needs. Most such needs are simple, and can thus be satisfied on the basis of references or lists. For the satisfaction of some of the most important needs, the convenient form, however, is a general graph. Individual THAS types and THAS-type variants can be tailored to satisfy situation-dependent information needs. Examples of THAS types are provided, most of which are supported in some form by the HyperSoft system. This indicates the appropriateness of the THAS set implemented. Possible extensions related to hypertextual support based on dynamic analysis of programs and automated analysis of system documentation are also briefly sketched.

4.7 "Evaluations of Hypertext Access from C Programs"

Koskinen, J. 1999c. Evaluations of hypertext access from C programs. Conditionally accepted to be published in *Journal of Software Maintenance: Research and Practice.*

Research problems and methods

The need to validate empirically the effectiveness of hypertext via usability studies in the software engineering context has been addressed, *e.g.* by Ziv and Osterweil (1995). In this paper the effectiveness of the support provided by the HyperSoft system is evaluated. Our main hypothesis is that HyperSoft enhances information retrieval performance in case of typical tasks as compared to conventional text browsing and search. The main part of the study consists of two independent series of classical laboratory experiments using a control group and a test task to find differences in performance between a HyperSoft group and a control group. The subjects of the first experiment series (N=23) were (on average) fourth-year computer science students, and the subjects of the second experiment series (N=47) were (on average) second-year computer science students. Thus, the subjects were novice programmers.

The test task and the sample program were the same in both experiments. The subjects, however, were not the same and the actual task set was varied, although the tasks were similar on both occasions. On each occasion, task performance between the group using HyperSoft and the group using the Borland C/C++ environment (its text browsing and search functions) were compared. We measured task performance in the case of sample information requests. The example program was a non-commercial chess program of about 2,500 LOC written in ANSI-C (Koskinen, 1993). The task sets were based on the results of the study by von Mayrhauser and Vans (1995b). The significance of the differences between the groups is determined using statistical methods (Student's *t*-tests, Mann-Whitney U tests, and two-way variance analysis).

Content and results

The paper gives the results of empirical evaluations of the HyperSoft system using computer science students as subjects. The results of the first experiment have been published in the proceedings of *IWPC'99* (*International Workshop on Program Comprehension*) (Koskinen, 1999a), which is the main international forum on program comprehension research.

The results clearly support our hypothesis as to the usefulness of the HyperSoft system and of transient hypertext support for software maintenance. Many of the differences in performance between the groups are statistically highly significant. The hypothesis that HyperSoft enhances task performance as compared to conventional text browsing and search was confirmed at a 0.000 level of risk. This means that the risk that the results are due to chance is

virtually non-existent. This result was achieved in both experiments, the results of the two experiments also support each other very well in several important regards. For the combined data, the same results were obtained both through variance analysis and *t*-tests. The efficiency in performance of the HyperSoft group was over two times better than that of the control group in both experiments. In general, the subjects using HyperSoft were able to find more complete answers to the questions posed and to perform the tasks more efficiently and in less time than the subjects in the control groups. Task-wise results are also analyzed in detail. It is probable that in case of more complex (more open) tasks, the benefits would be even greater, as has been reported by Chen, C. and Rada (1996). In case of more complex tasks, it is also feasible to apply multiple THASs.

The limitations of the empirical experiments are discussed. These limitations include the fact that only the information-seeking behaviour of novices was studied. Professional software maintainers may need more specified support than novices *cf.* (Soloway *et al.*, 1982; Gugerty & Olson, 1986; Cunniff & Taylor, 1987). However, especially program slices provide detailed information, which is of value to professionals also (Weiser, 1982). Moreover, the HyperSoft approach and system also make it possible to support flexibly other information needs of professional maintainers (as noted in Article VI), since THAS types can be tailored to meet emergent user needs. Although software modifications were not made, the general importance of the localization activities studied is clearly prominent even as such, since the selected task types were representative.

4.8 About the joint articles and other publications

The summarizing part and the Articles III (Koskinen, 1996c), and VII (Koskinen, 1999c) were written solely by the author. The author is the main contributor of Article VI (Koskinen *et al.*, 1999). The other joint work; Articles I (Koskinen *et al.*, 1994a), II (Salminen *et al.*, 1994a), IV (Paakki *et al.*, 1996), and V (Paakki *et al.*, 1997) were written in close collaboration by the authors. All the published articles included have been refereed by at least two international experts. All published/submitted conference papers and journal articles have been edited and revised for final publication by the author. Section 6 of article IV, and section 7 of article V were written solely by the author. The implementation of the HyperSoft system's back-end components, described in more detail in (Koskinen, 1996a; 1997), also represents the independent work of the author. The front-end components of the HyperSoft system were implemented by Nieminen (1996). The Articles I-IV appeared in the author's licentiate thesis (Koskinen, 1996b). This doctoral dissertation is an extension of that previous work. The new work includes the Article IV in revised and published form, Articles V, VI, VII, and the report (Koskinen 1997), which describes the design of the HyperSoft system v. 1.0's back-end components, the final implementation of the HyperSoft system (Koskinen *et al.*, 1997), and the other recent manuscripts. The research results have also been presented in the form of position papers (Salminen *et al.* 1994b; Koskinen *et al.* 1994b; Koskinen 1995).

5 DISCUSSION ON RESEARCH DIRECTIONS

There are many interesting options for further research. This chapter aims at proposing a further research agenda and describes some of the possible extensions and enhancements related to different aspects of HyperSoft. The current HyperSoft system is focused on supporting program comprehension and software maintenance by providing various THASs over source programs written in C. Further refined specialized support could be tailored for these purposes by introducing new THAS types, new view types, or by integrating new supplementary techniques into HyperSoft, see Figure 11.

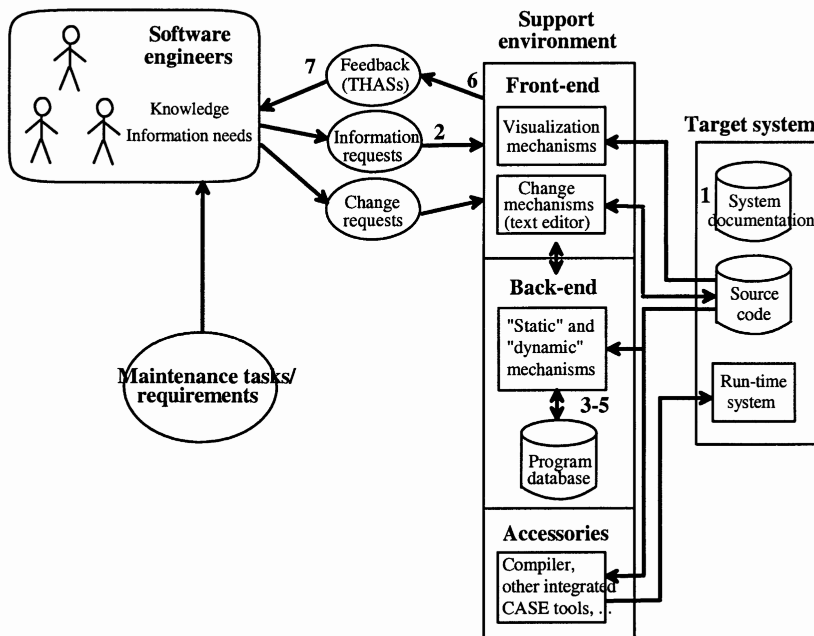


FIGURE 11 THAS-based maintenance support

The central elements (system components, databases, data sets, data flows, and actors) in supporting software maintenance through our approach are depicted in Figure 11. The target system is the one which is under maintenance. The support environment provides relevant information for its users (software engineers or maintainers). It consists of the programs which are needed to view, browse and change the source code of the target systems and other integrated support mechanisms. The figure shows the main data flows between the components by arrows. Data flows are represented at abstract level (internal dataflows of the HyperSoft system were depicted in Figure 2). The numerals (x) in the figure refer to the sections (5.x) listed below. Discussed issues include 1) model extensions of HyperSoft, 2) options to specify more complex information requests, 3) optimizations in central areas, 4) some remarks on introducing new THAS types, 5) language extensions, 6) visual representations of THASs, and 7) evaluations of the features of HyperSoft in various ways.

5.1 Model extensions

We have focused on improving the ability to obtain information from the source code on the basis of static analysis. The HyperSoft model underlying our approach, however, allows for the extension of access structures to cover such documentation, which is structured based on underlying grammar, as well, and for the use of dynamic analysis in identifying the required program parts. Thus, the numeral 1 in Figure 11 is located near the system documentation database. From the theoretical point of view, this model extension issue (in relation to the input information type) is the most important area of further research. One problem is the typical unavailability of structured documentation in case of legacy systems. The model extension issue is also discussed in Article VI.

Experiments on automatically transforming non-program text into hypertext are reported by Agosti and Allan (1997). Also, for example, parts of the documentation associated with the source code could automatically be transformed into hypertext. Documents may be marked-up by using, *e.g.* SGML (Goldfarb, 1990; Cowan *et al.*, 1994), or HTML (Musciano & Kennedy, 1998). In addition, comments (Riecken *et al.*, 1991) embedded within the source code could be structured systematically and thus handled within the model.

Program text can be considered as a special case of text, since it is not solely targeted at a human reader, but is also used to control the computer. This means that unlike most other texts, program text contains information which cannot be obtained without executing the program. Because of this, some of the relevant THAS types cannot be formed without information which is available only during the run-time of the program. Examples of these are dynamic program slices (Kamkar, 1993).

5.2 Query mechanisms

Most of the empirically verified information needs of professional C programmers can be satisfied with simple mechanisms such as those currently provided in HyperSoft, or by using QBE (Query By Example) (Zloof, 1977). If complex or very detailed information requests need to be formed, a dedicated (textual) query language is called for. For example, Paul and Prakash (1996) have presented a source code algebra which is stated to combine high expressive power with a flexible query paradigm. One of the related general problems, however, is the complexity of typical queries and required comprehension effort (Chan *et al.*, 1997). If query mechanisms (such as those represented in Appendix 1; part 7) were to be implemented in HyperSoft, they would probably mostly benefit those maintainers who are very capable in formalizing their information needs and have the necessary patience. Since forming a query interrupts the established way in which most people perform their programming and maintenance tasks, it is not clear whether such mechanisms would actually be called upon.

5.3 Technical optimizations

The main needs for optimizations lie in the areas of improving the efficiency of the interprocedural slicing and of compressing/pruning the necessary static structures, most notably parse trees. There is a clear tradeoff between whether to form a slice precisely or fast. The need to be able to tradeoff between time and precision in relation to whole-program analysis tools - and program slicing in particular - has been addressed by Atkinson and Griswold (1996). They recommend features for demand-driven computation, discarding, persistent storage of important data-structures, precision control, and termination control. HyperSoft includes some mechanisms similar to these, as noted in Section 3.3.

THASs are relatively small compared to parse trees. Since sufficient mass storage space is nowadays available at tolerable costs, the relatively large size of parse trees has mostly only indirect importance in slowing down the operation of the system, especially in case of interprocedural slicing. Because parse trees comprise most of the static program database (90%), performance may be optimized by compressing them. Regardless of the way that the static program database is formed, it may be compressed by means of standard packages, such as *pkzip*, *gnu-zip*, *compress*, or *compact*, which yield about 55-70 % compression rates (Katajainen & Mäkinen, 1990). Our simulations with the HyperSoft system and the example chess program (described in Section 3.4) show that decompression would slow down THAS formation by about 50 %. Hence, a tradeoff exists between whether to save disk space or to generate THASs faster.

Probably the most compact representation holding the same information as abstract syntax trees is the so-called production tree (Waddle, 1990). These structures are relatively simple to form and take only about 1/3 of the space of

abstract syntax trees. Special methods of coding the parsing process to be used as an aid in compressing syntactical information has also been suggested. These sorts of methods (Lelewer & Hirschberg, 1987; Cameron, 1988; Katajainen *et al.*, 1986; Peltola & Tarhio, 1991) yield at best a compression rate of about 85% (compared to the original files), see also, *e.g.* (Gil & Itai, 1999) for packing tree structures. The methods may rely, for example, on coding the program structure on the basis of the applied productions. These kinds of special methods are not used in HyperSoft. It should be noted that even though some of the methods yield high compression rates, they often also lead to long (de)compression times. If there is a tradeoff, time optimization should be preferred over space optimization in a system such as HyperSoft.

One way of improving slicing efficiency is to store the needed program information in the form of program (or system) dependency graphs (Horwitz & Reps, 1992) or combined C graphs (Kinloch & Munro, 1994). It is, however, not clear whether using program dependency graphs (as intermediate representations) would be an optimal solution for HyperSoft, since in HyperSoft a very large amount of program information has to be stored in order to support a sufficiently versatile THAS set, *cf.* Horwitz *et al.* (1988). If graphs of this kind are not used, the efficiency of (downward) slicing could nonetheless be improved by applying the so-called *in-out* (*definition-usage*) *sets* (Kamkar 1993). If the predefined in-out set for a specific function is available, it would only be necessary to analyze the function once for each different content of the set of relevant variables when entering that function (see Article III for the details of downward slicing and the set of relevant variables). The in-out sets would be determined during a batch process for all functions. The current rules for determining the contents of the relevant variable set within the HyperSoft system are described in Koskinen (1997, pp. 106-107). Our experiences with the slicing features of HyperSoft are such that there appears to be a general need for features such as the incremental generation of partial slices, for both cognitive and efficiency reasons. There also exist some additional options for optimizations which have been reported and classified in Koskinen (1996b, pp. 25-27).

5.4 New access structures

The introduction of new THAS types to HyperSoft is straightforward, as only a new THAS generation function needs to be implemented. Experiences with the current simple THAS types support the hypothesis as to their usefulness. The more complex THASs may be composed of elementary structures. Moreover, by using set operations, *cf.* Garg (1988), more refined structures can be produced. Our experiences with slicing suggest that it is useful to supplement it with less complete THASs, which can be generated more or less instantaneously. The simpler structures can be used to support the process in which the user fixes the seed for the complete slicing analysis. In Article VI we analyze the information needs which constitute the most important base for THAS types. This analysis shows that the most useful additional THAS types for inclusion in

HyperSoft are probably domain concept descriptions and lists of browsed locations. Some suggestions are also provided in Koskinen (1996b). Basically, the possibilities for extending the THAS set by introducing new THAS types based on, for example, the algorithms surveyed in Appendix 1, are extensive.

5.5 Language extensions

The explicit technical separation of the components in HyperSoft makes it relatively straightforward to extend the system with new languages. A new language may be supported by implementing a new analyzer component to form the static program database. The interface component focuses on the visualization based on the THASs which it gets as input from the generator component. If the relevant items of the new language can be described in the form which is currently used to store the static information, the THAS generator may be updated by simply extending its procedures. SQL extensions have already been completed in the form of a spin-off project within one of our partner enterprises (Suominen, 1997).

Versatile (multi-) language support is important. Possible object-oriented extensions of the HyperSoft THAS set have been considered in Tuovinen (1995). The possibilities of the analysis of C++ (Stroustrup, 1986; 1993) were considered during the project, including the options of using ANTRL (Parr & Quong 1995) or AnaGram (Parsifal, 1993) as the basis of parsing. Novel solutions would, however, be needed to split the C++ grammar into non-ambigous parts and to construct separate parsers for the sub-grammars. Although C++ is "merely" an extension of the C language, the analysis is problematic. The main reasons for this include the facts that it is difficult or impossible to form an *LALR(1)* grammar for C++, *cf. e.g.* (Parr & Quong, 1996), and C++ is very liberal in its syntax, making the identification of the semantic meaning of the symbols difficult. In C++ it is, *e.g.*, possible to have different symbols with the same name declared in a scope because the identity of a symbol is determined by its name and type. Unlike in C, declarations can be intermixed with statements, *cf.* Knapen *et al.* (1999). Dynamic binding and polymorphism introduce their own peculiarities, making the static analysis approach less promising, see *e.g.* Tonella *et al.* (1997). Reverse engineering tools for C++ has been proposed by Grass (1992), Chen, X. *et al.* (1996), Linos and Courtois (1996), and Yueh and Low (1997). The needs of the partner enterprises would also include support for COBOL; *cf.* The COBOL Center (1999) and even assembly language; *cf.* Chen, S. *et al.* (1990).

5.6 View enhancements

Software visualization requires human-computer interaction and use of graphical interfaces which are discussed, *e.g.* by Shneiderman (1992). There exists a

wide range of software visualization systems (Catalin-Roman & Fox, 1993; Vilela *et al.*, 1997). Our experiments support the hypothesis of the importance of supplementing overviews. The need for visual support for program slicing is also noted by Gallagher (1997). Various graphical views are provided by the HyperSoft system to complement the hypertextual representation of THASs (Nieminen, 1996).

Because slices can be very large, they may need to be sieved. The most useful information within a slice is probably concentrated on the most immediate calling levels of the original context function. The observations of Atkinson and Griswold (1996) support this hypothesis. Therefore, even if the slice is formed completely, it would be useful to be able to view only the *n* most immediate calling levels, first. Slicing structures can also be viewed in various abstract forms so that a general view of module cohesion and coupling can be formed, *cf.* the implemented module dependency views and (Ott & Thuss, 1989). HyperSoft is well suited to these kind of purposes, since the generation of THASs is separated from their representation to the user, and THASs can be augmented with the necessary information. Different ways to visually express THASs in HyperSoft have been suggested by Sillanpää (1997). One option is the use of distortion-oriented visualizations to further improve focusing on relevant items (Leung & Apperley, 1994). This could be useful, especially in the case of large slices.

5.7 Empirical studies

Possible evaluations include empirical end-user evaluations of the system usability. Further possible evaluations of HyperSoft should, in the ideal case, focus on professional maintainers performing real maintenance including program modifications. As noted, we have already performed data gathering in professional settings (Article IV). Truely reliable empirical evaluations with professionals performing real maintenance tasks would encounter many organizational and financial problems. These are the reasons why we decided to conduct laboratory experiments instead.

Another strategy is to continue performing controlled laboratory experiments evaluating different aspects of hypertext support by using computer science students as subjects for this purpose. There exist good methodological summaries. Experimentation is discussed from the methodological point of view, for instance, by Pfleeger (1997); Tichy (1998); Zelkowitz and Wallace (1998) and empirical studies of programmers by Shneiderman (1986). Reverse engineering tools have been evaluated, for example, by Storey *et al.* (1996) and user interface solutions have been compared by Jeffries *et al.* (1991). Analyses of the empirical software studies (Shneiderman, 1986) and theoretical frameworks (Haworth *et al.*, 1992) for software maintenance studies provide information about the aspects which could be studied and the methodologies which can be used by utilizing the existing HyperSoft system, according to our original intention of performing comparative empirical evaluations.

CONCLUSION

This dissertation represented a new approach to the support of software maintenance activities via enhancing information retrieval. It should be noted that software maintenance accounts for over half of the resources spent on information systems development. The approach combines automated (static program) analysis and transient hypertextual representation. The approach focuses on supporting the localization of relevant information from the source code.

In order to make changes to software without introducing side-effects, the relevant program parts need to be comprehended. During program comprehension, the programmer tries to form a mental model about the structure, operation, and purpose of the relevant software components. Such an understanding can be gained through viewing the source code and documentation. If adequate documentation does not exist, the information has to be "reverse-engineered" from the source code. The understanding is often hampered by the fact that the information needed is dispersed throughout the source code. The process of extracting the information can, however, be supported by various reverse engineering tools. During program comprehension efforts, the programmer typically browses the source code back and forth trying to find meaningful program segments and interdependencies between them. The process is clearly such that it can be supported by viewing program text as hypertext.

We have developed the HyperSoft model, which makes it possible to systematically support software maintenance and program comprehension processes by representing the program text as hypertext to the maintainer. The hypertext is formed automatically by analyzing the source programs and by extracting useful information which is stored in a program database. The transient hypertextual structures which are created automatically, on user request, on the basis of their situation-dependent information needs, are called THASs. THASs are created based on the information which has been stored into the program database. The use of transient structures instead of static structures eliminates the elaborate manual linking which is typical of conventional hypertext systems, helps to ensure the validity of the hypertextual structures, and helps to reduce the amount of static information needed. THASs are composed of the relevant program parts linked together on the basis of existing program

dependencies. The existing literature on techniques and algorithms for extracting the required program part and program dependency information have been surveyed. The development of a relational classification of program dependencies serves as a basis for the systematic planning of well-formed THAS types.

The HyperSoft system - an experimental software maintenance support tool - is used to demonstrate the implementability and convenience of the ideas presented and to experiment with various THASs. The system has been implemented within a project guided by an industrial steering group. The target language (C), and the currently provided THAS types have been selected by the group. The THAS types include occurrence lists of variables, functions, and user-defined types, forward and backward calling dependency structures, intraprocedural backward slices, and interprocedural forward slices. The system helps in focusing the maintainer's attention on those program parts which are relevant to the current maintenance task, and in navigating among those parts by following the hypertextual links generated by the system. The various graphical views can also be used to achieve the same purpose since they are linked to the original program text.

We have evaluated the approach in three ways. First, by small-scale testing in the partner companies, the results of which have suggested the usefulness of the approach. Second, by comparing the capabilities offered by HyperSoft to the information needs of software maintainers as revealed in a series of earlier empirical studies. And third, by two series of empirical experiments. The data on information needs derived from the earlier studies suggests that the THAS types, which are currently implemented in the HyperSoft system, provide a good coverage of support for the kinds of information typically needed, which can be produced via static program analysis. We conducted two separate experiments with computer science students as subjects. The outcome of these experiments clearly supported our hypothesis regarding the usefulness of the approach in typical information retrieval tasks, as compared to the conventional information-seeking capabilities of a widely used compiler environment. We have modelled the effects on task performance in multiple ways, as well as gathered subjective information related to the usability of the approach.

Finally, we have discussed some possible further research topics. We have presented some ideas for extending the scope of application of the approach, and ways to improve the formation of the THASs and to extend the set of supported THAS types. Most importantly, there are good possibilities for extensions and for the introduction of new interesting THAS types in a straightforward way. Since the support of multiple THASs during a session potentially requires great amount of statically stored information, some methods of dealing with this problem have been proposed. Parse tree abstraction and pruning are among the best available ways of reducing the amount of static information needed and improving the efficiency of THAS generation. The most effective way of applying HyperSoft ideas in practice would probably be via an integrated CASE environment. Programmers and maintainers constantly use compilers and editors in their work. Those tools have to deal with many of the same problems as HyperSoft, most importantly, efficient automated source code analysis and informative representation of the source code to the users.

REFERENCES[4]

Agosti, M. & Allan, J. 1997. Introduction to the special issue on methods and tools for the automatic construction of hypertext. *Information Processing & Management* 33 (2), 129-131.

Agosti, M., Crestani, F. & Melucci, M. 1996. Design and implementation of a tool for the automatic construction of hypertexts for information retrieval. *Information Processing and Management* 32 (4), 459-476.

Agosti, M., Crestani, F. & Melucci, M. 1997. On the use of information retrieval techniques for the automatic construction of hypertext. *Information Processing and Management* 33 (2), 133-144.

Agosti, M., Melucci, M. & Crestani, F. 1995. Automatic authoring and construction of hypertext for information retrieval. *ACM Multimedia Systems* 3 (1), 15-24.

Agrawal, H. 1994. On slicing program with jump statements. *ACM SIGPLAN Notices* 29 (6), 302-312. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'94).*

Aho, A.V. 1990. Algorithms for finding patterns in strings. In J. van Leeuwen (Ed.) *Handbook of Theoretical Computer Science (Vol. A).* Elsevier & MIT Press, 255-300.

Aho, A.V. & Johnson, S. 1974. LR parsing. *ACM Computing Surveys* 6 (2), 99-124.

Aho, A.V., Sethi, R., & Ullman, J. 1986. *Compilers - Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley.

Ajila, S. 1995. Software maintenance: an approach to impact analysis of objects change. *Software - Practice and Experience* 25 (10), 1155-1181.

Allan, J. 1995. *Automatic Hypertext Construction*. Ithaca, NY: Department of Computer Science, Cornell University (Diss.).

Allan, J. 1996. Automatic hypertext link typing. In *Proc. 7th ACM Conf. Hypertext*, 42-52. ACM Press.

Allan, J. 1997. Building hypertext using information retrieval. *Information Processing & Management* 33 (2), 145-159.

---

4 The acronyms used of the journal names are given at the end of the reference list.

Al-Zoubi, R. & Prakash, A. 1995. Program view generation and change analysis using attributed dependency graphs. *J. Software Maintenance: Research and Practice* 7 (4), 239-261.

Arunachalam, V. & Sasso, W. 1996. Cognitive processes in program comprehension: an empirical analysis in the context of software reengineering. *The J. Systems and Software* 34 (3), 177-189.

Ashley, M. & Bybvig, R. 1998. A practical and flexible flow analysis for higher-order languages. *ACM TOPLAS* 20 (4), 845-868.

Atkinson, D. & Griswold, W. 1996. The design of whole-program analysis tools. In M. Kavanaugh (Ed. production) *Proc. 18th Int. Conf. Software Engineering (ICSE'96)*. IEEE Computer Soc., 16-27.

Atkinson, D. & Griswold, W. 1998. Effective whole-program analysis in the presence of pointers. *ACM SIGSOFT Software Engineering Notes* 23 (6), 46-55. *ACM SIGSOFT 6th Int. Symp. Foundations of Software Engineering (FSE'6)*.

Baeza-Yates, R. & Gonnet, G. 1992. A new approach to text searching. *CACM* 35 (10), 74-82.

Ball, T. & Horwitz, S. 1992. Slicing programs with arbitrary control-flow. In P. Fritzson (Ed.) *LNCS* 749. Springer-Verlag, 206-222. *First Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'92)*.

Banker, R., Datar, S. & Kemerer, C. 1991. A model to evaluate variables impacting the productivity of software maintenance projects. *Management Science* 37 (1), 1-18.

Baratta-Perez, G., Conn, R., Finnell, C. & Walsh, T. 1994. Ada system dependency analyzer tool. *Computer* 27 (1), 49-55.

Beck, J. & Eichmann, D. 1993. Program and interface slicing for reverse engineering. In E. Straub (Ed. production) *Proc. 15th Int. Conf. Software Engineering (ICSE'93)*. Los Alamitos, CA: IEEE Computer Soc., 509-518.

Beeri, C. & Kornatzky, Y. 1990. A logical query language for hypertext systems. In *Hypertext: Concepts, Systems, and Applications. Proc. European Conf. Hypertext'90 (ECHT'90)*. Cambridge: Cambridge Univ. Press., 67-80.

Belkin, N. & Croft, W.B. 1992. Information filtering and information retrieval: two sides of the same coin? *CACM* 35 (12), 29-38.

Benedusi, P., Cimitile, A. & DeCarlini, U. 1989. A reverse engineering methodology to reconstruct hierarchical data flow diagrams for software maintenance. In *Proc. Int. Conf. Software Maintenance (ICSM'89)*. IEEE Computer Soc., 180-191.

Bennett, K. 1995. Legacy systems: coping with success. *IEEE Software* 12 (1), 19-22.

Bergeretti, J.-F. & Carre, B. 1985. Information-flow and data-flow analysis of while-programs. *ACM TOPLAS* 7 (1), 37-61.

Bertino, E., Rabitti, F. & Gibbs, S. 1988. Query processing in a multi-media environment. *ACM TOOIS* 6 (1), 1-41.

Bieman, J. & Ott, L. 1994. Measuring functional cohesion. *IEEE TOSE* 20 (8), 644-657.

Bigelow, J. 1988. Hypertext and CASE. *IEEE Software* 5 (2), 23-27.

Bigelow, J. & Riley, V. 1987. Manipulating source code in dynamic design. In S. Weiss & M. Shwartz (Ed.) *Proc. Hypertext'87 (1st ACM Conf. Hypertext).* ACM Press.

Binkley, D. 1998. The application of program slicing to regression testing. *Information and Software Technology* 40 (11/12), 583-594.

Binkley, D. & Gallagher, K. 1996. Program slicing. *AIC* 43, 1-50.

Bodik, R. & Gupta, R. 1997. Partial dead code elimination using slicing transformations. *ACM SIGPLAN Notices* 32 (5), 159-170. *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'97).*

Boehm, B. 1988a. A spiral model of software development and enhancement. *Computer* 21 (5), 61-72.

Boehm, B. 1988b. Understanding and controlling software costs. *IEEE TOSE* 14 (10), 1462-1477.

Bohner, S. & Arnold, R. 1996. *Software Change Impact Analysis.* IEEE Computer Soc.

Booch, G., Rumbaugh, J. & Jacobson, I. 1999. *The Unified Modeling Language User Guide.* Reading, MA: Addison-Wesley.

Brade, K., Guzdial, M., Steckel, M. & Soloway, E. 1994. Whorf: a hypertext tool for software maintenance. *Int. J. Software Engineering and Knowledge Engineering* 4 (1), 1-16.

Briand, L., Devanbu, P. & Melo, W. 1997. An investigation into coupling measures for C++. In *Proc. 19th Int. Conf. Software Engineering (ICSE'97).* New York: ACM Press, 412-421.

Brooks, F.P. Jr. 1987. No silver bullet - essence and accidents of software engineering. *Computer* 20 (4), 10-19.

Brooks, R. 1977. Towards a theory of the cognitive processes in computer programming. *IJMMS* 9, 737-751.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *IJMMS* 18 (6), 543-554.

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM TOPLAS* 12 (3), 341-395.

Burke, M. & Ryder, B. 1990. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE TOSE* 16 (7), 723-728.

Burkowski, F. 1992. An algebra for hierarchically organized text-dominated databases. *Information Processing & Management* 28 (3), 333-348.

Burnstein, I & Roberson, K. 1997. Automated chunking to support program comprehension. In P. Storms (Ed. production) *Proc. 5th Int. Workshop on Program Comprehension (IWPC'97).* Los Alamitos, CA: IEEE Computer Soc., 40-49.

CACM. 1994. *CACM* 37 (5). Theme issue on reverse engineering.

Caldiera, G. & Basili, V. 1991. Identifying and qualifying reusable software components. *Computer* 24 (2), 61-70.

Cameron, R. 1988. Source encoding using syntactic information source models. *IEEE Trans. Inform. Theory* 34 (4), 843-850.

Canfora, G. & Cimitile, A. 1992. Reverse-engineering and intermodular data flow: a theoretical approach. *J. Software Maintenance: Research and Practice* 4 (1), 37-59.

Canfora, G., Cimitile, A. & Munro, M. 1993. A reverse engineering method for identifying reusable abstract data types. In *Proc. 1st Working Conf. Reverse Engineering (WCRE'93)*, 73-82.

Canfora, G., Cimitile, A. & Munro, M. 1996a. An improved algorithm for identifying objects in code. *Software - Practice and Experience* 26 (1), 25-48.

Canfora, G., Cimitile, A., Munro, M. & Taylor, C. 1996b. Extracting abstract data types from C programs: a case study. In *Proc. Int. Conf. Software Maintenance (ICSM'96)*, 200-209.

Carmel, E., McHenry, W. & Cohen, Y. 1989. Building large, dynamic hypertexts: how do we link intelligently?. *J. Management Information Systems* 6 (2), 33-50.

Catalin-Roman, G. & Fox, K. 1993. A taxonomy of program visualization systems. *Computer* 26 (12), 11-24.

Chan, H., Wei, K. & Siau, K. 1997. A system for query comprehension. *Information and Software Technology* 39 (3), 141-148.

Chen, C. & Rada, R. 1996. Interacting with hypertext: a meta-analysis of experimental studies. *Human-Computer Interaction* 11 (2), 125-156.

Chen, S., Heisler, K., Tsai, W., Chen, X. & Leung, E. 1990. A model for assembly program maintenance. *Software Maintenance* 2, 3-32.

Chen, T. & Cheung, Y. 1993. Dynamic program dicing. In *Proc. Int. Conf. Software Maintenance - 1993 (ICSM'93)*, 378-385.

Chen, T. & Low, C. 1997. Error detection in C++ through dynamic data flow analysis. *Software - Concepts and Tools* 18 (1), 1-13.

Chen, X., Tsai, W.-T., Huang, H., Poonawala, M., Rayadurgam, S. & Wang, Y. 1996. Omega: an integrated environment for C++ program maintenance. In *Proc. Int. Conf. Software Maintenance (ICSM'96)*, 114-123.

Chen, Y.-F., Gansner, E. & Koutsofios, E. 1998. A C++ data model supporting reachability analysis and dead code detection. *IEEE TOSE* 24 (9), 682-694.

Chen, Y.-F., Nishimoto, M. & Ramamoorthy, C. 1990. The C information abstraction system. *IEEE TOSE* 16 (3), 325-334.

Chikofsky, E. & Cross, J. H. II. 1990. Reverse engineering and design recovery: a taxonomy. *IEEE Software* 7 (1), 13-17.

Choi, J.-D., Cytron, R. & Ferrante, J. 1991a. Automatic construction of sparse data flow evaluation graphs. In *Conf. Record of the 18th ACM Symp. Principles of Programming Languages (POPL'91)*. ACM Press, 55-66.

Choi, J.-D., Miller, B. & Netzer, R. 1991b. Techniques for debugging parallel programs with flowback analysis. *ACM TOPLAS* 13 (4), 491-530.

Choi, J.-D. & Ferrante, J. 1994. Static slicing in the presence of goto statements. *ACM TOPLAS* 16 (4), 1097-1113.

Cimitile, A. & De Carlini, U. 1991. Reverse engineering: algorithms for program graph production. *Software - Practice and Experience* 21 (5), 519-537.

Cimitile, A., De Lucia, A., Di Lucca, G. & Fasolino, A.R. 1999. Identifying objects in legacy systems using design metrics. *The J. Systems and Software* 44 (3), 199-212.

Clarke, L., Cormack, G. & Burkowski, F. 1995. An algebra for structured text search and a framework for its implementation. *The Computer J.* 38 (1), 43-56.

Cleary, C. & Bareiss, R. 1996. Practical methods for automatically generating typed links. In *Proc. 7th ACM Conf. Hypertext*. ACM Press, 31-41.

Computer. 1999. *Computer* 32 (8). Theme issue on data mining.

Conklin, J. 1987. Hypertext: an introduction and survey. *Computer* 20 (9), 17-41.

Conklin, J. & Begeman, M. 1989. gIBIS: a tool for all reasons. *JASIS* 40, 200-213.

Consens, M., Mendelzon, A. & Ryman, A. 1992. Visualizing and querying software structures. In *Proc. 14th Int. Conf. Software Engineering (ICSE'92)*. New York: ACM Press, 138-156.

Corbi, T. 1989. Program understanding: challenge for the 1990s. *IBM Systems J.* 28 (2), 294-306.

Corritore, C. & Wiedenbeck, S. 1999. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *IJHCS* 50 (1), 61-84.

Cowan, D., German, D., Lucena, C. & von Staa, A. 1994. Enhancing code for readability and comprehension using SGML. In *Proc. Int. Conf. Software Maintenance (ICSM'94)*, 181-190.

Creech, M., Freeze, D. & Griss, M. 1991. Using hypertext in selecting reusable software components. In J. Walker (Ed.) *Proc. Hypertext'91: 3rd ACM Conf. Hypertext*. New York: ACM Press.

Cross, J. H. II, Chikofsky, E., May, C.H. Jr. 1992. Reverse engineering. *AIC* 35, 199-254.

Cunniff, C. & Taylor, C. 1987. Representation form effects on novice's program comprehension. In Olson, G., Sheppard, S. & Soloway, E. (Ed.) *Proc. 2nd Workshop Empirical Studies of Programmers (ESP'87)*. Norwood, NJ: Ablex.

Cutillo, F., Fiore, R. & Visaggio, G. 1993. Identification and extraction of domain independent components in large programs. In *Proc. 1st Working Conf. Reverse Engineering (WCRE'93)*, 83-92.

Cybulski, J. & Reed, K. 1992. A hypertext-based software-engineering environment. *IEEE Software* 9 (2), 62-68.

Cytron, R., Ferrante, J., Rosen, B., Wegman, M. & Zadeck, F. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS* 13 (4), 451-490.

Darnell, P. & Margolis, P. 1991. *C: A Software Engineering Approach*. New York: Springer.

Date, C. 1987. *A Guide to the SQL Standard*. Reading, MA: Addison-Wesley.

Davies, S. 1990. The nature and development of programming plans. *IJMMS* 32 (4), 461-481.

Davis, A. 1995. Software prototyping. *AIC* 40, 39-63.

Dervin, B. & Nilan, M. 1986. Information needs and uses. In M. Williams (Ed.) *Ann. Review of Information Science and Technology (ARIST)* 21. Knowledge Industry Publications, 3-33.

van Deursen, A., Woods, S. & Quilici, A. 1997. Program plan recognition for Year 2000 tools. In P. Storms (Ed. production) *Proc. 4th Working Conf. Reverse Engineering (WCRE'97)*. Los Alamitos, CA: IEEE Computer Soc., 124-135.

Duncan, I. & Robson, D. 1996. An exploratory study of common coding faults in C programs. *J. Software Maintenance: Research and Practice* 8 (4), 241-256.

Dunlop, M. & van Rijsbergen, C. 1993. Hypermedia and free text retrieval. *Information Processing & Management* 29 (3), 287-298.

Edelstein, D. 1993. Report on the IEEE STD 1219 - 1993 - Standard for Software Maintenance. *ACM SIGSOFT Software Engineering Notes* 18 (4), p. 94.

Eisenstadt, M. 1997. My hairiest bug war stories. *CACM* 40 (4), 30-37.

Eyre-Todd, R. 1993. The detection of dangling references in C++ programs. *ACM LOPLAS* 2, 127-134.

Faustle, S., Fugini, M.G. & Damiani, E. 1996. Retrieval of reusable components using functional similarity. *Software - Practice and Experience* 26 (5), 491-530.

Field, J., Ramalingam, G. & Tip, F. 1995. Parametric program slicing. In *Proc. 22nd ACM SIGPLAN/SIGACT Symp. Principles of Programming Languages (POPL'95)*. ACM Press, 379-392.

Fiutem, R., Tonella, P., Antoniol, G. & Merlo, E. 1996. A cliche-based environment to support architectural reverse engineering. In *Proc. Int. Conf. Software Maintenance (ICSM'96)*, 319-328.

Fiutem, R., Tonella, P., Antoniol, G. & Merlo, E. 1999. Points-to analysis for program understanding. *The J. Systems and Software* 44 (3), 213-228.

Fletton, N. & Munro, M. 1988. Redocumenting software systems using hypertext technologies. In *IEEE Int. Conf. Software Maintenance 88 (ICSM'88)*, 54-59.

Fraisse, S. 1997. A task driven design method and its associated tool for automatically generating hypertexts. In M. Bernstein, L. Corr & K. Østerbye (Ed.) *The 8th ACM Conf. Hypertext - Hypertext'97*. ACM Press, 234-236.

French, J., Knight, J. & Powell, A. 1997. Applying hypertext structures to software documentation. *Information Processing & Management* 33 (2), 219-231.

Frisse, M. & Cousins, S. 1992. Models for hypertext. *JASIS* 43 (2), 183-191.

Fuggetta, A. 1993. A classification of CASE technology. *Computer* 26 (12), 25-38.

Fyson, M. & Boldyreff, C. 1998. Using application understanding to support impact analysis. *J. Software Maintenance: Research and Practice* 10 (2), 93-110.

Gallagher, K.B. 1992. Evaluating the Surgeon's Assistant: results of a pilot study. In *Proc. Int. Conf. Software Maintenance - 1992 (ICSM'92)*, 236-244.

Gallagher, K.B. 1997. Visual impact analysis. In *Proc. Int. Conf. Software Maintenance (ICSM'96)*. IEEE Computer Soc., 52-58.

Gannod, G. & Cheng, B. 1996. Using informal and formal techniques for the reverse engineering of C programs. In *Proc. Int. Conf. Software Maintenance (ICSM'96)*, 265-274.

Garg, P. 1988. Abstraction mechanisms in hypertext. *CACM* 31 (7), 862-870.

Garg, P. 1989. *Information Management in Software Engineering: A Hypertext Based Approach*. Los Angeles: University of Southern California (Diss.).

Garg, P. & Scacchi, W. 1989. Ishys: designing an Intelligent Software Hypertext System. *IEEE Expert* 4 (3), 52-63.

Garg, P. & Scacchi, W. 1990. A hypertext system to manage software lifecycle documents. *IEEE Software* 7 (3), 90-98.

Gellenbeck, E. & Cook, C. 1991. An investigation of procedure and variable names as beacons during program comprehension. In J. Koenemann-Belliveau, T. Moher & S. Robertson (Ed.) *Empirical Studies of Programmers: 4th Workshop (ESP'91)*. Norwood, NJ: Ablex, 65-81.

Gil, J. & Itai, A. 1999. How to pack trees. *J. Algorithms* 32 (2), 108-132.

Glass, R. & Vessey, I. 1995. Contemporary application-domain taxonomies. *IEEE Software* 12 (4), 63-76.

Goldfarb, C. 1990. *The SGML Handbook*. Y. Rubinsky (Ed.). Oxford: Oxford Univ. Press.

Gopal, R. 1991. Dynamic program slicing based on dependence relations. In *Proc. IEEE Int. Conf. Software Maintenance (ICSM'91)*, 191-200.

Grass, J. 1992. Object-oriented design archaeology with CIA++. *Computing Systems* 5 (1), 5-67.

Griswold, W., Atkinson, D. & McCurdy, C. 1996. Fast, flexible syntactic pattern matching and processing. In A. Cimitile & H. Muller (Ed.) *Proc. 4th Int. Workshop on Program Comprehension (IWPC'96)*. IEEE Computer Soc., 144-153.

Griswold, W. & Notkin, D. 1993. Automated assistance for program restructuring. *ACM TOPLAS* 2 (3), 228-269.

Griswold, W. & Notkin, D. 1995. Architectural tradeoffs for a meaning-preserving program restructuring tool. *IEEE TOSE* 21 (4), 275-287.

Grove, D., DeFouw, G., Dean, J. & Chambers, C. 1997. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices* 32 (10), 108-124. *Proc. OOPSLA'97*.

Grønbaek, K., Hem, J., Madsen, O. & Sloth, L. 1994. Systems: a Dexter-based architecture. *CACM* 37 (2), 65-74.

Grønbaek, K. & Trigg, R. 1994. Design issues for a Dexter-based hypermedia system. *CACM* 37 (2), 40-49.

Gugerty, L. & Olson, G. 1986. Comprehension differences in debugging by skilled and novice programmers. In E. Soloway & S. Iyengar (Ed.) *Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86)*. Norwood, NJ: Ablex.

Gunter, C., Mitchell, J. & Notkin, D. 1996. Strategic directions in software engineering and programming languages. *ACM Computing Surveys* 28 (4), 727-737.

Gupta, A. 1997. Program understanding using program slivers: an experience report. In *Proc. Int. Conf. Software Maintenance (ICSM'97)*. IEEE Computer Soc., 66-71.

Gupta, R., Harrold, M. & Soffa, M. 1996. Program slicing-based regression testing techniques. *Software Testing, Verification and Reliability* 6 (2), 83-111.

Hagemeister, J., Lowther, B., Oman, P., Yu, X. & Zhu, W. 1992. An annotated bibliography on software maintenance. *ACM SIGSOFT Software Engineering Notes* 17 (2), 79-84.

Halasz, F. 1988. Reflections on Notecards: seven issues for the next generation of hypermedia systems. *CACM* 31 (7), 836-855.

Halasz, F. & Schwartz, M. 1994. The dexter hypertext reference model. *CACM* 37 (2), 29-39.

Hall, M. & Kennedy, K. 1992. Efficient call graph analysis. *ACM LOPLAS* 1 (3), 227-242.

Harman, M. & Danicic, S. 1995. Using program slicing to simplify testing. *Software Testing, Verification and Reliability* 5, 143-162.

Harman, M. & Danicic, S. 1998. A new algorithm for slicing unstructured programs. *J. Software Maintenance: Research and Practice* 10 (6), 415-442.

Harman, M. & Gallagher, K.B. 1998. Program slicing. *Information and Software Technology* 40 (11/12), 577-582.

Harris, D., Reubenstein, H. & Yeh, A. 1995. Recognizers for extracting architectural features from source code. In *Proc. 2nd Working Conf. Reverse Engineering (WCRE'95)*. IEEE Computer Soc., 252-261.

Harrold, M. & Malloy, B. 1993. A unified interprocedural program representation for a maintenance environment. *IEEE TOSE* 19 (6), 584-593.

Harrold, M., Malloy, B. & Rothermel, G. 1993. Efficient construction of program dependence graphs. In *Proc. 1993 Int. Symp. Software Testing and Analysis (ISSTA'93)*, 160-170.

Harrold, M., Rothermel, G. & Sinha, S. 1998. Computation of interprocedural control dependencies. In M. Young (Ed.) *Proc. ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA'98)*. ACM Press, 11-20.

Harrold, M., Soffa, M. 1990. Computation of interprocedural definition and use dependencies. In *Proc. IEEE Comput. Soc. 1990 Int. Conf. Comput. Languages*. IEEE Computer Soc., 297-306.

Hart, J. & Pizzarello, A. 1996. A scalable, automated process for Year 2000 system correction. In M. Kavanaugh (Ed. production) *Proc. 18th Int. Conf. Software Engineering (ICSE'96)*. Los Alamitos, CA: IEEE Computer Soc., 475-484.

Hartman, J. 1991. Understanding natural programs using proper decomposition. In *Proc. 13th Int. Conf. Software Engineering (ICSE'91)*. IEEE Computer Soc./ACM Press, 62-73.

Hasti, R. & Horwitz, S. 1998. Using static single assignment form to improve flow-sensitive pointer analysis. *ACM SIGPLAN Notices* 33 (5), 97-105. *Proc. ACM SIGPLAN'98 Conf. Programming Language Design and Implementation (PLDI'98)*.

Haworth, D., Sharpe, S. & Hale, D. 1992. A framework for software maintenance: a foundation for scientific inquiry. *J. Software Maintenance: Research & Practice* 4 (2), 105-117.

Heisler, K., Kasho, Y. & Tsai, W. 1993. A reverse engineering model for C programs. *Information Sciences* 68, 155-189.

Henry, S. Humphrey, M. 1993. Object-oriented vs procedural programming languages: effectiveness in program maintenance. *JOOP* 6 (3), 41-49.

Hoffner, T., Kamkar, M. & Fritzson, P. 1995. Evaluation of program slicing tools. In *Proc. 2nd Workshop on Automated and Algorithmic Debugging (AADEBUG'95).* IRISA-CNRS, 51-69.

Hopkins, J. & Jernow, J. 1990. Documenting the software development process. In *Proc. SIGDOC'90.* ACM Press, 125-133.

Horwitz, S. 1990a. Identifying the semantic and textual differences between two versions of a program. In *Proc. SIGPLAN'90 Conf. Programming Language Design and Implementation (PLDI'90),* 234-246.

Horwitz, S. 1990b. Adding relational query facilities to software development environments. *Theoretical Computer Science* 73 (2), 213-230.

Horwitz, S., Prins, J. & Reps, T. 1988. On the adequacy of program dependence graphs for representing programs. In *Proc. 15th ACM Symp. Principles of Programming Languages (POPL'88).* New York: ACM Press, 146-157.

Horwitz, S., Prins, J. & Reps, T. 1989. Integrating non-interfering versions of programs. *ACM TOPLAS* 11 (3), 345-387.

Horwitz, S. & Reps, T. 1991. Efficient comparison of program slices. *Acta Informatica* 28 (8), 713-732.

Horwitz, S. & Reps, T. 1992. The use of program dependence graphs in software engineering. In *Proc. 14th Int. Conf. Software Engineering (ICSE'92).* New York: ACM Press, 392-411.

Horwitz, S., Reps, T. & Binkley, D. 1990. Interprocedural slicing using dependence graphs. *ACM TOPLAS* 12 (1), 26-60.

Hutchens, D. & Basili, V. 1985. System structure analysis: clustering with data bindings. *IEEE TOSE* 11 (8), 749-757.

Iio, K., Furuyama, T. & Arai, Y. 1997. Experimental analysis of the cognitive processes of program maintainers during software maintenance. In *Proc. Int. Conf. Software Maintenance (ICSM'97).* IEEE Computer Soc.

IJSEKE. 1994. *Int. J. Software Engineering and Knowledge Engineering* 4 (3). Special issue on reverse engineering.

Imagix. 2000. Imagix 4D. Product information available (10-Mar-00) in www-form at <URL: http://www.imagix.com>. Company: Imagix. Description: a reverse engineering tool (for C, C++).

IntegriSoft. 2000. HindSight. Product information available (10-Mar-00) in www-form at <URL: http://www.integrisoft.com/hindsight.htm>. Company: IntegriSoft. Description: a reverse engineering tool (for C, C++, Fortran).

IS. 1995. *IEEE Software* 12 (1). Special issue on legacy systems.

IS. 1998. *IEEE Software* 15 (4). Theme issue on managing legacy systems.

Iselin, B. 1988. Conditional statements, looping constructs, and program comprehension: an experimental study. *IJMMS* 28 (1), 45-66.

IST. 1998. *Information and Software Technology* 40 (11/12). Special issue on program slicing.

Jackson, D. & Ladd, D. 1994. Semantic diff: a tool for summarizing the effects of modifications. In *Proc. Int. Conf. Software Maintenance (ICSM'94),* 243-252.

Jackson, D. & Rollins, E. 1996. Abstraction mechanisms for pictorial slicing. In Cimitile, A. & Muller, H. (Ed.) *Proc. 4th Int. Workshop on Program Comprehension (IWPC'96)*. IEEE Computer Soc.

JASIS. 1994. *JASIS* 45 (3). Special issue on relevance.

JASIS. 1998. *JASIS* 49 (5). Special issue on knowledge discovery and data mining.

Jeffries, R., Miller, J., Wharton, C. & Uyeda, K. 1991. User interface evaluation in the real world: a comparison of four techniques. In *Proc. Conf. Human Factors in Computing Systems*. New York: ACM Press, 119-124.

Jiang, J., Zhou, X. & Robson, D. 1991. Program slicing for C - the problems in implementation. In *Proc. Int. Conf. Software Maintenance (ICSM'91)*. IEEE Computer Soc., 182-190.

Johmann, K., Liu, S.-S. & Yau, S. 1995. Context-dependent flow-sensitive interprocedural dataflow analysis. *J. Software Maintenance: Research and Practice* 7 (3), 177-202.

Jones, C. 1989. Software enhancement modelling. *J. Software Maintenance - Research and Practice* 1, 91-100.

Jones, C. 1997. Slow response to Year 2000 problem. *IEEE Software* 14 (3), 114-115 (an interview).

JSS. 1999. *The J. Systems and Software* 44 (3). Special issue on program comprehension.

Kafura, D. & Reddy, G. 1987. The use of software complexity metrics in software maintenance. *IEEE TOSE* 13 (3), 335-343.

Kamkar, M. 1993. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. Dept. of Computer and Information Science, Linköping Univ., Sweden. Linköping Studies in Science and Technology Dissertations 297 (Diss.).

Kamkar, M. 1995. An overview and comparative classification of program slicing techniques. *The J. Systems and Software* 31 (3), 197-214.

Kamkar, M. 1998. Application of program slicing in algorithmic debugging. *Information and Software Technology* 40 (11/12), 637-646.

Kaplan, S. & Maarek, Y. 1990. Incremental maintenance of semantic links in dynamically changing hypertext systems. *Interacting with Computers* 2 (3), 337-366.

Katajainen, J. & Mäkinen, E. 1990. Tree compression and optimization with applications. *Int. J. Foundations of Computer Science* 1 (4), 425-447.

Katajainen, J., Penttonen, M. & Teuhola, J. 1986. Syntax-directed compression of program files. *Software Practice and Experience* 16 (3), 269-276.

Kernighan, B. & Ritchie, D. 1988. *The C Programming Language* (2nd ed.). Englewood-Cliffs: Prentice Hall.

Kerola, P. & Oinas-Kukkonen, H. 1992. Hypertext system as an intermediary agent in CASE environments. In K. Kendall, K. Lyytinen & J. DeGross (Ed.) *The Impact of Computer Supported Technologies on Information Systems Development*. NY: North-Holland, 289-313.

Khoshgoftaar, T., Szabo, R. & Voas, J. 1995. Detecting program modules with low testability. In *Proc. Int. Conf. Software Maintenance (ICSM'95)*, 242-250.

Kinloch, D. & Munro, M. 1993. A combined representation for the maintenance of C programs. In *Proc. 2nd Int. Workshop on Program Comprehension (IWPC'93)*. IEEE Computer Soc., 119-127.

Kinloch, D. & Munro, M. 1994. Understanding C programs using the combined C graph representation. In *Proc. Int. Conf. Software Maintenance (ICSM'94)*, 172-180.

Knapen, G., Lague, B., Dagenais, M. & Merlo, E. 1999. Parsing C++ despite missing declarations. In B. Werner (Ed. production) *Proc. 7th Int. Workshop on Program Comprehension (IWPC'99)*. IEEE Computer Soc., 114-125.

Koenemann, J. & Robertson, S. 1991. Expert problem solving strategies for program comprehension. In *Proc. Conf. Human Factors in Computing Systems*. ACM Press, 125-130.

Kontogiannis. K., Galler, M., Demori, R., Bernstein, M. & Merlo, E. 1995. Pattern matching for design concept localization. In *Proc. 2nd Working Conf. Reverse Engineering (WCRE'95)*. IEEE Computer Soc., 96-103.

Korel, B. & Rilling, J. 1998. Dynamic program slicing methods. *Information and Software Technology* 40 (11/12), 647-660.

Koskinen, J. 1993. *Chess'93*. Included in (Koskinen *et al.*, 1997). Description: a non-commercial example ANSI-C input program of the HyperSoft system, performing chess analysis. 2,700 LOC.

Koskinen, J. 1995. HyperSoft: a hypertext approach to software maintenance support. In S. Assar & V. Plihon (Ed.) *Proc. 2nd Doctoral Consortium on Advanced Information Systems Engineering - 7th Int. Conf. Information Systems Engineering: Current Practice and Future Prospects (CAiSE'95)*, 30-31.

Koskinen, J. 1996a. *HyperSoft: Static Program Analyzer, Program Data Base and Access Structure Generator Components*. University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Working paper WP-35.

Koskinen, J. 1996b. *HyperSoft: Automated Hypertext Support for Software Maintenance*. University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Technical Reports TR-13. Licentiate thesis in computer science.

Koskinen, J. 1996c. Creating transient hypertextual access structures for C programs. In M. Kavanaugh (Ed. production) *Proc. 7th Israeli Conf. Computer Systems and Software Engineering (ICCSSE'96)*. IEEE Computer Soc., 56-65.

Koskinen, J. 1997. *HyperSoft: Back-end Components*. University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Technical Reports TR-17.

Koskinen, J. 1999a. Empirical evaluation of hypertextual information access from program text. In B. Werner (Ed. production) *Proc. 7th Int. Workshop on Program Comprehension (IWPC'99)*. IEEE Computer Soc., 162-169.

Koskinen, J. 1999b. *Empirical Evaluations of Hypertextual Information Access from Program Text*. University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Working paper WP-36.

Koskinen, J. 1999c. Evaluations of hypertext access from C programs. Conditionally accepted to be published in *J. Software Maintenance: Research and Practice.*

Koskinen, J. 1999d. A bibliography of reverse engineering and hypertext techniques. To appear at personal www home page under <URL: http://www.cs.jyu.fi>.

Koskinen, J., Nieminen, M. & Suominen, T. 1997. *HyperSoft system (v. 1.0).* Dept. of Computer Science and Information Systems, Univ. of Jyväskylä. Distribution disk. Also to appear at personal www home page under <URL: http://www.cs.jyu. fi>. Description: an experimental reverse engineering tool for software maintenance support (for ANSI-C, ESQL). 35,000 LOC.

Koskinen, J., Paakki, J. & Salminen, A. 1994a. Program text as hypertext - using program dependences for transient linking. In *Proc. 6th Int. Conf. Software Engineering and Knowledge Engineering (SEKE'94).* Skokie, IL: Knowledge Systems Institute, 209-216.

Koskinen, J., Salminen, A. & Paakki, J. 1994b. HyperSoft: viewing program text as hypertext to support software maintenance and comprehension. In: *Doctoral Program Seminar on IS Maintenance.* Helsinki: Swedish School of Economics and Business Adminstration.

Koskinen, J., Salminen, A. & Paakki, J. 1999. Hypertext support for information needs of software maintainers. Univ. of Jyväskylä, Jyväskylä, Finland. *Computer Science and Information Systems Reports, Working paper* WP-37. Submitted for publication to *IEEE Transactions on Software Engineering.*

Kozaczynski, W., Ning, J. & Engberts, A. 1992. Program concept recognition and transformation. *IEEE TOSE* 18 (12), 1065-1075.

Krasner, G. & Pope, S. 1988. A cookbook for using the model-view-controller interface paradigm in Smalltalk-80. *JOOP* 1 (3), 26-49.

Kuhlthau, C. 1991. Inside the search process: information seeking from the user's perspective. *JASIS* 42 (5), 361-371.

Kuikka, E. 1996. *Processing of Structured Documents Using a Syntax-Directed Approach.* Univ. Kuopio, Kuopio, Finland. Kuopio University Publications C-53 (Diss.).

Kuikka, E. & Salminen, A. 1997. Two-dimensional filters for structured text. *Information Processing & Management* 33 (1), 37-54.

Lakhotia, A. 1993a. Understanding someone else's code: analysis of experiences. *The J. Systems and Software* 23, 269-275.

Lakhotia, A. 1993b. Constructing call multigraphs using dependence graphs. In *Conf. Record of the 20th ACM Symp. Principles of Programming Languages (POPL'93).* ACM Press, 273-284.

Landi, W., Ryder, B. & Zhang, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. *ACM SIGPLAN Notices* 28 (6), 56-67. *Proc. SIGPLAN'93 Conf. Programming Language Design and Implementation (PLDI'93).*

Lee, B. & Hurson, A. 1993. Issues in dataflow computing. *AIC* 37, 285-333.

Leggett, J. & Schnase, J. 1994. Viewing Dexter with open eyes. *CACM* 37 (2), 76-86.

Lelewer, D. & Hirschberg, D. 1987. Data compression. *ACM Computing Surveys* 19 (3), 261-296.

Letovsky, S. 1986. Cognitive process in program comprehension. In E. Soloway & S. Iyengar (Ed.) *Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86)*. Norwood, NJ: Ablex, 80-98.

Letovsky, S. & Soloway, E. 1986. Delocalized plans and program comprehension. *IEEE Software* 3 (3), 41-49.

Leung, Y. & Apperley, M. 1994. A review and taxonomy of distortion-oriented presentation techniques. *ACM TOCHI* 1 (2), 126-160.

Liang, D. & Harrold, M. 1999. Efficient points-to-analysis for whole program analysis. *ACM SIGSOFT Software Engineering Notes* 24 (6), 199-215. *Proc. 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE'99).*

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., & Tulula, P. 1994. Visualizing program dependencies: an experimental study. *Software - Practice and Experience* 24 (4), 387-403.

Linos, P. & Courtois, V. 1996. A toolset for maintaining hybrid C++ programs. *J. Software Maintenance: Research and Practice* 8 (6), 389-420.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., & Tulula, P. 1993a. Facilitating the comprehension of C programs: an experimental study. In *Proc. 2nd Int. Workshop on Program Comprehension (IWPC'93)*. IEEE Computer Soc., 55-63.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., & Tulula, P. 1993b. CARE: an environment for understanding and re-engineering C programs. In *Proc. Int. Conf. Software Maintenance (ICSM'93)*. IEEE Computer Soc., 130-139.

Linos, P., Ososanya, E. & Karunakaran, V. 1999. Improving the visibility of graphical program displays: an experimental study. In B. Werner (Ed. production) *Proc. 7th Int. Workshop on Program Comprehension (IWPC'99)*. IEEE Computer Soc., 12-19.

Livadas, P. & Johnson, T. 1994. A new approach to finding objects in programs. *J. Software Maintenance: Research and Practice* 6 (5), 249-260.

Livadas, P. & Roy, P. 1992. Program dependency analysis. In *Proc. Int. Conf. Software Maintenance (ICSM'92)*. IEEE Computer Soc., 356-365.

Luqi. 1989. Software evolution through rapid prototyping. *Computer* 22 (5), 13-25.

MacLeod, I. 1991a. A query language for retrieving information from hierarchic text structures. *The Computer J.* 34 (3), 254-264.

MacLeod, I. 1991b. Text retrieval and the relational model. *JASIS* 42 (3), 155-165.

Mancl, D. & Havanas, W. 1990. A study of the impact of C++ on software maintenance. *Proc. IEEE Int. Conf. Software Maintenance (ICSM'90)*, 63-69.

Marcoccia, L. 1998. Building infrastructure for fixing the year 2000 bug: a case study. *J. Software Maintenance: Research and Practice* 10 (5), 333-352.

Marlowe, T. & Ryder, B. 1991. Hybrid incremental alias algorithms. In *Proc. 24th Hawaii Int. Conf. System Sciences (HICSS'91) (Vol. II)*, 428-437.

Martin, R. 1997. Dealing with dates: solutions for the Year 2000. *Computer* 30 (3), 44-51.

Matwin, S. & Ahmad, A. 1994. Reuse of modular software with automated comment analysis. In *Proc. Int. Conf. Software Maintenance (ICSM'94)*, 222-233.

von Mayrhauser, A. 1994. Maintenance and evolution of software products. *AIC* 39, 1-49.

von Mayrhauser, A. & Lang, S. 1999. On the role of static analysis during software maintenance. In B. Werner (Ed. production) *Proc. 7th Int. Workshop on Program Comprehension (IWPC'99)*. IEEE Computer Soc., 170-177.

von Mayrhauser, A. & Vans, A.M. 1995a. Program comprehension during software maintenance and evolution. *Computer* 28 (2), 44-55.

von Mayrhauser, A. & Vans, A.M. 1995b. Industrial experience with an integrated code comprehension model. *Software Engineering J.* 10 (5), 171-182.

von Mayrhauser, A. & Vans, A.M. 1995c. Program understanding models and experiments. *AIC* 40, 1-38.

von Mayrhauser, A. & Vans, A.M. 1996. Identification of dynamic comprehension processes during large scale maintenance. *IEEE TOSE* 22 (6), 424-437.

von Mayrhauser, A. & Vans, A.M. 1997a. Hypothesis-driven understanding processes during corrective maintenance of large scale software. In *Proc. Int. Conf. Software Maintenance (ICSM'97)*. IEEE Computer Soc., 12-20.

von Mayrhauser, A. & Vans, A.M. 1997b. Program understanding needs during corrective maintenance of large systems. In *Proc. 21st Ann. Computer Software & Applications Conf. (COMPSAC'97)*. IEEE Computer Soc., 630-637.

von Mayrhauser, A. & Vans, A.M. 1998. Program understanding during software adaptation tasks. In *Proc. Int. Conf. Software Maintenance (ICSM'98)*. IEEE Computer Soc., 316-325.

von Mayrhauser, A., Vans, A.M. & Howe, A. 1997. Program understanding behaviours during enhancement of large-scale software. *J. Software Maintenance: Research and Practice* 9 (5), 299-327.

McCabe, T. 1976. A complexity measure. *IEEE TOSE* 2 (4), 308-320.

McDonald, S. & Stevenson, R. 1998. Navigation in hyperspace: an evaluation of the effects of navigational tools and subject matter expertise on browsing and information retrieval in hypertext. *Interacting with Computers* 10 (2), 129-142.

Mendelzon, A. & Sametinger, J. 1995. Reverse engineering by visualizing and querying. *Software - Concepts and Tools* 16 (4), 170-182.

Misra, S. 1990. Evaluating CASE system characteristics: evaluative framework. *Information and Software Technology* 32 (6), 415-422.

Mizzaro, S. 1997. Relevance: the whole history. *JASIS* 48 (9), 810-832.

Monk, A., Walsh, P. & Dix, A. 1988. A comparison of hypertext, scrolling and folding as mechanisms for program browsing. In D. Jones & R. Winder (Ed.) *People and Computers IV*. Cambridge: Cambridge Univ., 421-435.

Moriconi, M. & Hare, D. 1986. The PegaSys system: pictures as formal documentation of large programs. *ACM TOPLAS* 8 (4), 524-546.

Moriconi, M. & Winkler, T. 1990. Approximate reasoning about the semantic effects of program changes. *IEEE TOSE* 16 (9), 980-992.

Moser, L. 1990. Data dependence graphs for Ada programs. *IEEE TOSE* 16 (5), 498-502.

Moulin, B. & Rousseau, D. 1992. Automated knowledge acquisition from regulatory texts. *IEEE Expert* 7 (5), 27-35.

Muller, H. & Klashinsky, K. 1988. Rigi: a system for programming-in-the-large. In *Proc. 10th Int. Conf. Software Engineering (ICSE'88)*, 80-86.

Muller, H., Orgun, M., Tilley, S., & Uhl, J. 1993. A reverse engineering approach to subsystem structure identification. *J. Software Maintenance: Research and Practice* 5 (4), 181-204.

Muller, H., Tilley, S., Orgun, M., Corrie, B. & Madhavji, N. 1992. A reverse engineering environment based on spatial and visual software interconnection models. In *Proc. Fifth ACM SIGSOFT Symp. Software Development Environments*. ACM Press, 88-98.

Murphy, G. & Notkin, D. 1996. Lightweight lexical source model extraction. *ACM TOSEM* 5 (3), 262-292.

Murphy, G., Notkin, D., Griswold, W. & Lan, E. 1998. An empirical study of static call graph extractors. *ACM TOSEM* 7 (2), 158-191.

Musciano, C. & Kennedy, B. 1998. *HTML: The Definitive Guide* (3rd ed.). O'Reilly & Assoc.

Newcomb, P. & Scott, M. 1997. Requirements for advanced Year 2000 maintenance tools. *Computer* 30 (3), 52-57.

Nielsen, J. 1989. The matters that really matter for hypertext usability. In F. Halasz & N. Meyrowitz (Ed.) *Proc. 2nd ACM Conf. Hypertext, Hypertext'89*. New York: ACM Press, 239-248.

Nielsen, J. 1990. The art of navigating through hypertext. *CACM* 33 (3), 296-310.

Nieminen, M. 1996. *HyperSoft järjestelmän käyttöliittymä ja sen kehittäminen* (*HyperSoft system: the user interface and its development*) (in Finnish). Univ. of Jyväskylä. Master's thesis in computer science.

Nieminen, M. & Koskinen, J. 1997. *HyperSoft: käyttäjän käsikirja* (*HyperSoft: User's Manual*) (in Finnish). Jyväskylä, Finland: HyperSoft project, Dept. of Computer Science and Information Systems, Univ. of Jyväskylä.

Ning, J., Engberts, A. & Kozaczynski, W. 1993. Recovering reusable components from legacy systems by program segmentation. In *Proc. 1st Working Conf. Reverse Engineering (WCRE'93)*, 64-72.

Ning, J., Engberts, A. & Kozaczynski, W. 1994. Automated support for legacy code understanding. *CACM* 37 (5), 50-57.

Noonan, R. 1985. An algorithm for generating abstract syntax trees. *Computer Languages* 10 (3/4), 225-236.

Nunamaker, J.F. Jr., Chen, M. & Purdin, T. 1991. Systems development in information systems research. *J. Management Information Systems* 7 (3), 89-106.

Nørmark, K. & Østerbye, K. 1994. Representing programs as hypertext. In B. Magnusson, G. Hedin & S. Minör (Ed.) *Proc. Nordic Workshop on Programming Environment Research (NWPER'94)*. LU-CS-TR: 94-127. Lund, Sweden: Lund Univ., 11-24.

Nørmark, K. & Østerbye, K. 1995. Rich hypertext: a foundation for improved interaction techniques. *IJHCS* 43 (3), 301-321.

Oinas-Kukkonen, H. 1997a. *Improving the Functionality of Software Design Environments by Using Hypertext*. Univ. of Oulu, Finland. Acta Univ. Ouluensis, A 296 (Diss.).

Oinas-Kukkonen, H. 1997b. Towards greater flexibility in software design systems through hypermedia functionality. *Information and Software Technology* 39 (6), 391-397.

Ott, L. & Bieman, J. 1998. Program slices as an abstraction for cohesion measurement. *Information and Software Technology* 40 (11/12), 691-700.

Ott, L. & Thuss, J. 1989. The relationship between slices and module cohesion. In *Proc. 11th Int. Conf. Software Engineering (ICSE'89)*. IEEE Computer Soc./ACM Press, 198-204.

Ottenstein, K. & Ottenstein, L. 1984. The program dependence graph in a software development environment. *ACM SIGPLAN Notices* 19 (5), 177-184. *ACM SIGSOFT Software Engineering Notes* 9 (3). *Proc. ACM SIGPLAN/SIGSOFT Symp. Practical Programming Developm. Environments.*

Paakki, J., Salminen, A. & Koskinen, J. 1996. Automated hypertext support for software maintenance. *The Computer J.* 39 (7), 577-597.

Paakki, J., Koskinen, J. & Salminen, A. 1997. From relational program dependencies to hypertextual access structures. *Nordic Journal of Computing* 4 (1), 3-36.

Palthepu, S., Greer, J. & McCalla, G. 1997. Cliche recognition in legacy software: a scalable, knowledge-based approach. In P. Storms (Ed. production) *Proc. 4th Working Conf. Reverse Engineering (WCRE'97)*. Los Alamitos, CA: IEEE Computer Soc., 94-103.

Pande, H., Landi, W. & Ryder, B. 1994. Interprocedural def-use associations for C systems with single level pointers. *IEEE TOSE* 20 (5), 385-403.

Parr, T. & Quong, R. 1995. ANTLR: a predicated-*LL(k)* parser generator. *Software - Practice and Experience* 25 (7), 789-810.

Parr, T. & Quong, R. 1996. *LL* and *LR* translators need *k*>1 lookahead. *ACM SIGPLAN Notices* 31 (2), 27-34.

Parsifal. 1993. *AnaGram™ - User's Guide*. Wayland, MA: Parsifal Software.

Paul, S. & Prakash, A. 1994a. A framework for source code search using program patterns. *IEEE TOSE* 20 (6), 463-475.

Paul, S. & Prakash, A. 1994b. Supporting queries on source code: a formal framework. *Int. J. Software Engineering and Knowledge Engineering* 4 (3), 325-348.

Paul, S. & Prakash, A. 1996. A query algebra for program databases. *IEEE TOSE* 22 (3), 202-217.

Peltola, H. & Tarhio, J. 1991. On syntactical data compression. In K. Koskimies *et al.* (Ed.) *Proc. 2nd Symp. Programming Languages and Software Tools*. Report A-1991-5. Dept. of Computer Science, Univ. of Tampere, Finland, 205-214.

Pennington, N. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19 (3), 295-341.

Perry, D. 1987. Software interconnection models. In *Proc. 9th Int. Conf. Software Engineering (ICSE'87)*. ACM Press, 61-69.

Pezze, M., Taylor, R. & Young, M. 1995. Graph models for reachability analysis of concurrent programs. *ACM TOSEM* 4 (2), 171-213.

Pirklbauer, K. 1992. A study of pattern-matching algorithms. *Structured Programming* 13 (2), 89-98.

Pfleeger, S. 1997. Experimentation in software engineering. *AIC* 44, 127-167.

Platoff, M. & Wagner, M. 1991. An integrated program representation and toolkit for the maintenance of C programs. In *Proc. Int. Conf. Software Maintenance (ICSM'91).*

Podgurski, A. & Clarke, L. 1990. A formal model of program dependences and its implications for software testing, debugging and maintenance. *IEEE TOSE* 16 (9), 965-979.

Pollock, L & Soffa, M. 1989. An incremental version of iterative data flow analysis. *IEEE TOSE* 11 (12), 1537-1549.

Pressman, R. 1997. *Software Engineering - A Practitioner's Approach* (4th ed.). McGraw-Hill.

Prieto-Diaz, R. 1991. Implementing faceted classification for software reuse. *CACM* 34 (5), 89-97.

Pugh, W. 1992. A practical algorithm for exact array dependence analysis. *CACM* 35 (8), 102-114.

Qiu, L. 1993. Analytical searching vs browsing in hypertext information retrieval systems. *Canadian J. Library & Information Science* 18 (4), 1-13.

Queille, J.-P., Voidrot, J.-F., Wilde, N. & Munro, M. 1994. The impact analysis task in software maintenance: a model and a case study. In *Proc. Int. Conf. Software Maintenance (ICSM'94).*

Rada, R. 1992. Converting a textbook into hypertext. *ACM TOIS* 10 (3), 294-315.

Rada, R. & Murphy, C. 1992. Searching versus browsing in hypertext. *Hypermedia* 4 (1), 1-30.

Rada, R., Wang, W., Mili, H, Heger, J. & Scherr, W. 1992. Software reuse: from text to hypertext. *Software Engineering J.* 7, 311-321.

Ragland, B. 1997. *The Year 2000 Problem Solver: A Five Step Disaster Prevention Plan*. McGraw-Hill.

Rajlich, V. & Varadarajan, S. 1999. Using the web for software annotations. *Int. J. Software Engineering and Knowledge Engineering* 9 (1), 55-72.

Ramalingam, G. & Reps, T. 1992. A theory of program modifications. In *Proc. Colloquium on Combining Paradigms for Software Development*. Springer-Verlag, 137-152.

Raymond, D. & Tompa, F.Wm. 1988. Hypertext and the Oxford English Dictionary. *CACM* 31 (7), 871-879.

Rayside, D., Kerr, S. & Kontogiannis, K. 1998. Change and adaptive maintenance detection in Java software systems. In *Proc. 5th Working Conf. Reverse Engineering (WCRE'98)*. IEEE Computer Soc., 10-19.

Red Hat. 2000. Cygnus Source Navigator v. 4.5. Product information available (10-Mar-00) in www-form at <URL: http://www.redhat.com/products/cygnus.html>. Company: Red Hat. Description: a reverse engineering tool.

Regelson, E. & Anderson, A. 1994. Debugging practices for complex legacy software systems. In *Proc. Int. Conf. Software Maintenance (ICSM'94)*, 137-145.

Reps, T. 1998. Program analysis via graph reachability. *Information and Software Technology* 40 (11/12), 701- 726.

Reps, T., Horwitz, S., Sagiv, M. & Rosay, G. 1994. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes* 19 (5), 11-20. *ACM SIGSOFT'94, Proc. 2nd ACM SIGSOFT Symp. Foundations of Software Engineering (FSE'2).* ACM Press.

Reps, T. & Rosay, G. 1995. Precise interprocedural chopping. *ACM SIGSOFT Software Engineering Notes* 20 (4), 41-52. G. Kaiser (Ed.) *SIGSOFT'95: Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engineering (FSE'3).*

Rich, C. & Waters, R. 1988. The Programmer's Apprentice: a research overview. *Computer* 21 (11), 10-25.

Rich, C. & Wills, L. 1990. Recognizing a program's design: a graph-parsing approach. *IEEE Software* 7 (1), 82-89.

Riecken, R., Koenemann-Belliveau, J. & Robertson, S. 1991. What do expert programmers communicate by means of descriptive commenting. In J. Koenemann-Belliveau, T. Moher & S. Robertson (Ed.) *Empirical Studies of Programmers: 4th Workshop (ESP'91).* Norwood, NJ: Ablex, 177-195.

Risku, V.-M. 1995. *Siivuttaminen ohjelmistojen ylläpidossa (Slicing in software maintenance)* (in Finnish). Univ. of Jyväskylä. Master's thesis in computer science.

Ritchie, D. 1993. The development of the C language. *ACM SIGPLAN Notices* 28 (3), 201-208. *Proc. 2nd ACM SIGPLAN Conf. History of Programming Languages (HOPL-II).*

Rivlin, E., Botafogo, R. & Shneiderman, B. 1994. Navigating in hyperspace: designing a structure-based toolbox. *CACM* 37 (2), 87-96.

Robson, D., Bennett, K., Cornelius, B. & Munro, M. 1991. Approaches to program comprehension. *The J. Systems and Software* 14 (2), 79-84.

Robertson, S. & Yu, C.-C. 1990. Common cognitive representations of program code across tasks and languages. *IJMMS* 33 (3), 343-360.

Rock-Evans, R. & Hales, K. 1992. *Reverse Engineering: Markets, Methods and Tools.* London: Ovum.

Rossiter, B., Sillitoe, T. & Heather, M. 1990. Database support for very large hypertexts. *Electronic Publishing* 3 (3), 141-154.

Rugaber, S., Ornburn, S. & LeBlanc, R. Jr. 1990. Recognizing design decisions in programs. *IEEE Software* 7 (1), 46-54.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. 1991. *Object-Oriented Modeling and Design.* Englewood-Cliffs, NJ: Prentice-Hall.

Ryder, B. & Paul, M. 1988. Incremental data flow analysis algorithms. *ACM TOPLAS* 10 (1), 1-50.

Salminen, A., Koskinen, J. & Paakki, J. 1994a. HyperSoft: an environment for hypertextual software maintenance. In B. Magnusson, G. Hedin & S. Minör (Ed.) *Proc. Nordic Workshop on Programming Environment Research (NWPER'94).* LU-CS-TR: 94-127. Lund, Sweden: Lund Univ., 25-37.

Salminen, A., Paakki, J. & Koskinen, J. 1994b. Incorporating hypertext functionality into software maintenance environments. In *Workshop on Corporating*

Hypertext Functionality into Software Systems, ACM European Conference on Hypermedia Technologies (ECHT'94). ACM Press.

Salminen, A., Tague-Sutcliffe, J. & McClellan, C. 1995. From text to hypertext by indexing. *ACM TOIS* 13 (1), 69-99.

Salminen, A. & Watters, C. 1992. A two-level structure for textual databases to support hypertext access. *JASIS* 43 (6), 432-447.

Salton, G., Allan, J. & Buckley, C. 1994. Automatic structuring and retrieval of large text files. *CACM* 37 (2), 97-108.

Salton, G., Buckley, C. & Smith, M. 1990. On the application of syntactic methodologies in automatic text analysis. *Information Processing & Management* 26 (1), 73-92.

Samadzadeh, M. & Wichaipanitch, W. 1993. An interactive debugging tool for C based on dynamic slicing and dicing. In *Proc. 21st Annual Computer Science Conf.* ACM Press, 30-37.

Schwarz, C. 1990. Automatic syntactic analysis of free text. *JASIS* 41 (6), 409-415.

SET. 2000. Discover. Product information available (10-Mar-00) in www-form at <URL: http://www.setech.com/products>. Company: SET Inc. Description: a reverse engineering tool (for ANSI C/C++ *et al.*).

Sharon, D. 1997. Year 2000 tool classification scheme. *IEEE Software* 14 (4), 107-111.

Shepherd, M., Watters, C. & Cai, Y. 1990. Transient hypergraphs for citation networks. *Information Processing & Management* 26 (3), 395-412.

Shneiderman, B. 1986. Empirical studies of programmers: the territory, paths, and destinations. In E. Soloway & S. Iyengar (Ed.) *Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86)*. Norwood, NJ: Ablex, 1-12.

Shneiderman, B. 1989. Reflections on authoring, editing, and managing hypertext. E. Barret (Ed.) *The Society of Text*. Cambridge, MA: MIT Press, 115-131.

Shneiderman, B. 1992. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (2nd ed.). Reading, MA: Addison-Wesley.

Sillanpää, R. 1997. *Hypertekstin visualisointi* (*Visualization of hypertext*) (in Finnish). Univ. of Helsinki. Master's thesis in computer science.

Sloane, A. & Holdsworth, J. 1996. Beyond traditional program slicing. *ACM SIGSOFT Software Engineering Notes* 21 (3), 180-186. S. Zeil (Ed.) *Proc. 1996 Int. Symp. Software Testing and Analysis (ISSTA'96)*.

Soloway, E., Adelson, B. & Ehrlich, K. 1988. Knowledge and processes in the comprehension of computer programs. In M. Chi, R. Glaser & M. Farr (Ed.) *The Nature of Expertise*. Hillsdale, NJ: Lawrence Erlbaum Ass., 129-152

Soloway, E., Bonar, J. & Ehrlich, K. 1983. Cognitive strategies and looping constructs: an empirical study. *CACM* 26 (11), 853-860.

Soloway, E. & Ehrlich, K. 1984. Empirical studies of programming knowledge. *IEEE TOSE* 10 (5), 595-609.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. 1982. What do novices know about programming? In B. Shneiderman & A. Badre (Ed.) *Directions in Human-Computer Interaction*. Norwood, NJ: Ablex.

Soloway, E., Pinto, J., Letovsky, S., Littman, D. & Lampert, R. 1988. Designing documentation to compensate for delocalized plans. *CACM* 31 (11), 1259-1267.

Sommerville, I. 1996. *Software Engineering* (5th ed.). Addison-Wesley.

Storey, M.-A. & Muller, H. 1995. Manipulating and documenting software structures using SHriMP views. In *Proc. Int. Conf. Software Maintenance (ICSM'95)*. IEEE Computer Soc., 275-284.

Storey, M.-A., Wong, K., Fong, P., Hooper, D., Hopkins, K. & Muller, H. 1996. On designing an experiment to evaluate a reverse engineering tool. In *Proc. 3rd Working Conf. Reverse Engineering (WCRE'96)*. IEEE Computer Soc., 31-40.

Storey, M.-A., Wong, K. & Muller, H. 1997. How do program understanding tools affect how programmers understand programs. In P. Storms (Ed. production) *Proc. 4th Working Conf. Reverse Engineering (WCRE'97)*. Los Alamitos, CA: IEEE Computer Soc., 12-21.

Stotts, P. & Furuta, R. 1989. Petri-net-based hypertext: document structure with browsing semantics. *ACM TOIS* 7 (1), 3-29.

Straker, D. 1992. *C Style Standards and Guidelines*. UK: PHI.

Stroustrup, B. 1986. *The C++ Programming Language*. Reading, MA: Addison-Wesley.

Stroustrup, B. 1993. A history of C++: 1979-1991. In *HOPL-II, The Second ACM SIGPLAN History of Programming Languages Conference*. New York: ACM, 271-297.

Suetens, P., Fua, P. & Hanson, A. 1992. Computational strategies for object recognition. *ACM Computing Surveys* 24 (1), 5-61.

Suominen, T. 1997. *Upotetun SQL:n analysointi HyperSoft järjestelmässä* (*The analysis of embedded SQL in HyperSoft system*) (in Finnish). Univ. of Helsinki. Master's thesis in computer science.

Tague, J., Salminen, A. & McClellan, C. 1991. A complete formal model for information retrieval systems. In A. Bookstein, Y. Chiaramella, G. Salton & V. Raghavan (Ed.) *Proc. ACM SIGIR'91*. ACM Press, 14-20.

TakeFive. 2000. Sniff+. Product information available (10-Mar-00) in www-form at <URL: http://www.takefive.com/products/sniff+.html>. Company: TakeFive software. Description: a reverse engineering tool (C, C++ *et al.*).

Tebbutt, J. 1999. User evaluation of automatically generated semantic hypertext links in a heavily used procedural manual. *Information Processing and Management* 35 (1), 1-18.

The COBOL Center. 1999. Product information available (14-Apr-00) in www-form at <URL: http://www.infogoal.com/cbd/cbdtol.htm>. Description: a list of tools (for COBOL).

Tichy, W. 1998. Should computer scientists experiment more. *Computer* 31 (5), 32-40.

Tilley, S., Paul, S. & Smith, D. 1996. Towards a framework for program understanding. In A. Cimitile & H. Muller (Ed.) *Proc. 4th Int. Workshop on Program Comprehension (IWPC'96)*. IEEE Computer Soc., 19-28.

Tip, F. 1995. A survey of programming slicing techniques. *J. Programming Languages* 13 (3), 121-189.

Tonella, G., Fiutem, R. & Merlo, E. 1997. Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In *Proc. 19th Int. Conf. Software Engineering (ICSE'97)*. New York: ACM Press, 433-443.

Tuovinen, A.-P. 1995. *Analyzing, Understanding and Maintaining Object-Oriented Programs*. Helsinki: HyperSoft project, Dept. of Computer Science, Univ. of Helsinki.

Turver, R. & Munro, M. 1994. An early impact analysis technique for software maintenance. *J. Software Maintenance: Research and Practice* 6 (1), 35-52.

Utting, K. & Yankelovich, N. 1989. Context and orientation in hypermedia networks. *ACM TOIS* 7 (1), 58-84.

Venkatesh, G. 1995. Experimental results from dynamic slicing of C programs. *ACM TOPLAS* 17 (2), 197-216.

Verilog. 2000. Logiscope. Product information available (10-Mar-00) in www-form at <URL: http://www.csverilog.com/products/logiscop.htm>. Company: Verilog. Description: a reverse engineering tool (for 80+ languages).

Vessey, I. 1987. On matching programmers' chunks with program structures: an empirical investigation. *IJMMS* 27 (1), 65-89.

Vessey, I. 1989. Towards a theory of computer program bugs: an empirical test. *IJMMS* 30 (1), 23-46.

Vilela, P., Maldonado, J. & Jino, M. 1997. Program graph visualization. *Software - Practice and Experience* 27 (11), 1245-1262.

Visaggio, G. 1997. Relationships between comprehension and maintenance activities. In P. Storms (Ed. production) *Proc. 5th Int. Workshop on Program Comprehension (IWPC'97)*. Los Alamitos, CA: IEEE Computer Soc., 4-16.

Waddle, V. 1990. Production trees: a compact representation of parsed programs. *ACM TOPLAS* 12 (1), 61-83.

Wan, J. & Bieber, M. 1996. GHMI: a general hypertext data model supporting integration of hypertext and information systems. In *Proc. 29th Hawaii Int. Conf. Systems Sciences (HICSS'96)*.

Wang, Y., Tsai, W.-T. *et al.* 1996. The role of program slicing in ripple effect analysis. In *Proc. 8th Int. Conf. Software Engineering and Knowledge Engineering (SEKE'96)*, 369-376.

Watters, C. & Shepherd, M. 1990. A transient hypergraph-based model for data access. *ACM TOIS* 8 (2), 77-102.

Weide, B., Ogden, W. & Zweben, S. 1991. Reusable software components. *AIC* 33, 1-65.

Weiser, M. 1982. Programmers use slices when debugging. *CACM* 25 (7), 446-452.

Weiser, M. 1984. Program slicing. *IEEE TOSE* 10 (4), 352-357.

Weiser, M. & Lyle, J. 1986. Experiments on slicing-based debugging aids. In E. Soloway & S. Iyengar (Ed.) *Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86)*. Norwood, NJ: Ablex, 187-197.

Welsh, J. & McKeag, M. 1980. *Structured System Programming*. N.J.: Prentice Hall.

Wichman, B., Canning, A., Clutterbuck, D., Winsborrow, L., Ward, N. & Marsh, D. 1995. Industrial perspective on static analysis. *Software Engineering J.* 10 (2), 69-75.

Wiedenbeck, S. 1986. Beacons in computer program comprehension. *IJMMS* 25 (6), 697-709.

Wiedenbeck, S. 1991. The initial stage of program comprehension. *IJMMS* **35** (4), 517-540.

Wiedenbeck, S. & Fix, V. 1993. Characteristics of the mental representations of novice and expert programmers: an empirical study. *IJMMS* 39 (5), 793-812.

Wilde, N., Blackwell, K. & Justice, R. 1998. Understanding data-sensitive code: one piece of the Year 2000 puzzle. *ACM SIGSOFT Software Engineering Notes* 23 (5), 75-80.

Wilde, N., Chapman, A. & Richardson, R. 1994. The extensible dependency analysis tool set: a knowledge base for understanding industrial software. *Int. J. Software Engineering and Knowledge Engineering* 4 (4), 521-534.

Wills, L. 1990. Automated program recognition: a feasibility demonstration. *Artificial Intelligence* 45 (1/2), 113-171.

Wilson, R. & Lam, M. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM SIGPLAN'95 Conf. Programming Language Design and Implementation (PLDI'95)*, 1-12.

Wilson, T. 1994. Information needs and uses: fifty years of progress?. In V. Vickery (Ed.) *Fifty Years of Information Progress*. Aslib, 15-51.

Wong, K. 1996. *Rigi User's Manual Version 5.4.1*. Product information available (10-Mar-00) in www-form at <URL: http://www.rigi.csc.uvic.ca/rigi/manual/user.html>. Description: a reverse engineering tool (for C).

Wright, P. 1991. Cognitive overheads and prostheses: some issues in evaluating hypertexts. In J. Walker (Ed.) *Proc. ACM Conf. Hypertext'91: 3rd ACM Conf. Hypertext*. New York: ACM Press, 1-12.

Wu, S. & Manber, U. 1992. Fast text searching allowing errors. *CACM* 35 (10), 83-91.

Yang, H., Luker, P. & Chu, W. 1997. Code understanding through program transformation for reusable component identification. In P. Storms (Ed. production) *Proc. 5th Int. Workshop on Program Comprehension (IWPC'97)*. Los Alamitos, CA: IEEE Computer Soc., 148-159.

Yang, W. 1991. Identifying syntactic differences between two programs. *Software - Practice and Experience* 21 (7), 739-755.

Yau, S. & Chang, P. 1988. A metric of modifiability for software maintenance. In *Proc. Int. Conf. Software Maintenance (ICSM'88)*. IEEE Computer Soc., 374-381.

Yau, S. & Tsai, J. 1987. Knowledge representation of software component interconnection information for large-scale software modifications. *IEEE TOSE* 13 (3), 355-361.

Younger, E. & Bennett, K. 1993. Model-based tools to record program understanding. In *Proc. 2nd Int. Workshop on Program Comprehension (IWPC'93)*. IEEE Computer Soc., 119-127.

Yueh, T. & Low, C. 1997. Error detection in C++ through dynamic data flow analysis. *Software - Concepts and Tools* 18 (1), 1-13.

Yur, J.-S., Ryder, B., Landi, W. & Stocks, P. 1997. Incremental analysis of side effects for C software system. In *Proc. 19th Int. Conf. Software Engineering (ICSE'97)*. New York: ACM Press, 422-432.

Zelkowitz, M. & Wallace, D. 1998. Experimental models for validating technology. *Computer* 31 (5), 23-31.

Zhou, M. & Tompa, F. Wm. 1998. The suffix-signature method for searching for phrases in text. *Information Systems* 23 (8), 567-588.

Ziv, H. & Osterweil 1995. Research issues in the intersection of hypertext and software development environments. *LNCS* 896, 268-279. *Workshop on Software Engineering and Human-Computer Interaction*. Springer-Verlag.

Zloof, M. 1977. Query-by-example: a database language. *IBM Systems J.* 16 (4), 324-343.

Zvegintzov, N. 1997. A resource guide to Year 2000 tools. *Computer* 30 (3), 58-63.

Østerbye, K. 1992. Structural and cognitive problems in providing version control for hypertext. In D. Lucarella, J. Nanard, N. Nanard & P. Paolini (Ed.) *Proc. ACM European Conf. Hypertext (ECHT'92)*, 33-42.

Østerbye, K. 1995. Literate Smalltalk programming using hypertext. *IEEE TOSE* 21 (2), 138-145.

Østerbye, K. & Nørmark, K. 1993. *The Vision and the Work in the HyperPro Project*. Dept. of Mathematics and Computer Science, Aalborg Univ., Aalborg, Denmark. Technical report: R-93-2012.

Østerbye, K. & Nørmark, K. 1994. An interaction engine for rich hypertexts. In *Proc. ACM European Conf. Hypermedia Technologies (ECHT'94)*. ACM Press 167-176.

Journal name acronyms used

AIC	*Advances in Computers*
ACM LOPLAS	*ACM Letters on Programming Languages and Systems*
ACM TOCHI	*ACM Transactions on Computer-Human Interaction*
ACM TOIS	*ACM Transactions on Information Systems*
ACM TOOIS	*ACM Transactions on Office Information Systems*
ACM TOPLAS	*ACM Transactions on Programming Languages and Systems*
ACM TOSEM	*ACM Transactions on Software Engineering and Methodology*
CACM	*Communications of the ACM*
IEEE TOSE	*IEEE Transactions on Software Engineering*
IJHCS	*International Journal of Human-Computer Studies*
IJMMS	*International Journal of Man-Machine Studies*
JASIS	*Journal of the American Society for Information Science*
JOOP	*Journal of Object-Oriented Programming*
LNCS	*Lecture Notes in Computer Science*

APPENDIX 1 Algorithmic solutions for software analysis

Since hypertext is formed automatically in our approach, the availability of automated techniques is essential. This appendix gathers and classifies references to the important sources on automated analysis in software engineering. The list aims to be representative, but not exhaustive. The survey is based on the publications listed in Appendix 2. The references have been selected from a wide set of sources. Selection criterias have favored: relevance to the promising HyperSoft further research areas, high status of the publication forum, originality, and recent results.

1) General structural analysis

This category comprises general techniques for program decomposition and extraction of objects and aggregates. The techniques relate to:

- identification of objects (Suetens *et al.*, 1992; Cutillo *et al.*, 1993; Muller *et al.*, 1993; Livadas & Johnson, 1994; Canfora *et al.*, 1996a; Murphy & Notkin, 1996; Cimitile *et al.*, 1999);
- chunking, abstraction, and clustering (Hutchens & Basili, 1985; Burnstein & Roberson, 1997);
- determination of concepts and cliches (Wills, 1990; Rich & Wills, 1990; Kozaczynski *et al.*, 1992; Fiutem *et al.*, 1996; Palthepu *et al.*, 1997); and
- restructuring (Griswold & Notkin; 1993; 1995).

2) Techniques based on matching special criteria

2a) General techniques

This category includes techniques for identifying program parts which satisfy some special criteria in relation to some aspect of the development of the software system. For example, the purpose may be the identification of:

- Y2K incompliant pieces of code (Hart & Pizzarello, 1996; van Deursen *et al.*, 1997; Martin, 1997; Wilde *et al.*, 1998);
- other problematic features, see for example (Eyre-Todd, 1993);
- architectural features (Harris *et al.*, 1995; Fiutem *et al.*, 1996);
- reusable components (Caldiera & Basili, 1991; Creech *et al.*, 1991; Canfora *et al.*, 1993; Ning *et al.*, 1993; Faustle *et al.*, 1996; Yang *et al.*, 1997);
- abstract data types (Canfora *et al.*, 1993; 1996b); and
- differences between versions of a program (Horwitz, 1990a; Yang, 1991).

2b) Metrics

Software metrics are calculated measures, which could be used while retrieving software components matching specific criteria in relation to:

- program complexity and general maintainability (McCabe, 1976; Kafura & Reddy, 1987);
- modifiability (Yau & Chang, 1988);
- testability (Khoshgoftaar *et al.*, 1995);
- cohesion (Bieman & Ott, 1994; Ott & Bieman, 1998); and
- coupling (Briand *et al.*, 1997).

3) Text and documentation analysis

This category entails a wide set of techniques which can be used in conjunction (especially in relation to the analysis of software documents and comments) with more specialized techniques of program analysis. Examples of available techniques include:

- text analysis and free text retrieval (Salton *et al.*, 1990; Schwartz, 1990; Baeza-Yates & Gonnet, 1992; Wu & Manber, 1992; Dunlop & van Rijsbergen, 1993; Salton *et al.*, 1994; Zhou & Tompa, 1998);
- comment analysis (Matwin & Ahmad, 1994);
- data mining and concept extraction (Moulin & Rousseau, 1992; JASIS, 1998; Computer, 1999); and
- pattern matching (Aho, 1990; Pirklbauer, 1992; Kontogiannis *et al.*, 1995; Griswold *et al.*, 1996).

4) Formation of trees and graphs

The structure and operation of programs can be described for the purposes of program visualization and storage of program information by using various (graphical) diagrams (trees and general graphs), including

- abstract syntax trees (Noonan, 1985);
- production trees (Waddle, 1990);
- call graphs (Hall & Kennedy, 1992; Lakhotia, 1993b; Grove *et al.*, 1997);
- data flow graphs (Benedusi *et al.*, 1989);
- control dependency graphs (Cytron *et al.*, 1991); and
- program (dependency) graphs (Cimitile & Carlini, 1991; Horwitz & Reps, 1992; Harrold *et al.*, 1993).

5) Program dependency analysis

5a) General analysis

Most notably, program dependency analysis includes data flow and data dependency analysis, and other variants. More precisely, the research include

- definitions of data models, for instance: Chen, Y.-F. *et al.* (1990) (for C), Chen, Y.-F. *et al.* (1998) (for C++), Rayside *et al.* (1998) (for Java) and Moser (1990) (for Ada);
- general data flow and data dependency analysis (Bergeretti & Carre, 1985; Ryder & Paul, 1988; Pollock & Soffa, 1989; Burke, 1990; Burke & Ryder, 1990; Choi *et al.*, 1991a; 1991b; Lee & Hurson, 1993; Chen, T. & Low, 1997; Ashley & Bybvig, 1998);
- interprocedural dependency analysis (Johmann *et al.*, 1995; Harrold *et al.*, 1998);
- intermodular data flow analysis (Canfora & Cimitile, 1992);
- side-effect-, change- and impact- analysis (Moriconi & Winkler, 1990; Landi *et al.*, 1993; Jackson & Ladd, 1994; Al-Zoubi & Prakash, 1995; Bohner & Arnold, 1996; Wang *et al.*, 1996; Yur *et al.*, 1997);
- determination of definition-use dependencies (Harrold & Soffa, 1990; Pande *et al.*, 1994);
- alias analysis (Marlowe & Ryder, 1991);
- reachability analysis (Pezze *et al.*, 1995; Reps, 1998);

- array dependency analysis (Pugh, 1992); and
- dead code detection (Bodik & Gupta, 1997; Chen, Y.-F. *et al.*, 1998).

5b) C program analysis

The research on the language area of HyperSoft includes

- representation models (Heisler *et al.*, 1993; Kinloch & Munro, 1993);
- program analysis techniques (Pande *et al.*, 1994; Wilson & Lam, 1995; Gannod & Cheng, 1996; Yur *et al.*, 1997); and
- tools, most notably those represented in (Chen, Y.-F. *et al.*, 1990; Jiang *et al.*, 1991; Platoff & Wagner, 1991; Gallagher, 1992; 1997; Linos *et al.*, 1993a; 1993b; Samadzadeh & Wichaipanitch, 1993; Venkatesh, 1995; Jackson & Rollins, 1996).

6) Program slicing

6a) Variants of program slicing

There exists several variants of program slicing. Algorithms and solutions related to specific variants include the following:

- static slicing (Choi & Ferrante, 1994);
- dynamic slicing (Gopal, 1991; Kamkar, 1993; Korel & Rilling, 1998);
- hybrid slicing combining static and dynamic analysis (Gupta *et al.*, 1996);
- parametric program slicing (Field *et al.*, 1995);
- pictorial slicing (Jackson & Rollins, 1996);
- slicing based on dependency graphs (Horwitz *et al.*, 1990);
- dicing (Chen, T. & Cheung, 1993; Samadzadeh & Wichaipanitch, 1993);
- generalized slicing (Sloane & Holdsworth, 1996);
- sliving (Gupta, 1997); and
- chopping (Reps & Rosay, 1995).

6b) Solutions in the problem areas of slicing

The special problem areas of slicing are discussed and solutions proposed in:

- analysis of unstructured programs (Ball & Horwitz, 1992; Agrawal, 1994; Choi & Ferrante, 1994; Harman & Danicic, 1998);
- pointer analysis (Wilson & Lam, 1995; Atkinson & Griswold, 1998; Hasti & Horwitz, 1998; Fiutem *et al.*, 1999; Liang & Harrold, 1999);
- comparison of program slices (Horwitz & Reps, 1991); and
- slicing as used in program integration (Horwitz *et al.*, 1989).

7) Query mechanisms

Query mechanisms have been studied extensively. These mechanisms can be used when specifying information requests to be passed to the support environment. The research includes

- general query languages used in programming, *e.g.* QBE (Zloof, 1977), SQL (Date, 1987);

- mechanisms for specifying information requests for structured documents (Mac Leod, 1991a; 1991b; Burkowski 1992; Clarke *et al.*, 1995; Kuikka & Salminen, 1997);
- query mechanisms for reverse engineering (Consens *et al.*, 1992; Mendelzon & Sametinger, 1995);
- query languages for programming environments (Horwitz, 1990b; Paul & Prakash, 1994a; 1994b; 1996); and
- query mechanisms for hypertext environments (Bertino *et al.*, 1988; Beeri & Kornatzky, 1990).

APPENDIX 2 Surveyed sources

This appendix lists the journals, bulletins and conference proceedings which have been checked in relation to the literature survey in Section 2.3, see (Koskinen, 1999d). In Table 3 the sources are ordered according to the number of relevant articles found. Second column denotes the status of the survey, the meanings being: * most of the 90's checked, + most of the recent (97-99) volumes checked, - volumes not checked systematically (citations received from elsewhere). The found articles have been classified. The meanings of the other column labels are as follows: PC (program comprehension issues), RE (reverse engineering issues), HT (hypertext issues), Gen. (general maintenance issues or surveys), Other (unclassified material), and Sum (total number of sources.

TABLE 3 Surveyed sources

Sources	**S**	**PC** 100	**RE** 279	**HT** 164	**Gen.** 65	**Oth.** 156	**Sum** 764
Journals		59	160	99	37	83	438
Communications of the ACM (CACM)	*	10	15	18	0	7	50
IEEE Transact. Software Engineering (IEEE TOSE)	*	5	26	1	3	15	50
Int. J. Human-Computer Studies (IJHCS/ IJMMS)	*	18	1	3	0	8	30
IEEE Software	*	2	11	6	2	7	28
Computer (IEEE Computer)	*	1	12	3	5	5	26
Information Processing and Management	*	1	0	19	0	5	25
J. of Software Maintenance (JSM)	*	3	12	1	2	6	24
Information and Software Technology	*	1	10	1	0	7	19
Software - Practice & Experience	*	0	14	2	0	2	18
ACM TOPLAS	*	0	15	0	0	0	15
The J. Systems and Software	+	6	7	0	0	0	13
ACM Transact. Information Systems (TOIS/ TOOIS)	*	0	1	9	1	1	12
J. Amer. Soc. for Information Science (JASIS)	*	1	1	6	2	2	12
J. Algorithms	*	0	1	3	0	6	10
Advances in Computers (AIC)	*	1	3	0	4	0	8
The Computer J.	+	0	1	4	0	2	7
ACM Tr. Software Eng. and Methodology (TOSEM)	*	0	6	0	0	0	6
ACM Computing Surveys	*	1	3	0	2	0	6
IBM Systems J.	*	0	0	1	3	1	5

continues

TABLE 3 continued

Sources (journals...)	S	PC	RE	HT	Gen.	Other	Total
Int. J. of Software and Knowledge Engineering	*	0	3	2	0	0	5
J. of Object-Oriented Programming (JOOP)	+	0	0	1	2	2	5
J. of the ACM	+	0	0	0	0	5	5
ACM SIGSOFT Software Engineering Notes (bulletin/excl. conf. proc:s)	+	1	0	0	4	0	5
Human-Computer Interaction	+	1	0	2	1	0	4
Interacting with Computers	-	0	0	3	1	0	4
IEEE Expert/ Intellig. Systems and their Application	+	0	0	3	0	1	4
ACM Letters on Progr. Lang. and Systems (LOPLAS)	*	0	3	0	0	0	3
Software - Concepts and Tools	*	0	3	0	0	0	3
Software Engineering J.	+	1	1	1	0	0	3
Structured Programming	*	0	3	0	0	0	3
ACM SIGPLAN Notices (bulletin/excl. conf. proc:s)	+	2	1	0	0	0	3
ACM Transact. Computer-Human Interaction (TOCHI)	*	0	1	1	0	0	2
---Other journals---	-	4	6	9	5	1	25

Proceedings		36	84	38	8	45	211
ICSM (Int. Conf. Software Maintenance)	*	6	24	2	2	3	37
IWPC (Int. Ws. Program Comprehension)	*	10	6	1	0	5	22
ICSE (Int. Conf. Software Engineering)	*	4	12	1	0	2	19
ESP (Workshop on Empirical Studies of Programmers)	*	8	1	1	0	2	12
HT (ACM Hypertext Conf.)	*	0	0	11	0	0	11
WCRE (Working Conf. Reverse Engineering)	*	3	7	0	0	1	11
SIGIR (ACM SIGIR Ann. Int. Conf. R. & D. Inf. Retr.)	*	0	1	5	2	3	11
POPL (ACM SIGPLAN Symp. Princ. of Progr. Lang.)	-	0	5	0	0	4	9
CHI (Conf. Human Factors in Computing Systems)	+	2	2	0	0	3	7
PLDI (ACM SIGPLAN Conf. Progr. Lang. Des. & Impl.)	+	0	7	0	0	0	7

continues

TABLE 3 continued

Sources (proceedings...)	**S**	**PC**	**RE**	**HT**	**Gen.**	**Other**	**Total**
ECHT (European Conf. Hypertext/ Hypermedia Techn.)	-	0	0	5	0	1	6
FSE (ACM SIGSOFT Int. Symp. Found. Sw. Engin.)	-	0	6	0	0	1	6
CASCON (IBM Centre for Advanced Studies Conf.)	-	1	2	0	0	2	5
HICSS (Hawaii Int. Conf. System Sciences)	-	0	2	3	0	0	5
ACM SIGSOFT/SIGPLAN Symp. Pract. Progr. Env.	-	0	3	0	0	0	3
ISSTA (ACM SIGSOFT Int. Symp. Sw. Test. and Anal.)	+	0	3	0	0	0	3
SIGMOD (ACM Conf. Management of Data)	-	0	0	0	0	3	3
OOPSLA (Conf. Object-oriented Systems and Lang.)	-	0	1	0	0	2	3
RIAO (Conf. Intellig. Text and Image Handling)	-	0	0	2	1	0	3
SEKE (Int. Conf. Software Eng. and Knowl. Eng.)	-	0	1	1	0	0	2
---Other proceedings---	-	2	1	6	3	13	20

Other sources		5	35	27	20	28	115
Books	-	0	7	6	17	15	45
Reports	-	2	24	7	0	4	37
Articles in books	-	3	0	9	3	3	18
Ph.D. theses	-	0	4	5	0	6	15

YHTEENVETO (FINNISH SUMMARY)

Tutkimuksen tavoitteena on kehittää hypertekstiperusteinen lähestymistapa vanhojen ohjelmistojen (legacy systems) ylläpidon tukeen sekä arvioida lähestymistapaa. Ylläpidon suuri kustannusvaikutus ohjelmistotuotantoon on yleisesti tunnettu. Ohjelmateksti on keskeinen resurssi pyrittäessä ymmärtämään ja ylläpitämään vanhoja ohjelmistoja käänteistekniikkatyökalujen avulla. Toisaalta, World Wide Webin kasvu kuvastaa hypertekstin tärkeyttä tekstimuotoisen informaation yleiskäyttöisenä esityskeinona.

Lähestymistavassa ohjelmatekstiä tarkastellaan väliaikaisena hypertekstinä, joka koostuu ohjelmanosista, joiden väliset linkit mahdollistavat nopean epälineaarisen selauksen. Väliaikaiset hypertekstihakurakenteet (transient hypertextual access structures; THAS) muodostetaan tyydyttämään ohjelmien ylläpitäjien tilannekohtaisia tietotarpeita. Hakurakenteiden automaattinen muodostus eliminoi työlään manuaalisen linkityksen, mikä on erityisen ongelmallista vanhojen järjestelmien ja usein muutettavan lähdekoodin ollessa kyseessä. Työssä on kehitetty tasoittainen HyperSoft -niminen malli tähän tarkoitukseen, toteutettu malliin perustuva työväline ja arvioitu sen oletettua hyödyllisyyttä empiirisesti.

Malli erottaa ohjelmatekstistä neljä tasoa: lähdekooditaso, syntaktinen taso, hakurakennetaso ja käyttöliittymätaso. Syntaktinen taso sisältää lähdekoodin jäsennyspuuna. Hakurakennetaso taas sisältää joukon erilaisia hakurakennetyyppejä, jakautuen viittauksiin, listoihin, joukkoihin, puihin ja yleisiin verkkoihin. Hakurakenteiden automaattinen muodostus perustuu ohjelma-analyysitekniikoiden soveltamiseen. Ohjelmanosien väliset linkit muodostetaan perustuen ohjelmariippuvuuksiin. Ohjelmariippuvuuksien olennaisten ominaisuuksien selvittämiseksi ne on karakterisoitu ja luokiteltu ohjelmaosien välisinä relaatioina. Ohjelma-analyysitekniikoita on kartoitettu.

Lähestymistavan toteutus on HyperSoft -järjestelmä, joka on kokeellinen ohjelmien ylläpidon tukiväline. Järjestelmän toteuttamista ohjasivat työhön liittyneen projektin yhteistyöryhmäyritysten (Nokia Tutkimuskeskus, Novo-ryhmä ja TT-Tieto) edustajat. Järjestelmässä tuettu kieli (C) ja toteutettu hakurakennejoukko on valittu yhteistyöyritysten tarpeiden perusteella. Toteutettu hakurakennejoukko sisältää määrittelyviittaukset, esiintymälistat, kutsukaaviot ja ohjelmaviipaleet.

HyperSoft -lähestymistavan, -järjestelmän ja toteutettujen hakurakennetyyppien hyödyllisyyttä työssä arvioidaan kolmella tavalla. Ensinnäkin, järjestelmä annetaan arvioitavaksi yhteistyöyrityksiin. Toiseksi, HyperSoft:in tarjoamia mahdollisuuksia verrataan aikaisemmissa empiirisissä tutkimuksissa esiin tuotuihin ohjelmien ylläpitäjien keskeisiin tietotarpeisiin. Kolmanneksi, hyödyllisyyttä arvioidaan tilastollisesti kahdessa erillisessä testisarjassa. Testisarjat vertailevat tietojenkäsittelytieteiden opiskelijoiden tiedonhakutehtävien suorituksen tehokkuutta käytettäessä HyperSoft:ia ja Borland C/C++ -ympäristöä. Tulokset tukevat selkeästi hypoteesiamme lähestymistavan hyödyllisyydestä. Tutkimuksessa tuodaan myös esille mahdollisen jatkotutkimuksen keskeiset kohdealueet ja niihin liittyviä tutkimusongelmia.

ORIGINAL ARTICLES

I

PROGRAM TEXT AS HYPERTEXT: USING PROGRAM DEPENDENCES FOR TRANSIENT LINKING

Koskinen, J., Paakki, J. & Salminen, A. 1994. In *Proceedings of the 6th International Conference on Software Engineering and Knowledge Engineering* (*SEKE'94*). Skokie, IL: Knowledge Systems Institute, 209-216.

(C) 1994, KSI. Reproduced with permission.

Program Text as Hypertext:
Using Program Dependences for Transient Linking

Jussi Koskinen, Jukka Paakki and Airi Salminen
Department of Computer Science and Information Systems
University of Jyväskylä
P.O.Box 35, SF-40351 Jyväskylä, Finland
email: {koskinen, paakki, airi} @jytko.jyu.fi

Abstract

Hypertext is text with nonlinear browsing capabilities. The program comprehension process typically involves viewing the source code in various nonlinear ways which hypertext representation can support. In this paper we will describe program text as a two-level structure consisting of a static, hierarchic structure and a dynamic access structure. In the access structure, the nonlinear browsing is supported by transient links created dynamically by the user during the work process. The paper explores the possibilities to create transient links based on well-known program dependences, and to use the links in the static and dynamic program analysis.

1. Introduction

Hypertext consists of text fragments called nodes. Links exist connecting these nodes so that data access is driven by the user viewing successive text fragments by following the links. The creation of hypertext means creating a set of nodes each containing a piece of text, and links between the nodes. The creation can be done basically in two different ways: either directly in a hypertext framework where text is handled as a set of nodes all the time, or by converting an existing linear text to hypertext. Considering software engineering environments, an example of the first approach is the HyperPro environment representing software as a set of nodes and links, instead of traditional files [29]. In this paper we are going to study the second approach: the possibilities to convert an existing program text to hypertext. In the conversion the key problem is to determine the text fragments to represent nodes and the links between them. Thus we are going to explore the potential hypertext nodes and links over a program text.

A hypertext conversion may be a single process creating from a given text a set of static nodes and links [11] or it may be a more dynamic process where the reader may specify a set of transient nodes and links during text reading for his or her current information needs [21], [24]. A static hypertext structure is possible if there is a clear fragmentation in the original text following the golden rules of hypertext [22]:

* there is a large body of information organized into numerous fragments,
* the fragments relate to each other, and
* the user needs only a small fraction at a time.

However, there seldom is one unique fragmentation and a unique set of links which is suitable for different readers in their different information needs. For example, [20] showed that the static fragmentation and linking of the Oxford English Dictionary was not possible.

Program text as an application area for hypertext conversions is interesting because program text fragments and their relationships to each other have been extensively studied within programming language research and there are lot of methods and tools for automatic recognition of the fragments and their relationships. These methods and tools offer support for the automatic conversion of program text to hypertext as well. In this paper we are going to describe a model for viewing program text as hypertext and to show what kind of hypertext structures may be created using well-known program dependences, i.e. relationships between program parts. We are not looking for a static set of nodes and links but different possibilities for dynamic specifications of structures to support hypertext access.

2. Why hypertext reading capabilities for program text

Software maintenance is the largest cost element in the life of a software system and the process of program comprehension takes about 50 % of the time spent on it [7]. Program comprehension is a prerequisite for various programming tasks and it is laborious while working with unfamiliar, poorly documented or large source code collections [8]. Thus there is a continuous and recognized need for automated support of program comprehension,

maintenance assistance, metrics and information abstraction.

Programmers tend to group program parts in a non-sequential order while attempting to understand programs [15]. Hypertext can aid program comprehension by providing an easy mechanism for a programmer to shift between program parts which create and satisfy information needs emerging during the work process. Relations between the parts are based on program dependences [e.g. 18] and their relevance depends on the task a programmer is working on. The selection of dependences to be supported and the way they are supported is important in order to avoid the problem of too intense link-structures in hypertext.

According to an evolutionary model of software development, such as the spiral model, source code is created, modified and maintained throughout the development process. The following traditional classification of maintenance types [e.g. 19] is relevant to both initial creation and enhancement of program text: 1) *corrective maintenance* (debugging) includes the diagnosis, localization and correction of errors based on effects they introduce, 2) *adaptive maintenance* includes modification of software to properly interface with a changing environment, 3) *perfective maintenance* includes addition of new capabilities, modifications to existing functions and general enhancements based on recommendations received from users, and 4) *preventive maintenance* comprises changes to improve the future maintainability or reliability, or to provide a better basis for future enhancements. All of these maintenance types include various common tasks related to the comprehension of program text and to the localization of its specific parts.

3. Two levels of program text: syntactic structure and access structure

Program text with hypertext access capabilities may be modelled as a two-level structure where the syntactic structure is separated from the access structure [21]. Program text with hypertext access capabilities is a triple (G, X, A) where G is a *context-free grammar*, X a *syntactic structure*, and A an *access structure.*

Syntactic structure

The syntactic structure X is a parse tree for the program with respect to the grammar G. (The basic notions concerning grammars and parse trees may be found e.g. in [1]). The linear character string representation of the whole program text is the string consisting of the terminal symbols in X, from left to right. Each nonterminal of the grammar represents a set of text entities in X. Therefore, a nonterminal is called a *text type*. For example, we may have text types like *statement*, *type_declaration*, or *identifier* if these symbols appear as nonterminals in the grammar. The text entities associated with a text type *t* in the syntactic structure X are called parts of type *t* and they are represented by nonterminal nodes (and the respective subtrees) in X. Since we wish each of the parts correspond to an identifiable substring in the character string representation of the whole program text, we regard a nonterminal node *n* a *part* of X only if it is not a single child of its parent. The string produced by concatenating the terminal symbols of the subtree with a part *n* as its root (from left to right) is the *value* of *n*. The nonterminal node labels indicate types of parts such that the label of a single child of a parent renames a part.

Access structure

For browsing purposes hypertext is often modelled as a directed graph, i.e. a pair (Z, E) where Z is a set of elements called *nodes* and E a set of node pairs called *links*. For more general data access capabilities hypertext has been modelled by a *hypergraph* [21], [25], [26]. A hypergraph is also a pair (Z, E), where Z is a set of nodes. The members of E are however now any subsets of Z, not only binary subsets. We will use the term *edge* for the members of E in a hypergraph and the term *link* for binary edges. An edge represents a relationship between a number of nodes. For example, in an access structure of a program text there may be an edge consisting of all output statements of the program. All nodes of the edge share the common property that they are of the text type *output_statement*. If a hypergraph is regarded as a state of a search session then it may be called a *transient hypergraph* [26]. During a session, the user changes the hypergraph by generating new edges corresponding to his or her information needs.

In this paper we consider the access structure A of a program text (G, X, A) as a transient hypergraph whose nodes are parts of the syntactic structure X. The nodes of the access structure thus stand for subroutines, variables, declarations, statements, etc. The access structure may be more or less static but the idea behind the model is to allow the reader to change the access structure dynamically e.g. by specifying a new access structure as a special kind of graph, or by extending an old access structure by new edges.

4. Access structures for static analysis

Analysis of computer programs is a well-established field with a large number of powerful methods, algorithms, and tools. The analysis activities can be roughly divided into two categories: *static analysis* which is made without

running the program, and *dynamic analysis* which is made interleaved with the execution. Static analysis involves typical compiler-oriented tasks, such as lexical analysis (scanning), syntax analysis (parsing), name analysis, and type resolving and checking. Dynamic analysis also covers certain validation activities, such as checking of array indices and data references, but also tasks pertaining to the run-time behavior of the program. These include e.g. tracing of data and control flow, and bookkeeping of variable updates. Dynamic validation activities are necessary in cases when they cannot be done statically by the compiler, while behavioral tasks are needed when the user wants to experiment with the program, a typical example being the debugging of program errors.

Results obtained from static analysis are valid for each execution of the program, and therefore they need to be computed only once for a program. The price of such generality is the imprecise character of the analysis. Dynamic analysis, on the other hand, only concerns one particular execution of the program with precise results. As a drawback, this approach inevitably introduces some run-time overhead due to separately computing the statistics for each execution of the program.

In this section we discuss standard concepts belonging to the static program analysis category, and in the next section we will explore the dynamic analysis category. We concentrate especially on such concepts that are of obvious use when considering programs as a hypertext structure with graph-like features. The presented graph structures are well-known to the extent that they are used in standard literature on programming, compiler construction, and software engineering. For a broader discussion on the general applications of the graphs, refer e.g. to [1] and [19].

4.1. Structural links

The syntactic structure of programs is conventionally described as a *parse tree*. A parse tree represents the hierarchy of language elements in a program, according to a context-free grammar. A parse tree (or its compressed variant, an *abstract syntax tree*) is the central data structure underlying any language processing task. Hence, it is a natural choice to consider the parse tree also as a basis for a hypertext structure over a program.

Usually a maintainer is not interested in the whole program but instead wants to focus his/her attention on some relevant entities only. One straightforward way to support this is to restrict the nodes of the access structure to be of some specific type. Then a (preorder, postorder) traversal over such a partial tree would yield a focused and abstracted view of the program. An example case which is typical in software engineering would be to select the parts standing for modules as the access nodes, giving an overall design architecture of the program. In general, this restriction facility provides a means to produce a hierarchic abstraction of the program in terms of an arbitrary text type.

The problem of building a parse tree for an input according to an underlying context-free grammar has been one of the most important issues in computer science, the reason being its central role in any kind of language processing. Thus, there exists a large variety of parsing algorithms that can be employed as a basis for constructing a hypertext access structure. The hypertext construction (e.g. with vertically and horizontally restricted parts) can be made either incrementally, interleaved with the parsing process, or as a separate phase, after building the parse tree by a parser. In this paper we do not further discuss the algorithmic aspects of constructing a parse tree -based access structure, but refer to standard literature on parsing, such as [1].

4.2. Module dependence graphs

An important aspect of software quality is *modularity*; that is, the construction of software as an architecture of independent and yet interrelated components. An effectively modular architecture provides a solid framework for the whole software engineering process, from analysis down to testing and maintenance.

Effectivity of modularity can be measured with two conventional criteria, *cohesion* and *coupling*. Cohesion characterizes the relative functional strength of a module, and coupling measures the relative interdependence among modules. A cohesive module ideally performs just one task; hence, high cohesion is preferred in software engineering. Coupling, on the other hand, should be low since in that case modules are just loosely dependent on each other's services and thus more autonomous and easier to manage.

While the concepts of cohesion and coupling are rather abstract, it is possible to approximate them quantifiably. A metric system, as demonstrated for cohesion e.g. in [6] and for coupling e.g. in [16], can then expose those modules that have been poorly designed or implemented and that should be rebuilt for better testability and maintainability.

Since cohesion and coupling are related to module collaboration, it is essential to grasp the dependences between program modules. These dependences take the form of a directed graph whose nodes are modules and whose links are the client-supplier dependences. Hence, the term *module dependence graph* (or "package diagram" [4]) can be coined for this structure. The module dependence graph can be constructed automatically for a given program and it can be directly employed as a

hypertext access structure. Besides providing a basis for an analysis of coupling, such an access structure is useful in ordinary maintenance tasks as well since it exhibits the dependences among the software modules and their externally relevant components. For instance, such an information is most valuable in designing a focused black box strategy for integration and validation testing in connection with a maintaining change ovar a certain set of modules.

5. Access structures for static and dynamic analysis

In the previous section we presented graph structures originally defined for static analysis of programs. When considering the different maintenance classes introduced in Section 2, the access structures for static analysis most notably support preventive maintenance for improving future maintenance and reliability. The construction process of the access structures can be characterized as *reverse engineering* or *re-engineering*, two paradigms associated with the preventive maintenance class. For an overview of these techniques, refer to [19].

In this section we discuss graph structures that not only concern static properties of programs but its dynamic aspects as well. That is, the graph structures represent such relationships and dependences between program parts that approximate the run-time behavior of the program. When reflecting these graphs and the corresponding access structures into the maintenance classification, support for corrective and perfective maintenance especially is provided. From a more concrete point of view, these access structures for dynamic analysis most notably assist the debugging phase of maintenance, providing facilities both for locating a bug in the program and for preventing the introduction of new bugs due to correcting an old one.

5.1. Call graphs

Call graphs are a standard structure describing the run-time behavior of programs. A call graph abstracts program execution by representing it on the level of procedure and function calls and their interdependences. When associated with an underlying parse tree, a call graph includes a link from each node standing for a procedure call to the node standing for the declaration of the called procedure. Such a graph makes it possible to trace the execution of the program forwards in terms of the sequence of procedure calls.

Call graphs are utilized in a number of maintenance support tools, e.g. [3], [7], [17]. A hypertextual access structure generated from a call graph makes it possible to understand the overall execution pattern of the program, a prerequisite for any maintenance activity. A two-directional annotated linking structure provides for both forward and backward tracing of interprocedural execution with also a possibility to study the parameter passing strategy. Moreover, a forward access policy makes it possible to analyze how the execution behavior would be affected by an update within a certain procedure. An example of a call graph is given in Figure 1a.

5.2. Graphs representing control and data flow information

The two most important dependences existing between program parts are *control flow* and *data flow dependences*. Control dependences are features of a program's control structure, and data flow dependences are features of its use of variables [18]. Control dependences are determined by using control flow analysis and data dependences by data flow analysis [e.g. 1] or incremental data flow analysis [9] of program text. These dependences can be represented in various graphs.

A *data flow graph* can be used e.g. in optimization. In program design the analogous form is a *data flow diagram* which typically depicts interprocedural data flow information. These forms have their limitations because of the difficulties to relate them with the control flow information representations.

A *control flow graph* is a directed graph whose nodes represent program statements/parts and whose links represent possible transfers of control between them. In program design the equivalent form is known as a *flow chart.* Control flow could be represented as the access structure of hypertext so that nodes correspond to statements/statement blocks and links correspond to the transfer of control. The analogy to hypertext is seen in e.g. [1, p. 591]. Further, control flow may be structured or non-structured (containing gotos, breaks and exits).

Possible hypertext access structures can be derived from the graphs described above either by first choosing a set of nodes to create an edge and then ordering the nodes of the edge (e.g. on linear or execution order), or by creating a set of new links directly between the nodes of the graph.

5.3. Graphs created by slicing

Experience has shown that even the abstracted representations for programs, such as parse trees and graphs presented above, tend to become very large for sizable programs. Too large a volume of support data soon becomes a burden on the maintainer to extract the specific information that is needed in the particular activity at hand. That is why there must be some way to focus one's

```
class deque {
  Type* seq;
  Type size,left,right;
  void underflow() {};
  void overflow() {};
public:
  deque(Type sz=STACK_DEF_SIZE);
  void insert_left(Type item);
  void insert_right(Type item);
  Type remove_right();
};

class stack:private deque {
public:
  stack(Type sz=STACK_DEF_SIZE):
                    deque(sz) {}
  void push(Type item)
    {deque::insert_right(item);}
  Type pop()
    {return deque::remove_right();}} ;

deque::deque(Type sz) {
  seq=new Type[sz+1];
  size=sz;left=right=0;}

void deque::insert_right(Type item) {
  if(right<size) right++; else right=0;
  if(left==right) overflow();
  seq[right]=item;}

Type deque::remove_right() {
  if(left==right) underflow();
  Type item=seq[right];
  if(right>0) right--; else right=size;
  return item;}

int main(void) {
  int i,s=STACK_SIZE, error_cnt=0;
  Type a[ARRAY_SIZE];
  stack sta(s);

  for(i=0;i<=ARRAY_SIZE-1;i++) {
    a[i]=pow(2,i); sta.push(a[i]);
  }
  for(i=ARRAY_SIZE-1;i>=0;i--) {
    if((sta.pop()) != a[i])
      ++error_cnt;
  }
  return error_cnt;
}
```

(1a) Call graph

```
class deque {
  Type* seq;
  Type size,left,right;
  void underflow() {};
  void overflow() {};
public:
  deque(Type sz=STACK_DEF_SIZE);
  void insert_left(Type item);
  void insert_right(Type item);
  Type remove_right();
};

class stack:private deque {
public:
  stack(Type sz=STACK_DEF_SIZE):
                    deque(sz) {}
  void push(Type item)
    {deque::insert_right(item);}
  Type pop()
    {return deque::remove_right();}} ;

deque::deque(Type sz) {
  seq=new Type[sz+1];
  size=sz;left=right=0;}

void deque::insert_right(Type item) {
  if(right<size) right++; else right=0;
  if(left==right) overflow();
  seq[right]=item;}

Type deque::remove_right() {
  if(left==right) underflow();
  Type item=seq[right];
  if(right>0) right--; else right=size;
  return item;}

int main(void) {
  int i,s=STACK_SIZE, error_cnt=0;
  Type a[ARRAY_SIZE];
  stack sta(s);

  for(i=0;i<=ARRAY_SIZE-1;i++) {
    a[i]=pow(2,i); sta.push(a[i]);
  }
  for(i=ARRAY_SIZE-1;i>=0;i--) {
    if( sta.pop()) != a[i])
      ++error_cnt;
  }
  return error_cnt
}
```

(1b) Forward slice

Figure 1. Access structures for a C++ program

attention to that sub-information only that is relevant for solving the current maintenance problem.

Program slicing is one powerful automated method that has been suggested for minimizing a program representation. As proposed already in the original introduction of slicing [27], the method has mostly been applied in debugging for extracting a piece of the program as the suspect for an externally visible symptom of a bug. Later adaptations of slicing within the debugging area are presented e.g. in [10].

Intuitively, a program slice (in its original meaning) consists of all those statements of a program that might affect the value of a given variable at a given program point. When integrated with debugging, the slice is constructed with respect to an output variable whose value has been manifested as being incorrect at some externally observable program point, such as the end of the program. Since the process starts from the result and flows backwards with respect to the execution, the method is called *backward slicing*. An alternative scheme is *forward slicing* that extracts those statements of a program that might be affected by a variable (at some program point). When related to the maintenance classification, slicing therefore most naturally fits with the corrective and perfective maintenance categories. General ideas of applying slicing in program maintenance are discussed e.g. in [18] and in [12]. Special application areas, in addition to debugging, include e.g. analysis of cohesion, static testing, regression testing, incremental testing, and program integration.

Slices can be constructed statically for all the executions of the program or dynamically for one particular execution only. Usually a slice is generated from a unified program representation, a *program dependence graph* [13]. The nodes of a program dependence graph represent the statements and predicates (e.g. the control expressions of conditional statements). The links represent several kinds of control and data dependences.

While a slice has originally been defined in terms of a variable (and some program point), the concept can be rather easily and naturally generalized. Such an extended view is taken e.g. in [2] where a slice is defined in terms of any exported component of a module, such as a procedure, providing assistance for optimal program understanding at preventive maintenance and reverse engineering. In the same style, the *slicing criterion* can actually be defined as a transitive property of any program component. From a simplistic graph-oriented point of view, a slicing criterion can be given as any node in a directed graph, and a slice generated according to that criterion is the transitive closure of the graph over that node. When reflecting this view into the dual slicing classification, a backward slice includes all those nodes of the graph that have a directed path *to* the indicated criterion node, and a forward slice contains all those nodes that reside on a directed path *from* the criterion node.

Such a general view provides for employing slices as a hypertextual access structure for a broader class of maintenance tasks than mentioned above. For instance, a forward slice over a node for a type definition would consist of all those statement nodes that contain an element of that type and that should therefore be checked when modifying the type definition.

As an example Figure 1 shows two access structures for hypertext, created from a call graph (Figure 1a) and from a forward slice (Figure 1b) over a C++ program. The call graph illustrates how the call of procedure *push* (indicated with an emphasized oval) propagates through the program via other procedure calls. The access structure can be used e.g. to extract the functional decomposition of the procedure *push* during preventive maintenance, and to trace its behaviour during corrective maintenance. The forward slice, with the emphasized part as the slicing criterion, shows how the *item* value produced in the assignment statement propagates through the program. Hence, traversing this access structure makes it possible to analyze how a maintaining modification on the indicated statement would affect the rest of the program.

5.4. Class hierarchies in object-oriented programs

The object-oriented programming paradigm is most notably founded on the principles of *classification* and *inheritance*. These concepts make the object-oriented approach a most powerful one in abstracting common behavior as a collection of reusable and related classes. Software economy is enhanced by (ideally) describing each property in a single (super)class only and by directly inheriting it to those related (sub)classes that also share that property.

An access structure based on class hierarchies collects the program parts for related classes into a graph. Since a class can be considered as a variant of a (static) module, this facilitates the analysis of a general software architecture in the manner discussed in Section 4.2. However, in contrast to conventional modules, classes have dynamic properties as well since they give rise to active *objects* during the execution of the program. Hence, an access structure founded upon the class hierarchy provides for a more flexible maintenance framework than that based on static modules.

One well-known special problem encountered in object-oriented programming is the so called "yoyo" phenomenon [23]. This happens when an inherited

operation is applied on an object, and the execution of that operation sends the control back to the object itself using the "self" ("this") reference associated with another inherited operation, etc. Resolving of inherited operations traverses the class hierarchy bottom-up (that is, from subclasses towards superclasses), whereas following the self references makes the traversal flow top-down. As a consequence, the resolving process moves like a yoyo up and down over the class hierarchy.

Understanding the behavior of a program is hard if it exhibits the "yoyo" phenomenon and the class hierarchy is deep. The access structure in hypertext can be tuned to support the analysis of this anomaly by including two-directional links between the class parts. In that way both top-down and bottom-up analysis of the problematic class hierarchies is supported.

6. Access structures based on common properties of text parts

In Sections 4 and 5 we considered potential access structures which were different graph-like representations of programs. In Section 4.1 we mentioned the need to restrict the nodes of a graph structure e.g. to be of a specific text type. By modelling the access structure as a hypergraph we have a chance to use an edge to represent the set of nodes relevant to a reader at a moment. The edge may then be further used to create a transient linking for browsing purposes, either automatically or by the specification of the reader. In the following paragraphs we will consider some properties for parts of potential interest in specifying edges. At the end of the section we will discuss ways to use the edges to create transient linking. By a *property* we mean a predicate whose truth value for a part in a parse tree may be determined by an algorithm. As predicates properties may of course be combined with logical operators. On the other hand, specified edges can easily be handled with set operations.

There is quite a number of different properties which may be specified for parts of any structured text (i.e. text defined by a grammar; for example, SGML text). Examples of this kind of general properties are the above mentioned properties testing the text type of a part, or properties testing the value of a part. With these properties a programmer is able to create e.g. an edge for all occurences of a given identifier. For preventive maintenance and restructuring of a program it is important to identify the textual similarities of program parts. Identical parts should be abstracted so that needless redundancy is eliminated. Similarity properties could be defined based on the values of parts.

Properties of usual interest in structured text concern the containment structure. (A part can be defined to be contained in another part if it is a node in the subtree whose root the other part is.) The properties are needed e.g. for finding specified parts inside or outside some other specified parts. Properties testing the level of internesting of parts may be defined for any structured text but in case of program text they are especially interesting. With such properties e.g. innermost loops, having potential time-criticality, can be identified for optimization.

Complexity metrics and algorithms, such as described in [14] have been developed to determine program parts that are excessively complex. With these metrics the complexity may be used as a criterion for finding parts needing redesign. Also algorithms for identifying language specific, non-standard features have been developed [19]. Such features may cause problems e.g. in porting. In a software engineering environment supporting the identification of the parts with non-standard features the programmer may properly comment these parts.

After an edge consisting of a number of nodes has been specified, there are, in principle, different ways for creating transient links between the nodes. For example, it is possible that the edge is applied as a filter to a given graph. The result of the filtering is a new graph whose nodes are those nodes of the old graph which also occur in the edge. For determining the links of the new graph there are two straightforward methods. First, if the old graph is a chain (consisting of connected links), then the new graph is a chain where links occur between the chosen nodes. Another possibility is to choose the links of the new graph as those links of the old graph whose both nodes appear in the filter. The first case means ordering the nodes of the edge according to the order specified in the old graph. There are of course also other possibilities to specify an order on the nodes of an edge and thus to create linking between the nodes. For example, the programmer might have a possibility to order, and thus also to browse, a set of subroutines in the alphabetic order of their names. In using the complexity of program parts as a selection criterion, the order of the nodes could be based on the complexity metrics values. In many cases a useful order is the textual order of parts.

7. Related and future work

Many of the current program maintenance environments and tools include capabilities to find some of the program dependences and to support program comprehension. Thus many of the dynamic access structures proposed in this paper have already been implemented. Related works include: [3], [5], [7], [17], [28], and [29]. Our goal however is to build an environment according to the

two-level model. The implementation will be made using some appropriate metacompiler based on attribute grammars. The environment is intended to be such that it will make possible to test different access structures, their implementation and use for different program maintenance tasks.

8. Summary and conclusions

We described program text as a two-level structure consisting of a syntactic structure defined by the programming language grammar and an access structure defined dynamically by the reader of the program text. The syntactic structure was a hierarchy of parts and the access structure was a hypergraph, consisting of nodes and edges. The edges of the hypergraph were any sets of nodes, not only binary sets (i.e. links). We explored the potential of the well-known program dependences to be used for creating access structures. First we considered the known graph structures originally introduced in the programming language literature as potential access structures. Secondly we discussed using common properties of program parts as criteria in specifying an edge for an access structure, and different ways to create transient links from such an edge.

For programming environments the hypertext approach based on the two-level model is promising because automatic transformation of program text into hypertext is possible by using the syntax of the programming language together with the methods and tools developed for the automatic program analysis. Our approach answers to the request for task-orientation by making it possible to use different access structures for different information retrieval purposes, both for general comprehension support and for localization of interesting program parts.

9. References

[1] Aho, A.V., Sethi, R., & Ullman, J.D., "Compilers - Principles, Techniques, and Tools", Addison-Wesley, 1986.

[2] Beck, J. & Eichmann, D., "Program and Interface Slicing for Reverse Engineering", In: Proc. 15th Int. Conf. on Software Engineering, Baltimore, Maryland, IEEE Computer Society Press, 1993, 509-518.

[3] Bigelow, J. & Riley, V., "Manipulating Source Code in DynamicDesign", In: Hypertext '87 Papers, 397-408.

[4] Booch, G., "Software Engineering with Ada", (2nd ed.), Benjamin-Cummings, 1986.

[5] Brown, P., "Integrated Hypertext and Program Understanding Tools", IBM Syst. J. 30, 3, 363-392.

[6] Calliss, F. & Cornelius, B., "Potpourri Module Detection", In: Proc. 1990 Conf. on Software Maintenance, IEEE Computer Society Press, 1990, 46-51.

[7] Cleveland L., "A Program Understanding Support Environment", IBM Syst. J. 28, 2, 324-344.

[8] Corbi, T., "Program Understanding: Challenge for the 1990's", IBM Syst. J. 28, 2, 294-306.

[9] Ferrante, J., Ottenstein, K. & Warren, J., "The Program Dependence Graph and Its Use in Optimization", ACM Trans. Program. Lang. Syst. 9, 3, July 1987, 319-349.

[10] Fritzson, P., Gyimothy, T., Kamkar, M., & Shahmehri, N., "Generalized Algorithmic Debugging and Testing", In: Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation, Toronto, Ontario, SIGPLAN Notices 26, 6, 317-326.

[11] Furuta, R., Plaisant, C., & Shneiderman, B., "Automatically Transforming Regularly Structured Linear Documents into Hypertext", Electronic Publishing 2, 4, (Dec. 1988), 211-229.

[12] Gallagher, K.B. & Lyle, J.R., "Using Program Slicing in Software Maintenance", IEEE Trans. Softw. Eng. 17, 8, (Aug. 1991), 751-761.

[13] Horwitz, S. & Reps, T., "The Use of Program Dependence Graphs in Software Engineering", In: Proc. 14th Int. Conf. on Software Engineering, Melbourne, Australia, 1992, IEEE Computer Society Press.

[14] Kafura, D. & Reddy, G., "The Use of Software Complexity Metrics in Software Maintenance", IEEE Trans. Softw. Eng. 13, 3, (Mar. 1987), 335-343.

[15] Letovsky, S. & Soloway, E., "Delocalized Plans and Program Comprehension", IEEE Software 3, 3, 41-49.

[16] Offutt, A.J., Harrold, M.J., & Kolte, P., "A Software Metric System for Module Coupling", J. Syst. Softw. 20, 3, (1993), 295-308.

[17] Oman, P., "Maintenance Tools", IEEE Software, 1990 (5), 59-64.

[18] Podgurski, A. & Clarke, L.A., "A Formal Model of Program Dependences and its Implications for Software Testing, Debugging and Maintenance", IEEE Trans. Softw. Eng. 16, 9 (Sep. 1990), 965-979.

[19] Pressman, R.S., "Software Engineering - A Practitioner's Approach" (3rd ed.), McGraw-Hill, 1992.

[20] Raymond, D.R. & Tompa, F. Wm., "Hypertext and the Oxford English Dictionary", Commun. ACM 31, 7, (July 1988), 871-879.

[21] Salminen, A. & Watters, C., "A Two-level Structure for Textual Databases to Support Hypertext Access", J. American Society for Information Science 43, 6 (July 1992), 432-447.

[22] Shneiderman, B., "Reflections on Authoring, Editing, and Managing Hypertext", In: The Society of Text, Barret, Ed. (ed.), MIT Press, Cambridge, MA, 1989, 115-131.

[23] Taenzer, D., Ganti, M., & Podar, S., "Object-Oriented Software Reuse: The Yoyo Problem", J. Object-Oriented Program. 2, 3, (1989), 30-35.

[24] Tague, J., Salminen, A. & McClellan, C., "A Complete Formal Model for Information Retrieval Systems", Proc. ACM SIGIR'91, ACM-Press, 14-20.

[25] Tompa, F. Wm., "A Data Model for Flexible Hypertext Database Systems", ACM Trans. on Inf. Syst. 7, 1 (Jan. 1989), 85-100.

[26] Watters, C.R, & Shepherd, M.A., "A Transient Hypergraph-based Model for Data Access", ACM Trans. on Inf. Syst. 8, 2 (April 1990), 77-102.

[27] Weiser, M., "Programmers Use Slices When Debugging", Comm. ACM 25, 7, (July 1982), 446-452.

[28] Wilde, N. & Thebaut, S., "The Maintenance Assistant: Work in progress", J. Syst. Softw. 9, 1, 3-17.

[29] Østerbye, K. & Nørmark, K. (1993) "The Vision and the Work in the HyperPro Project", Technical report, Institute for Electronic Systems, Department of Mathematics and Computer Science.

II

HYPERSOFT: AN ENVIRONMENT FOR HYPERTEXTUAL SOFTWARE MAINTENANCE

Salminen, A., Koskinen, J. & Paakki, J. 1994. In B. Magnusson, G. Hedin & S. Minör (Eds.) *Proceedings of the Nordic Workshop on Programming Environment Research* (*NWPER'94*). LU-CS-TR: 94-127. Lund, Sweden: Lund Univ., 25-37.

Reproduced with permission.

HyperSoft: An Environment for Hypertextual Software Maintenance

Airi Salminen, Jussi Koskinen, Jukka Paakki
Department of Computer Science and Information Systems
University of Jyväskylä
P.O. Box 35, SF-40351 Jyväskylä, Finland
email: {airi, koskinen, paakki} @jytko.jyu.fi

Abstract
HyperSoft, an environment for program maintenance is introduced. In HyperSoft, program text is viewed as a two-level structure consisting of a static, hierarchic, syntactic structure and a dynamic access structure. HyperSoft is directed to supporting program comprehension by providing a maintainer with capabilities to view programs as hypertext and to dynamically (during the session) specify the required access structures. The access structures most notably assist systematic program maintenance by providing a flexible navigation over related program parts. In this paper the emphasis is laid on the architectural and technical aspects of the environment.

1. Introduction and background

Software maintenance is the largest cost element in the life of a software system. Program comprehension is a prerequisite for various maintenance and programming tasks and it is laborious while working with unfamiliar, poorly documented or large software. Thus there is a continuous and recognized need for automated support of program comprehension and maintenance.

Hypertext is text with non-linear browsing capabilities. It consists of text fragments called *nodes* and *links* connecting these nodes. Data access is driven by the user who is viewing successive text fragments by following the links. Hypertext provides an easy-to-use mechanism to view related (and possibly very distant) program fragments together on a screen. It is a natural way of complementing the linear program representation because programmers tend to group program parts in a non-sequential order while trying to understand programs [Wei82, LeS86]. Especially in the case of old and large (legacy) systems the source code is often the only accurate description of the system. This means that program comprehension needs to be done solely on source code viewing and browsing. Attempts to comprehend programs take up to half of the time spent on software maintenance [Cle89], emphasizing the need for automatic support.

Viewing program text as hypertext is especially interesting because program text fragments and their relationships to each other have been extensively studied within programming language research, and because there are lots of methods and tools for automatic recognition of the fragments and their relationships. Therefore, in the HyperSoft method the approach is to automatically extract relevant relationships between the elements of a source program, and to evolve these relationships into a hypertextual access structure.

The HyperSoft method was originally introduced in [KPS94] and [PSK94]. HyperSoft is aimed at supporting program *comprehension and maintenance* by enhancing hypertextual source code viewing and browsing. The rest of the paper is organized as follows. The HyperSoft method is briefly discussed in Section 2. The general architecture of the

HyperSoft system is described in Section 3, followed by a more detailed description of its two central components, the program data base (Section 4) and the generic user interface (Section 5). Related work is discussed in Section 6, and conclusions are drawn in Section 7.

2. The HyperSoft method

Although the hypertext approach, i.e. regarding text as a set of fragments and links between them, is clearly applicable on program text [cf. Shn89], one major problem is the fact that there is no single universal fragmentation nor a unique set of links which would be suitable for all the possible maintenance tasks. Therefore, in the HyperSoft method program text is considered as *dynamic hypertext*. This means that the maintainer may dynamically specify a set of transient nodes and links to meet his or her current information requests and to view the source code collection accordingly.

A hypertext structure is usually considered as an alternative, or an addition, to a linear text structure. From the point of view of program understanding, the most important structure is the hierarchic structure of the program, defined by the grammar of the programming language. Therefore, in the HyperSoft method we model program text with hypertext access capabilities as a two-level structure where the hierarchic syntactic structure is separated from the access structure [SaW92].

Syntactic structure

The syntactic structure of our hypertext is a parse tree for the program with respect to its context-free grammar. The linear character string representation of the whole program text is the string consisting of the terminal symbols of the parse tree, from left to right.

Each nonterminal of the grammar represents a set of program text parts. Therefore, we call a nonterminal of the grammar a *text type*. In a C grammar, for example, there may be text types 'program', 'function-definition', and 'declaration'. Given any C program, each of the names stands for a set of text parts. In the parse tree, the parts are represented by nonterminal nodes (and the respective subtrees). Each program contains one part of type 'program' which is represented by the root node of the corresponding parse tree. Since we wish each part in the parse tree correspond to an identifiable substring in the program text, we regard a nonterminal node a *part* only if it is not a single child of its parent. The labels of nonterminal nodes indicate types such that the label of a single child of a parent renames a part.

Access structure

For browsing purposes hypertext is often modelled as a directed graph, i.e. a pair (Z, E) where Z is a set of elements called *nodes* and E is a set of node pairs called *links*. For more general data access capabilities, hypertext has also been modelled by a hypergraph [SaW92]. A hypergraph is also a pair (Z, E), where Z is a set of nodes. The members of E are however now any subsets of Z, not only binary subsets. We will use the term *edge* for the members of E in a hypergraph and the term link for binary edges. An edge thus represents a relationship between a number of nodes.

In HyperSoft, the hypertextual access structure of a program text is a hypergraph whose nodes are parts in the syntactic structure of the program. The hypertext nodes thus stand for such program elements as variables, declarations, statements, etc. Some portion of an access structure may be created at the time of program parsing. The approach behind the HyperSoft method is, however, that of dynamic hypertext allowing the reader to change the

access structure dynamically, e.g. by specifying a new access structure as a special kind of graph or an edge consisting of nodes sharing a common property. Since edges are sets, set operations may be applied to them. At the moment of browsing, the current access structure is a directed graph.

Automatic generation of hypertext

The program analysis methodology provides for a large selection of techniques that can be applied for *automatically* creating different kinds of hypertextual access structures to support different kinds of maintenance tasks. Notice the emphasis on flexible and *dynamic* creation of the relevant access structure(s): instead of restricting to a fixed and unadjustable access structure (as in the manual case), the automated method makes it possible to obtain exactly that (and only that) access structure which most properly fits with the particular maintenance task at hand.

The idea behind applying program analysis techniques on the creation of a hypertext structure is rather simple: well-known *program dependences*, i.e. relationships between program parts, are extracted either as a set or as a graph. We have investigated graph structures that, when represented in a hypertext format, can be applied as a navigation tool in common maintenance tasks. These graphs are standard tools within the disciplines of programming, language implementation, and general software engineering [Pre92, ASU86]. Graph-like structures that are generally used to represent programs and whose implications to hypertext formation we have analyzed in [KPS94] include the following:

* parse tree (as a general framework for program analysis and navigation),
* call graph (representing possible chains of procedure calls),
* control flow graph (representing possible execution orders of statements),
* definition/usage graph of program variables (linking defining occurrences of symbols with applied ones),
* data flow graph (demonstrating the propagation of data through the program),
* program dependence graph (unifying control and data flow),
* forward slice (showing how program statements are affected by a computed data value),
* backward slice (showing which statements may affect a specific data value),
* module dependence graph (specifying the static software architecture),
* class hierarchies in object-oriented programs (representing the inheritance relationships between the objects in the program).

In addition to the well-known graph structures, there are program dependences determined simply by a common property shared by a set of parts. The property may be, for example, the type of the parts, the complexity metric of the parts, or the layout of the parts. Several of such potentially interesting properties are studied in [KSP94] as well as ways to use an edge as a basis for a new graph.

The most useful access structures to purposes of the industrial partners we are collaborating with include: forward slices, call graphs, input/output access structures, and backward slices. Therefore we are first concentrating on these in our implementation of HyperSoft. In order to automatically generate the access structures, program parts belonging to a certain fragmentation and relations between these fragments need to be identified. We will employ the existing algorithms [Ryd79, HaS90, HoR90, HoR92, HMR93, Kam93] to form these structures, when applicable. These graphs may serve as our access structures as such or after some modifications.

3. The system architecture

The HyperSoft method is founded on the combination of static program analysis and dynamic hypertext access. The method is intended to support software maintenance; hence the capabilities to change the program have to be included in a HyperSoft maintenance environment. In such an environment, the source code update may be more or less integrated with the hypertextual program reading. In the first HyperSoft environment we are designing, the source code will be edited using a general purpose text editor. While the HyperSoft method is language independent, the environment will support the maintenance of software systems whose implementation language is C with or without embedded SQL fragments.

The general architecture of the HyperSoft environment is shown in Figure 1. It consists of three process components: Static Program Analyzer (SPA), Generic User Interface (GUI), and Editor. SPA analyses the source code collection using the C and SQL grammars, and stores the analysis information in the Program Data Base (PDB). The implementation of SPA is carried out with automated tools, based on the principles mentioned in the previous section. GUI is used by the maintenance programmer for source code reading. The maintainer will start the reading by specifying which category of access structures shall be provided to support the current maintenance activity. Genericity in the user interface component means that the current access structure category determines the specific user interface to be used for navigation. An access structure request causes the update of the PDB. For the source code changes general purpose editors will be available within the environment. After the changes in the code, SPA is used to update the PDB. A more detailed description of PDB and GUI will be given in the following sections.

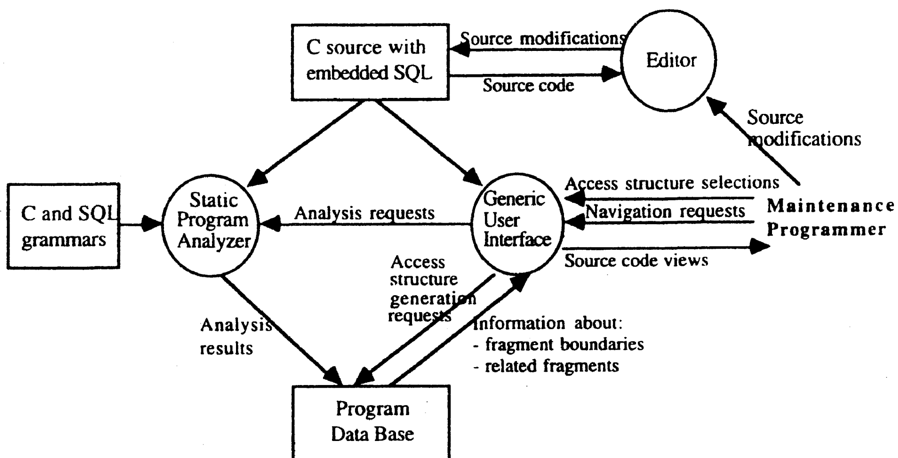


Figure 1. General architecture of HyperSoft.

4. The program data base

The program data base component (PDB) of the HyperSoft system stores the information necessary for dynamically creating and traversing the access structures for the program under maintenance. Since PDB captures static properties of a program, it closely resembles an architecture found in conventional compilers: the syntactic structure of the program is presented as a *parse tree* (for each program file), and the static semantic properties are stored in a *symbol table*. The third central component of PDB is a selection of *access structures* one of them being considered as the currently active one. Finally, the *program text* must be connected with the actual PDB so as to provide actual program views to the user.

Since a complete exhaustive analysis of the program for each dynamic access structure would obviously be too expensive in practice, the parse tree and the symbol table will be available in the program data base for the whole maintenance session. They are created while constructing the first access structure by a user's request, and will be consulted for constructing the subsequent ones due to additional needs of information. In other words, once created, PDB will contain all the information that is needed for constructing the requested access structures, without having to re-invoke the exhaustive static analyzer. Modifications in the program will be reflected into the persistent data structures efficiently as incremental updates.

It is also possible to have some central access structures, such as architectural descriptions of the program, constantly available in PDB. Most access structures, however, are sensible to be created just by need because they involve a specification of construction criteria. A call graph, for instance, is often needed for one specific subroutine only, and a slice is usually constructed with respect to a specific variable at some program point.

The main internal components of the program data base, the parse tree, the symbol table, and the (current) access structure(s), form an interconnected whole where the different characteristics of the program are stored in the most convenient component. The complete set of properties for a program element can thus be found by collecting all the information fragments from the integrated PDB components. As an external component, the actual program (files) are linked with the internal components of PDB.

The connection between the PDB components is sketched in Figure 2. The parse tree is attributed e.g. with links from its parts to the corresponding concrete program fragments (by indicating the enclosing textual positions in the program file) and with links from the symbolic parts to their symbol table entries. The symbol table contains information needed for analyzing the symbolic entities of the program during the construction of access structures. This includes normal compiler oriented information, such as the name of the entity and its type (if any). For effectively creating an access structure of successive program parts for a given symbol, each symbol table entry is also associated with a linked list of applied nonterminal instances (i.e., parts) in the parse tree. As an example, Figure 2 depicts with thick arrows how a flow-dependence access structure from the declaration of variable 'error_cnt' to all its applied occurrences in the program can be directly obtained from the program data base.

To be effective, the program data base provides an optimized implementation for some central operations. A complete (attributed) parse tree is usually too large for a practical internal representation of the program, as verified by practical experiences with multi-pass compilers and compiler writing systems. Therefore, the parse tree is *abstract* in the sense that it contains only the central nonterminal nodes (that is, the parts) and no terminal nodes. Since the default order of nodes within an edge of a hypergraph is usually their textual

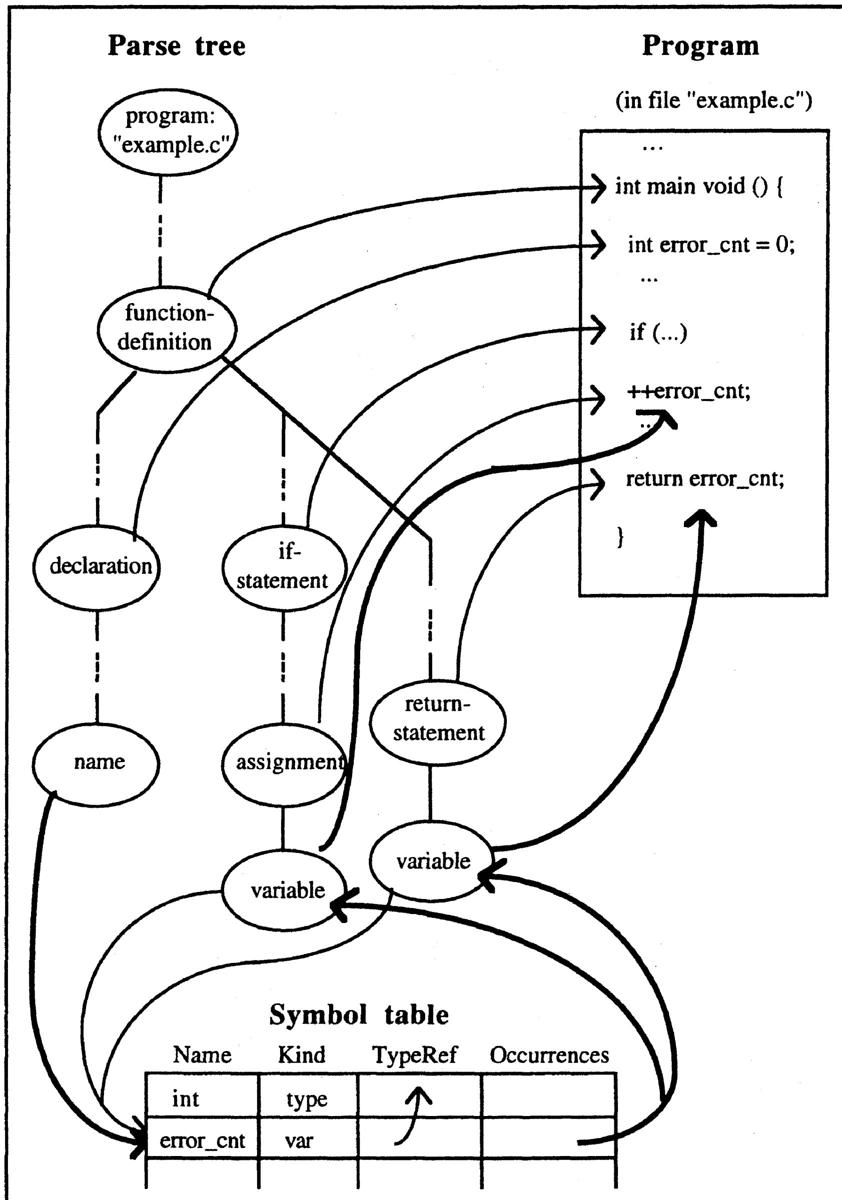


Figure 2. Program data base.

order of appearance in the program, the nodes of a parse tree are associated with an ordinal number with respect to a total preorder. This provides for fast linear analysis of control and data flow (both forward and backward). Some access structures are founded on the structural properties of the program; e.g. one often needs to know whether an indicated 'assignment' part is contained by an 'if-statement' part. The program data base implements a fast mechanism to deduce such structural properties for parts of the central text types. Similarly, a fast mechanism is provided to locate all the parts of a given text type in the parse tree. Finally, in order to resolve certain user-given access structure specifications that depend on some particular program point, the system also maintains links from program files to the parse tree. This facility is needed e.g. when computing a slice with respect to a given variable occurrence in the program. Consequently, HyperSoft connects the textual variable occurrences to their corresponding parts in the parse tree.

The access structures within the program data base run over nodes of the parse tree. While only one access structure at a time is actively on the screen of the user interface (see Section 5), all the possible access structures are potentially available. Hence, a node may be contained in several access structures, depending on the text type of the node. To support such a multi-directional character of nodes, they can be seen as possessing a set of text type dependent links. Some examples of links for different types of nodes are given in Table 1.

Link type	Source node	Target node
NextSimilarStat, e.g. NextOutputStat	output statement	next (or previous) output statement in preorder
IsDefinedAs	variable, constant, function occurrence	corresponding definition, declaration or prototype
IsUsedIn	variable, constant, function definition or declaration	variable, constant application, function call
IsImplementedAs	function call	function code/implemention
AffectsFWS	statement in a forward slice	statement in the same forward slice that is affected by the source node
IsAffectedByBWS	statement in a backward slice	statement in the same backward slice that affects the source node

Table 1. Examples of link and node types.

5. The generic user interface

User interface issues are important in all hypertext systems, and especially in tools intended for reading large programs. The well-known problems in hypertext systems are disorientation and cognitive overhead [Con87]. Disorientation means the tendency to lose one's sense of location in the hyperspace. Cognitive overhead refers to the additional effort and concentration necessary to retain several tasks or trails at one time. Browsing of source code and navigation through it should be made as easy as possible so that the user can concentrate on the maintenance task instead of technical details.

A method extensively used for solving the disorientation problem is to impose a hierarchic structure on the hypertext, see e.g. [AMY88] and [RBS94]. This is the approach in the HyperSoft method as well: a hypertextual access structure is always defined over a hierarchic structure and the hierarchic structure is always a basis for a potential new access structure.

The dynamic creation of the current access structure, based on the current information request by the maintainer, is our way to reduce both the disorientation and the cognitive overhead problems. In many systems, cognitive overhead occurs in the process of reading hypertext, which tends to present the reader with a large number of choices about which links to follow [Con87]. In our approach, a *generic user interface* supports different access structures, and appropriate browsing strategies [SSR86] are designed for the specific access structure categories. The number of possible available links is minimized by showing in a window only those which support the specific information needs, one immediate solution being to assign one access structure to one window. Focusing the attention of the maintainer on one specific access structure at a time also helps in orientation.

A graphical user interface with multiple windows will be used. The main menu of the system will provide access to file loading routines, a possibility to open a standard editor window for changing the program text, standard search functions, and a menu of access structures that are available for generation at the moment. Program text is represented within the windows. Figure 3 illustrates possible windows associated with the access structures. The figure shows a situation where a user has opened windows to view program text based on access structures of general architectural composition (a), calling dependences (b), flow-dependences (c), and forward slicing (d). In maintenance, the access structure in (a) illustrates the general module architecture of the program and thus helps understanding it. The call graph in (b) shows how the emphasized function call (indirectly) activates other functions, thus supporting both program comprehension and tracing. The structure in (c) specifies which program parts are affected by a modification on the emphasized type definition, while (d) approximates the propagation of the value at the emphasized computation through the rest of the program.

Program parts providing non-linear browsing capabilities, corresponding to hypertext *button areas*, are represented in Figure 3 as ovals. Note that for illustrative purposes, the windows (a)-(d) in Figure 3 contain a chain of access nodes, while actually only one of them (with some surroundings for contextual orientation) is visible at a time. Note also that a node may be contained in several access structures, as is the case in (c) and (d). The access structures (b), (c), and (d) involve a generation criterion which is indicated by shadowing the corresponding node. The user navigates through the program by first indicating a part that serves as the starting point for the specific kind of browsing. Because browsing is often possible to two directions there need to be an easy way to specify the order of browsing. Direct transfer to the first and last node of the access structure is also provided.

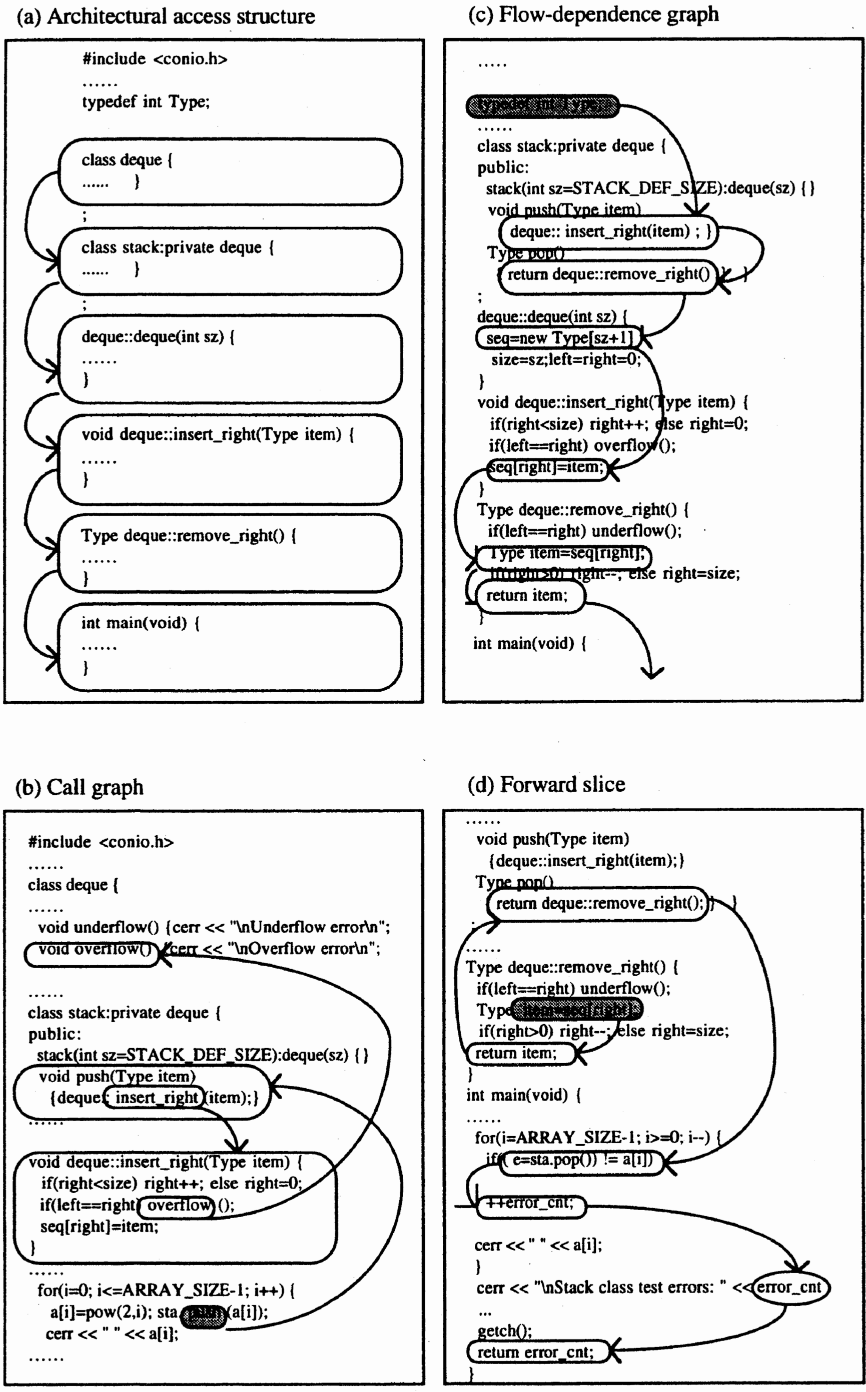


Figure 3. Access structure windows.

Clicking on the button areas will (as a default) cause a new window to be opened (for the reasons of representation this is not depicted in Figure 3). The source text related to this button, and to the corresponding access structure is loaded onto the window. This process can be continued, or backtracked by the user by closing the windows. Opening of new windows helps to keep track of the user's orientation, but also the current window can be used to represent new text if the user so decides. There should be a mechanism to store the current navigation positions within the program files and the access structures for recalling the same navigation procedure later. Requests for generation of access structures are made through the main menu. If a program part needs to be specified by means of a generation criterion, the user is advised to select the corresponding program part from the program text on the active window.

The user should have information about the relative location of each text fragment on the screen. Hence, information about the context which a program part belongs to needs to be expressed somehow. In case of large nodes, information should be abstracted (elision) such that unnecessary details are hidden. Examples of the possible ways to economically represent program text on the screen are given in e.g. [San89, OmC90, Bro91, Ray92].

6. Related work

Our work is related with *software hypertext systems* and more generally with *program comprehension support tools*. In most software hypertext systems [BiR87, GaS90, CyR92] the emphasis is on providing support for linking separate documents, source files, requirements etc. together. In contrast, our approach concentrates on providing support for program viewing and browsing on the intra-modular source code level.

One of our central aims is to provide hypertextual fragmentation and access structures automatically in order to eliminate the need for manual linking. This kind of approach is employed e.g. in DynamicDesign [BiR87, Big88] in which source code is represented in a hypertext format, based on a call tree. Another related system is CodeNavigator [Bro91] in which flow relationships are represented as directed graphs. As a more general solution, our method makes it possible to produce multiple links and fragmentations for various tasks by automatic generation of access structures.

Various access structures are also supported in the Mjølner BETA System [San89]. The base language is BETA, an object-oriented programming language. In this system the supported links include also so-called program semantical links. Structured objects are represented as abstract syntax trees and semantical links are generated automatically. Supported link types of this category include definition-use and superclass relationships.

The work in the HyperPro project [ØsN93] has concentrated on data modelling and storage aspects of hypertext. Based on these results a prototype environment called HyperPro has been implemented. A set of node and link instances form a network which is called the program network or the hyperstructure. HyperPro is a generic (language-independent) environment, which is accommodated to a specific programming language. A key problem is to decide the criteria for fragmenting the source code. A convenient level of granularity is suggested to be at the procedure (method) level.

A similar approach as ours is employed in PUNS (Program Understanding Support Environment) [Cle89] which produces multiple views of source code collections to support the program comprehension process. Functionally this is very close to what we aim at. In addition, our approach includes a model to represent different access structures consistently as hypergraphs. We also concentrate on providing transient access structures created by the user in order to save memory and processing time during hypertext formation, as well as

to provide customized support for varying information needs of the maintainer. This emphasis on *dynamic* creation of a hypertextual program representation reflects the central role of the source program in the HyperSoft method. Note that this is in contrast to programming environments that emphasize the linking of the *static* documentation with the source code, in which case it is a natural choice to create a persistent hyperstructure already during program development.

A recent work on hypertext tools for software maintenance is Whorf [BGS94] which provides explicit hypertextual support for visualizing and understanding delocalized plans while using an as-needed strategy. Strategy is supported through multiple, concurrent views of the software with instant, easy access to additional views.

Because most access structures we have proposed as a part of the HyperSoft method are based on well-known program dependences, they have been implemented in various ways in many program comprehension support tools, of which the nearest to our work are the following.

* Dependence Analysis Tool Set [Oma90b] provides a basis for determining program dependences and to understand the complex interrelationhips in large C programs.
* The Smart System environment for the C language [Oma90a] generates call graphs and data dependence trees to document the program structure.
* Surgeon's Assistant [Oma90b] slices up C programs, extracts pertinent information, and displays data links and related characteristics such that changes and influence of selected structures can be tracked.
* EDSA (Expert Dataflow and Static Analysis Tool) [Oma90b] statically analyzes and slices Ada programs. It can be used to determine the effects of changes and possible side-effects.

7. Conclusions

The hardest problems in software maintenance are

* to understand the program and
* to localize the program parts that should be modified.

These problems are most serious when maintaining large legacy systems that have evolved in various versions, often without any proper documentation. When the size and complexity of the systems grow and original programmers are no longer available, an increased portion of human resources is bound to the maintenance. Because of the capital invested on these systems and because of knowledge they contain it is hard to justify throwing them away although their maintenance may be very hard and costly. This emphasizes the need for comprehension support tools. An advanced solution to manage the complexity of software maintenance is to provide support for automatically extracting the relevant parts from the program and for flexibly navigating through such a focused view.

Since the HyperSoft method presented in this paper is founded on the combination of hypertext and static program analysis, its implementation as the HyperSoft system will be based on these two techniques. The maintainer will start program reading by specifying which category of access structures shall be provided to support the current maintenance activity. The system will analyze the software and produce the requested dependence graphs. The graphs will be elaborated into a hypertextual access structure which is provided to the user through a generic user interface. Finally, the user interface makes it possible to flexibly navigate through the program and simultaneously introduce the maintaining actions

into the program text with an editor. In principle, it would be possible to integrate incremental updating of the program data base with program navigation and editing, based on the techniques developed in the context of language-based editors (see e.g. [ReT89]). This advancement would, however, make the implementation more complex and inefficient and is therefore left for further study.

The HyperSoft system will be targeted to industrial legacy systems under extensive maintenance. We are currently collecting legacy systems that will be supported by the concrete HyperSoft system, and starting the implementation of the HyperSoft system, based on the design presented in this paper.

Acknowledgements. The comments of the referees have been helpful for improving the presentation.

References

AMY88 Akscyn, R.M., McCracken, D.L. & Yoder, E.A., "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations", Commun. ACM 31, 7, (July 1988), 820-835.

ASU86 Aho, A.V., Sethi, R., & Ullman, J.D., "Compilers - Principles, Techniques, and Tools", Addison-Wesley, 1986.

BGS94 Brade, K., Gudzial, M., Steckel, M. & Soloway, E., "Whorf: A Hypertext Tool for Software Maintenance", International Journal of Software Engineering and Knowledge Engineering 4, 1, (Mar. 1994), 1-16.

Big88 Bigelow, J., "Hypertext and CASE", IEEE Software, Mar. 1988, 23-27.

BiR87 Bigelow, J. & Riley, V., "Manipulating Source Code in DynamicDesign", In: Hypertext'87 Papers, 397-408.

Bro91 Brown, P., "Integrated Hypertext and Program Understanding Tools", IBM Syst. J. 30, 3, 363-392.

Cle89 Cleveland L., "A Program Understanding Support Environment", IBM Syst. J. 28, 2, 324-344.

Con87 Conklin, J., "Hypertext: An Introduction and Survey", Computer (IEEE) 20, 9, (1987) 17-41.

CyR92 Cybulski, J.L. & Reed, K., "A Hypertext-based Software-engineering Environment", IEEE Software, Mar. 1992, 62-68.

GaS90 Garg, P.K. & Scacchi, W., "A Hypertext System to Manage Software Lifecycle Documents", IEEE Software, May 1990, 90-98.

HaS90 Harrold, M., & Soffa, M., "Computation of Interprocedural Definition and Use Dependencies", In: Proceedings of the IEEE Comput. Soc. 1990 Int. Conf. on Comput. Languages, New Orleans, LA, Mar. 1990, 297-306.

HMR93 Harrold, M., Malloy, B. & Rothermel, G., "Efficient Construction of Program Dependence Graphs", In: ISSTA'93, Proc. of the 1993 Int. Symp. on Software Testing and Analysis, 160-170.

HoR90 Horwitz, S., Reps, T., & Binkley, D., "Interprocedural Slicing Using Dependence Graphs", ACM Transactions on Programming Languages and Systems 12, 1, (Jan. 1990), 26-60.

HoR92 Horwitz, S. & Reps, T., "The Use of Program Dependence Graphs in Software Engineering", In: Proc. 14th Int. Conf. on Software Engineering, Melbourne, Australia, IEEE Computer Society Press.

Kam93 Kamkar, M., "Interprocedural Dynamic Slicing with Applications to Debugging and Testing", PhD thesis, Linköping Studies in Science and Technology Dissertations No. 297, Department of Computer and Information Science, Linköping Univ.

KPS94 Koskinen, J., Paakki, J. & Salminen, A., "Program Text as Hypertext: Using Program Dependences for Transient Linking", SEKE'94, Software Engineering and Knowledge Engineering Conference, Jurmala, Latvia, June 21-23, 1994, to appear.

LeS86 Letovsky, S. & Soloway, E., "Delocalized Plans and Program Comprehension", IEEE Software, May 1986, 41-49.

Oma90a Oman, P., "CASE Analysis and Design Tools", IEEE Software, May 1990, 37-43.

Oma90b Oman, P., "Maintenance Tools", IEEE Software, May 1990, 59-65.

OmC90 Oman, P. & Cook, C.R., "The Book Paradigm for Improved Software Maintenance", IEEE Software, Jan. 1990, 39-45.

Pre92 Pressman, R.S., "Software Engineering - A Practitioner's Approach" (3rd ed.), McGraw-Hill, Singapore, 1992.

PSK94 Paakki, J., Salminen, A. & Koskinen, J., "Automated Hypertext Support for Software Maintenance", submitted for publication.

Ray92 Raymond, D.R. "Flexible Text Display with Lector", Computer (IEEE) 25, 8, (Aug. 1992), 49-60.

RBS94 Rivlin, E., Botafogo, R. & Shneiderman, B., "Navigating in Hyperspace: Designing a Structure-based Toolbox", Commun. ACM, 37, 2, 87-96.

ReT89 Reps, T. & Teitelbaum, T., "The Synthesizer Generator", Springer-Verlag, 1989.

Ryd79 Ryder, B., "Constructing the Call Graph of a Program", IEEE Transactions on Software Engineering, SE-5, 3, (May 1979), 216-225.

San89 Sandvad, E., "Hypertext in an Object-Oriented Programming Environment", In: WOODMAN'89: Workshop on Object-Oriented Document Manipulation, Rennes, France, 1989.

SaW92 Salminen, A. & Watters, C., "A Two-level Structure for Textual Databases to Support Hypertext Access", J. American Society for Information Science 43, 6 (July 1992), 432-447.

Shn89 Shneiderman, B., "Reflections on Authoring, Editing, and Managing Hypertext", In: The Society of Text, Barret, E. (ed.), MIT Press, Cambridge, MA, 115-131.

SSR86 Shneiderman, B., Shafer, P., Roland, S. & Weldon, L., "Display Strategies for Program Browsing - Concepts and Experiment", IEEE Software, May 1986, 7-15.

Wei82 Weiser, M., "Programmers Use Slices When Debugging", Commun. ACM 25, 7, (July 1982), 446-452.

ØsN93 Østerbye, K. & Nørmark, K., "The Vision and the Work in the HyperPro Project", Technical report, Institute for Electronic Systems, Department of Mathematics and Computer Science, 1993.

III

CREATING TRANSIENT HYPERTEXTUAL ACCESS STRUCTURES FOR C PROGRAMS

Koskinen, J. 1996. In *Proceedings of the 7th Israeli Conference on Computer Systems and Software Engineering* (*ICCSSE'96*). Los Alamitos, CA: IEEE Computer Soc., 56-65.

(C) 1996 IEEE. Reproduced with permission.

Creating Transient Hypertextual Access Structures for C Programs

Jussi Koskinen
Department of Computer Science and Information Systems
University of Jyväskylä
P.O. Box 35, SF-40351 Jyväskylä, Finland
Internet: koskinen@cs.jyu.fi

Abstract

The paper describes how hypertextual access structures can be formed to support the maintenance of C programs. The access structures are formed automatically based on the HyperSoft model and method developed earlier. The automatic creation is based on syntactical fragmentation of program text and on relationships between these fragments. The access structures are transient, meaning that instead of storing them permanently they are created on user request. HyperSoft enables flexible navigation between the program parts which are relevant to a certain maintenance situation. The HyperSoft system currently supports C language and five access structures: occurrence lists for variables and functions, forward and backward calling dependence structures, intraprocedural backward slices, and interprocedural forward slices. Access structures are represented to the user as a set of highlighted nodes and graphical links on top of the original program text. The HyperSoft system has been developed in co-operation with the four largest software houses in Finland.

1. Introduction

HyperSoft is an ongoing project during which a hypertext model and method [13], [19] and a system have been planned [21] and implemented. *HyperSoft* method combines the hypertext modeling and program analysis approaches to produce automatically various access structures which can be used to support tasks like program comprehension, debugging, and impact analysis. While using *HyperSoft,* the maintainer initiates the access structure generation by pointing some relevant program part and the desired operation. *HyperSoft* then generates the structure and shows it to the user as hypertext. The *HyperSoft* system consists of three relatively independent main components suggested by the model. The static program analyzer creates parse trees (abstract syntax trees) [2] and other relevant static structures which are stored into the static program database. The access structure generator then uses that information to produce various access structures, which are stored into the dynamic program database. Finally, these structures are represented to the user as hypertext through the generic user interface.

The four largest software houses in Finland: KT-Tietokeskus, Nokia Research Center, Tietotehdas and VTKK-Kuntajärjestelmät, have co-operated with us during the project. The steering group of the project have partaken to the guidance of the project by reviewing the suggested *HyperSoft* functionalities. Since most of the maintenance problems of the partner enterprises are related to legacy systems written in C [11], C was chosen as the first language to be supported in the system. The system runs on IBM-PC & compatibles and its user interface works under Microsoft Windows. Five access structures which were considered most relevant to the needs of the partner enterprises have been implemented.

HyperSoft is targeted to software maintenance support. The problems related to software maintenance are prominent and well-known [18]. The software maintainer has to somehow grasp the understanding over the existing software so that required maintenance tasks can be performed. The identification of the relevant program parts and their interdependences is a generic task required for example in impact analysis, debugging, and testing. Moreover, in a certain maintenance situation, the maintainer has specific information requests, which should be satisfied. So there should be a versatile set of access structures of which the maintainer could select the one that is most appropriate to the situation. Because the nature of software maintenance is interactive, there should be effective and flexible mechanisms to specify the foci of interest and to view the relevant program parts.

This paper concentrates on the back-end issues of forming hypertextual access structures. First, the main

concepts of the *HyperSoft* model are provided in Section 2 and the general architecture of the implemented *HyperSoft* system is described in Section 3. The needed static structures to form the hypertextual access structures are described in Section 4 and the implemented access structures in Section 5. Since the slices are the most complex of the implemented access structures, they are described in a more detailed level. *HyperSoft*'s slicing capabilities are compared with other existing slicing tools.

2. Creating access structures based on the HyperSoft model

Program text is modelled in *HyperSoft* as a two-level structure where the syntactic structure is separated from the access structure [22]. This basic model is adapted to the needs of software engineering context. Program text with hypertext access capabilities is a triple *(G, X, A)* where *G* is a context-free grammar, *X* a syntactic structure, and *A* an access structure. The syntactic structure is a parse tree for the program with respect to the grammar *G*. In *HyperSoft* parse trees represent the hierarchic structure of the program text, so that non-terminals of the parse tree are called *text types*. The corresponding text entities are called *parts* of that type. For browsing purposes hypertext is modelled as a directed graph *(Z, E)* where *Z* is a set of *nodes*, and *E* a set of node pairs, called *links*. The access structure is a transient (hyper)graph whose nodes are parts of the syntactic structure. These access structures may be formed automatically according the *HyperSoft* method by using various program analysis techniques. The *HyperSoft* model and method are described in detail in [13].

2.1. Access structure nodes

Parts of the access structure have their value and type. In case of C programs the syntactical text type may be, for example *identifier*, *function call*, *function definition* or *expression* (see [11], Section A13). The value is the corresponding string in the program text. The elementary nodes may be collected into a set of nodes, thus forming a compound structure. These compound structures are viewed in HyperSoft as access structures. The set of relevant text types is dependent on the access structure and the software maintenance task that it should support.

2.2. Access structure dependences

There is a multitude of possible relations (dependences) between the elementary and compound fragments of programs. The typical relations include the *data flow*, *control flow*, and *calling dependences*. These relations and their variants are also supported in many program comprehension support tools, like CIA [4] and EDATS [25]. In principle, a system based on the *HyperSoft* model can support any dependences which may be specified between the program fragments. Note that *HyperSoft* differs from the so-called *software hypertext systems*, like [5] and [6] in a sense that the hypertextual structures are formed automatically. So, no manual linking is needed in *HyperSoft*.

2.3. Combining sets of related nodes as access structures

Access structures may be simple references, sets, chains, trees, or graphs. Some of the possible access structures have been categorized in Table 1. Structures that are currently implemented in *HyperSoft* are denoted with an asterisk. The *singleton* kind structures simply give the user a single (reference) link through which he or she can navigate within the program text, *sets* contain associated but not ordered elements, *lists* link a set of related elements together, *trees* provide a basis for versatile browsing and finally *graphs* often require associated map-views to manage with the structure. A single access structure may be composed of nodes which are connected based on various program dependences. For example the calling dependence structures of *HyperSoft* consist of two types of nodes (function call and implementation) and dependences (data flow and structural) binding them. Slices in turn consist of many types of nodes bound together based on two main program dependences (data flow and control flow dependence).

Form	Examples
Singleton:	*Declaration, Context-Function, Structure Beg./End Reference
Sets/Lists:	*Occurrence-, Instance-, Definition-, Usage-, Global-Variable-, Modification-Lists
Trees:	*Forward *Backward and Intermodular Calling Dependence Structures
Graphs	*Backward and *Forward Slices

Table 1. Examples of different access structure types.

3. HyperSoft architecture

The general architecture of the *HyperSoft system* is represented in Figure 1. The *analyzer* component supports the analysis of error-free C programs and produces the static program database for the source programs. The analyzer is built using the AnaGram metacompiler [17] and C preprocessor package delivered with AnaGram. The *generator* creates the transient hypertextual access structures. The *program database* is a repository storing the information passed between the analyzer and the generator. The generator and *interface* components are separated in a sense that the interface is language independent. Multi-windowed browsing of source code is supported. *HyperSoft* is also integrated to the Programmer's File Editor (PFE). *HyperSoft* currently constitutes about 30,000 lines of code. The implementation of the back-end part of the *HyperSoft* system is more widely reported in [12] and the development of the interface component and its special features in [16].

The general order of forming and representing the necessary data structures within the *HyperSoft* is shown in Figure 2. Most notably the parse tree and symbol table generation functionalities are separated. This sort of a separation makes radical parse tree pruning easier, because the pruning can be based on the contextual information stored into the parse tree.

4. Creation of the static structures

The structure of the analyzer component and its relationships to other components is depicted in Figure 3. A parser using the Kernighan & Ritchie [11] (Section A13) style C grammar is extended to produce the needed static information. The program database consists of the static and dynamic parts. The static database consists of the global symbol table, local symbol tables, and abstracted parse trees. The choice of using parse trees to hold most of the necessary static information is based on the *HyperSoft* model, where the syntactic structure of the program is represented as a parse tree. Access structures which are created during the maintenance session are stored into the dynamic database, which is separated from the analyzer component. The program database is implemented as a collection of DOS files.

AnaGram metacompiler and the C preprocessor package: AnaGram metacompiler contains an LALR(1) parser generator which creates a parser based on a grammar. Like most parser generators it creates a parser file, written in C or C++ based on the user-defined syntax files. This parser file is then compiled with some C/C++ compiler in order to form the executable parser. In *HyperSoft,* the generated parser is used to form the program database. The preprocessor and C parser constitute together about 580 productions.

Forming the static program database: In *HyperSoft,* the access structures are transient, meaning that they are generated during the maintenance session. This helps to reduce the size of the needed static database. Information which have to be stored statically in order to form the hypertextual access structures include: textual positions and types of the relevant program parts (stored into parse trees), category of the symbols (stored into symbol tables) and occurrences of those symbols (stored into occurrence

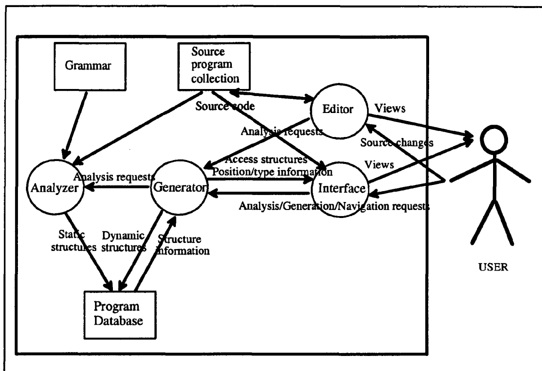


Figure 1. The general architecture of *HyperSoft*.

1) Preprocessing, parsing, parse tree formation and validation.
2) Parse tree abstraction and compression.
3) Creation of symbol tables, based on the forward preorder traversal of the already formed parse trees.
4) Access structure generation, based on the static program database produced in the phases 1 and 3.
5) Access structure representation to the user via a generic, language independent, graphical user interface, including text and navigation views.

Figure 2. The process of forming hypertext structures in *HyperSoft*.

lists). The definitions for the elements of these structures are given in Figure 4.

The static information is stored into the database during a batch process. There exists static linkages 1) from each symbol definition node of the parse tree to the adequate symbol table row, 2) from each occurrence list element to the corresponding parse tree node, and 3) a position reference from each parse tree node to the original program text. The local symbol table contains the local variables and scope information, whereas the global table gathers all the intermodular information, that is, global variables and functions which are not preceded by the *static* keyword. Table 2 summarizes the elementary syntactical structures that are needed in forming the currently implemented access structures.

5. Creation of the dynamic structures

The dynamic access structures of the *HyperSoft* are formed by the access structure generator (see Figure 5). The generator uses the static information stored into the database and passes information about the formed access structures to the interface component. The MVC model [14] is used to separate the access structure manipulation and representation to the user. Currently the generator supports five access structures: 1) occurrence lists, 2) forward calling and 3) backward calling dependence structures, 4) intraprocedural backward slices, and 5) interprocedural forward slices. Occurrence lists and calling structures are useful in many different maintenance situations. Backward slices are useful in debugging and forward slices, for example, in attempts to estimate the possible side-effects of a proposed change. All the access structures consist of nodes of a similar type.

5.1. Occurrence lists and call graphs

An occurrence list (see Figure 6, left pane) is an example of a Set/List kind structure containing the occurrences of a specified symbol. Presently occurrence list is available for variables, functions and macro-usages. Symbols having different scopes are separated in the structure. The links can be used to move to next occurrence. All access

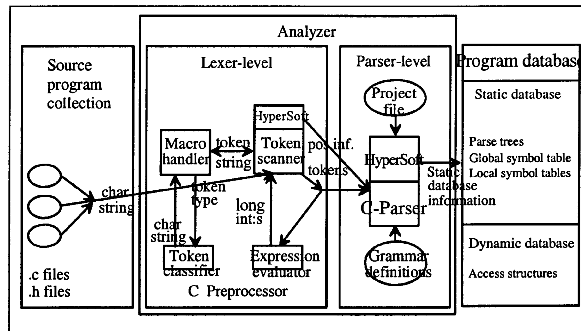
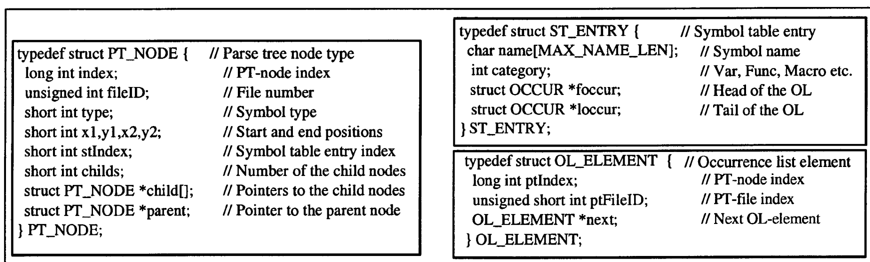


Figure 3. Components producing the static information in *HyperSoft*.



```
typedef struct PT_NODE {        // Parse tree node type
  long int index;               // PT-node index
  unsigned int fileID;          // File number
  short int type;               // Symbol type
  short int x1,y1,x2,y2;        // Start and end positions
  short int stIndex;            // Symbol table entry index
  short int childs;             // Number of the child nodes
  struct PT_NODE *child[];      // Pointers to the child nodes
  struct PT_NODE *parent;       // Pointer to the parent node
} PT_NODE;
```

```
typedef struct ST_ENTRY {          // Symbol table entry
  char name[MAX_NAME_LEN];         // Symbol name
  int category;                    // Var, Func, Macro etc.
  struct OCCUR *foccur;            // Head of the OL
  struct OCCUR *loccur;            // Tail of the OL
} ST_ENTRY;
```

```
typedef struct OL_ELEMENT {        // Occurrence list element
  long int ptIndex;                // PT-node index
  unsigned short int ptFileID;     // PT-file index
  OL_ELEMENT *next;                // Next OL-element
} OL_ELEMENT;
```

Figure 4. Central data structures of the static program database.

Access structure	Needed static structures (types of the parse tree nodes)
Occurrence list	identifier
Calling structures	+ function-call, function-definition
Slicing structures	+ statement, declaration, expression, type-definition, parameter-list, pointer-id, statement-category

Table 2. The needed minimal syntactical information to form some of the access structures.

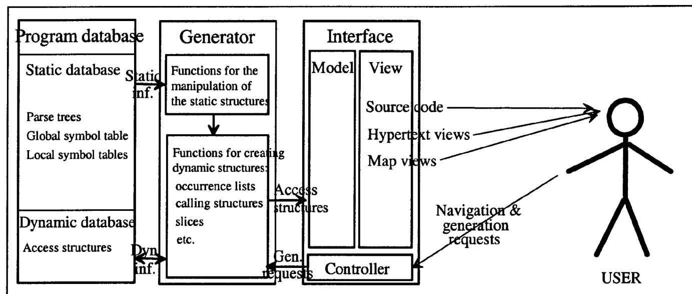


Figure 5. Components using the static and dynamic information in *HyperSoft*.

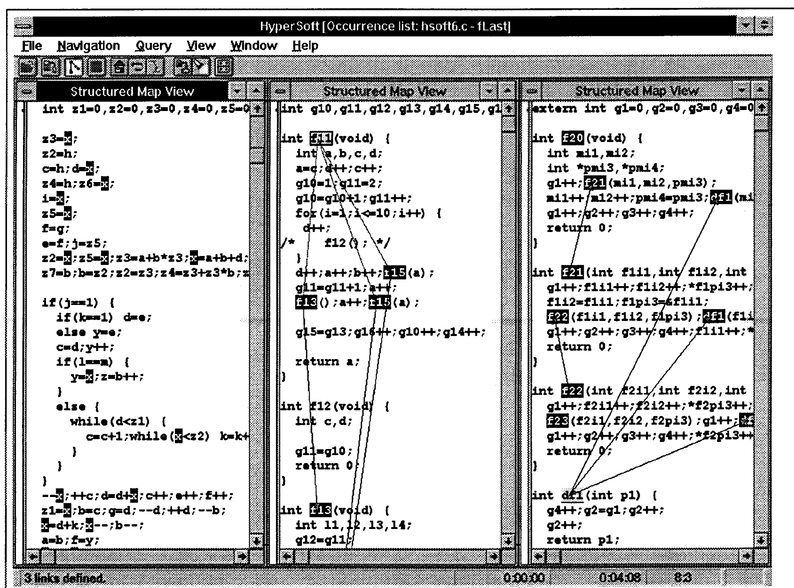


Figure 6. Occurrence list and calling structures.

structure nodes are represented as highlighted screen elements bound together with links enabling a fast transition between them. If there exists only one link related to a certain node, *HyperSoft* performs the transition when the screen element is activated. This is the case for occurrence lists. If there exists multiple target nodes, a pop-up menu is shown from which the selection is made. Note that *HyperSoft* supports different levels of showing the links, either 1) only links originating from the active node are shown, or 2) only the interprocedural links are shown, or 3) all intra-modular links are shown. In most of the oncoming figures all defined links are shown.

The calling dependence structures are useful for example in attempts to estimate the effects of changing a function. In *HyperSoft,* all access structures concentrate on the area specified by the user. Both calling structures contain nodes corresponding to function calls and function implementations. In a forward calling structure (Figure 6, central pane) there exists links from the relevant function calls to the corresponding implementations. For each of these implementations links are then formed to the function calls within their body based on the structural information. The backward calling structure (Figure 6, right pane) shows where a certain function may have been called from. Relevant function implementation nodes corresponding to the function names of the function implementation are linked to the calls of that function based on a backward calling dependence. These nodes in turn are linked to their context function implementations based on a structural dependence.

5.2. Slicing structures

Slicing [26] means the extraction of relevant statements from the source programs into the slice. The focus of interest is specified by the *slicing criterion*, which typically is a variable occurrence within the program text. Slicing then proceeds backwards or forwards, and possibly spreads into other functions (*interprocedural slicing*). Backward slicing is typically used as an aid to debugging, whereas the main application area of the forward slicing is impact analysis. Slicing systems can be differentiated from each other by the main characteristics, which are summarized in Table 3, adapted from [7]. Note, however, that unlike the other systems, *HyperSoft* is not a pure slicing system. Instead, slices are simply one category of possible access structures within *HyperSoft*. Boxes whose content is typed in boldface indicate similarities between *HyperSoft* and other systems supporting slicing.

HyperSoft currently supports intraprocedural backward slicing and interprocedural forward slicing. Slicing is variable based and the linkages are formed mainly on "statement" level. The slice is represented embedded within its original context by using both textual and graphical views. *HyperSoft* currently does not support the analysis of pointer variables (except in function calls) [8] or the analysis of unstructured flow of control (*goto* statements) [3].

Program parts: Program fragments which may be included into the slice are of types (see [11]): *statement* ; elementary statements, *init-declarator*; initializations of the assignment statement set, and *expression*; for example predicate conditions. In some special cases also nodes of the type *identifier* are included into the slice.

Dependences: The relevance of each statement is determined by its data- and control flow dependences to the slicing criterion. Let x and y be variables in certain program points within the source code, then if the changes of the value of y have effect on the value of x (through assignment statements), then x is *data- dependent* on y, and if y occurs in a predicate controlling whether some or any

Dimension/Tool	Kamkar's tool	Spyder	Schatz' tool	FOCUS	WPIS	HyperSoft
Direction	Backward	B	B	B	**Forward/B**	**Forward/Backw.**
Type of analysis	Dynamic	**D/Static**	**S**	**S**	**S**	**Static**
Static structures	PDG	PDG	SDG	CFG	PDG	**Parse trees**
Slicing criterion	Function-level	**Variable**	**V**	**V**	**V**	**Variable**
Scope	**Interproced.**	**Inter-**	Intraprocedural	Intra-	**Inter-**	**Inter-/Intra-**
Language	Pascal subset	**C**	Fortran subset	**C subset**	Pascal-like	**C**
ViewsText/Graph.	T/G, Extracted	**T, Embedded**	T, Extracted	**T, Embedded**	T, Extracted	**T/G, Embedded**
Linkage	**Block-level**	-	-	-	-	**Statement level**
Focus	Deb./Testing	Debugging	Not stated	Debugging	Program integr.	**Impact analysis**
Reference	[10]	[1]	Based on [23]	[15]	Based on [20]	[12]

Table 3. Slicing dimensions and implementations

statement which has effect on x will be executed or not, then x is *control-dependent* on y.

Formation of the slice: Construction of the slice requires the traversal of the structured program text started from the slicing criterion point, by following the programs's control flow and the determination of whether the traversed statements are parts of the slice. The necessary information can be stored in *parse trees* [2], *abstract syntax trees*, *production trees* [24] or in different kinds of *program dependence graphs* (PDGs) [9], such as *control flow graphs* (CFG) or *system dependence graphs* (SDG). In *HyperSoft* the slicing is based on *static iterative solving of data flow equations* via the backward or forward preorder traversals and analyses of the parse subtrees corresponding to the relevant functions. In case of intermodular function calls, the corresponding new active parse tree and local symbol table are recreated from disk. There are three main reasons to the selection of the applied parse tree based approach: 1) it is consistent with the *HyperSoft* model, 2) slicing structures are only one category of possible *HyperSoft* access structures, whereas the PDG:s are tailored for slicing, 3) the updating of the program database after modification of the sources is more straightforward than in case of using the PDG:s.

Representation of the slices: Slices are traditionally represented to the user as a set of program statements extracted from the original program text. This is not the case in *HyperSoft*, because we feel that the context information about the original program text is important in order to understand the slice contents. Therefore, in *HyperSoft* the slices are represented as a set of highlighted screen elements inside the program text. The idea of representing the internals of slices to the user as graphical links set on top of the program text is not applied in other slicing tools. In most slicing tools the slice is just represented as a set of statements.

Linkages: The use of intra-slice linkages necessitates the restriction of the number of links to those which are most useful to the maintainer. Therefore we have decided to apply the statement level linkage instead of variable based. The control dependence linkages are excluded from the slicing structures, since these links are *implicit* in a sense that the variable names inside the predicates reveal the reason why a certain (compound) statement is included into the slice. Informally, the linkages are formed so that they reveal the reason why a certain statement is included into the slice. Formal definition, and the rules for the formation of a slice can be found from [12].

Intraprocedural backward slices: A backward slice is a set of statements which may have influence to the slicing criterion. Figure 7 represents a part of an intraprocedural backward slice as seen through the *HyperSoft* interface. Slicing has been initiated from the occurrence of variable x in statement $x=d+k$. The slice can be used to find out the statements affecting a certain statement or to gain general comprehension about the purposes of the variables.

Interprocedural forward slices: A forward slice consists of a set of statements to which the slicing criterion may have influence on. Interprocedural slicing analysis is rather complex. The well-known "calling context problem" [9] is related to the static upward slicing. Informally, *downward slicing* means the analysis of functions which are called from a certain function, wheras the *upward slicing* means the analysis of functions from which a certain function is called from. During the static analysis, all the upward calling chains of the initial function (from which the slicing has been started) have to be checked out. This is the main cause of the efficiency problems related to the static interprocedural slicing. Figure 8 shows the originating module of an interprocedural forward slice whose formation is based on complete slicing analysis. The slicing has been initiated from the variable occurrence *f2pi3* within the function *f22*. A certain function can have been included to the slice based on both the upward and downward slicing analyses. There are several ways which cause a linkage to be formed, including direct or indirect dependences, downward and upward slicing analysis and the usages of global variables.

6. Summary and conclusion

We represented the architecture of the *HyperSoft* system which enables viewing program text as hypertext. The architecture is suggested by the *HyperSoft* model where the static syntactical structure and dynamic access structures are separated from each other. Ways to produce the necessary static and dynamic information, needed to view C programs as hypertext, were given. Five actually implemented access structures selected by the industrial steering group of the project were described. The access structures enable fast viewing of relevant program parts and transition between the dependent program parts. All hypertextual access structures of *HyperSoft* are represented as set of nodes within the original program text, which makes the investigation of the node surroundings easy. Also the intra-slice dependences are represented both graphically and hypertextually as embedded into the program text.

The *HyperSoft* system has been implemented in order to investigate the practical possibilities of applying the *HyperSoft model* and *method*. The first phase of the *HyperSoft* project has shown that the idea of representing

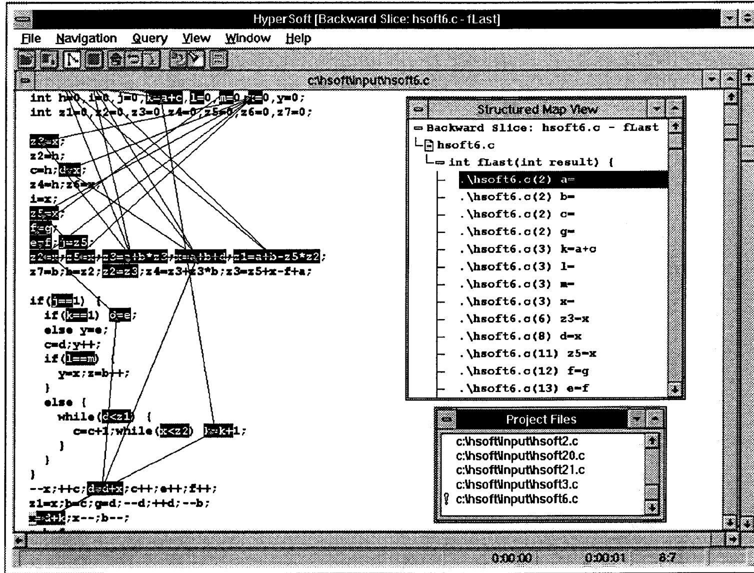


Figure 7. Intraprocedural backward slice.

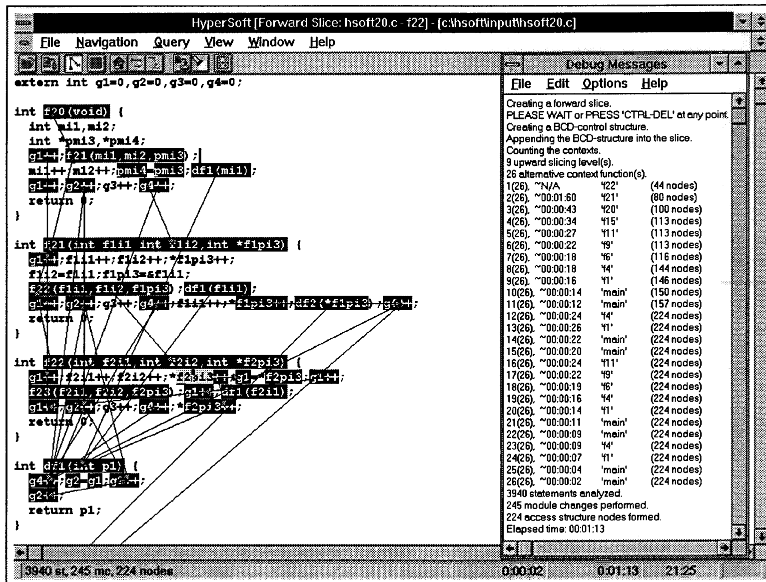


Figure 8. Interprocedural forward slice.

the program text and its internal dependences as hypertext is working in practice. The preliminary experiences with the *HyperSoft* system among the representatives of the partner enterprises and the involved professional maintainers have been positive. The idea of transient hypertext, that is, generating the hypertextual access structure on user request, makes it possible to reduce the amount of needed statically stored information.

The efficient implementation of full static slicing in case of very large programs is difficult. This means that in these situations the slicing has to be performed as a batch process, emphasizing the importance of supporting flexible slicing criteria and partial formation of the slices. The partial formation based on the user-defined number of upward and downward slicing levels is already supported in the latest version of *HyperSoft*. Since parse trees are needed in forming the versatile set of *HyperSoft* access structures, and because they constitute most of the static program database, their abstraction is the most promising way of boosting the efficiency of the system. Different strategies of developing the system performance are described in [12].

The main emphasis during the next phase of the project is targeted to the further refinement of the C language support. Also support for the incremental updates of the static database and an editor integration will be provided. Although the currently implemented access structures are mainly based on data-flow kind dependences, the general idea of representing the dependent program parts as hypertext could also be used to deal with, for example, the differences between different versions of a single program. Possibilities to extend the current access structure set are discussed in [12]. Because the generator and interface components are separated from each other, the introduction of new languages is in principle straightforward. The implementation of an analyzer for embedded SQL database queries has been started in form of a spin-off project.

Acknowledgements: The financial support of TEKES (Technology Development Center of Finland), Jyväskylän Kauppalaisseuran Säätiö, KT-Tietokeskus, Nokia Research Center, TT-Tieto (formerly Tietotehdas) and VTKK-Kuntajärjestelmät is gratefully acknowledged.

References

[1] Agrawal, H. *Towards Automatic Debugging of Computer Programs*. Ph.D. thesis, Purdue Univ., West Lafayette, Indiana, USA, 1991.

[2] Aho, A., Sethi, R. & Ullman, J. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] Choi, J.-D. & Ferrante, J. "Static slicing in the presence of goto statements". *ACM TOPLAS* 16, 4 (July 1994), pp. 1097-1113.

[4] Chen, Y.-F., Nishimoto, M. & Ramamoorthy, C. "The C information abstraction system". *IEEE Transactions on Software Engineering* 16, 3 (March 1990), pp. 325-334.

[5] Cybulski, J. & Reed, K. "A hypertext-based software-engineering environment", *IEEE Software*, Mar. 1992, pp. 62-68.

[6] Garg, P. & Scacchi, W. "A hypertext system to manage software lifecycle documents", *IEEE Software*, May 1990, pp. 90-98.

[7] Hoffner, T. *Evaluation and Comparison of Program Slicing Tools*. Technical report, LiTH-IDA-R-95-01, Linköping University, Linköping, Sweden, 1995.

[8] Horwitz, S., Pfeiffer, P. & Reps, T. "Dependence analysis for pointer variables". *Proc. SIGPLAN'89 Conf. Programming Language Design and Implementation, ACM SIGPLAN Notices* 24, 7, pp. 28-40, 1989.

[9] Horwitz, S., Reps, T. & Binkley, D. "Interprocedural slicing using dependence graphs". *ACM TOPLAS* 12, 1, pp. 26-60, 1990.

[10] Kamkar, M. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. Ph.D. thesis, Linköping Studies in Science and Technology Dissertations, No. 297, Dept of Computer and Information Sc., Linköping Univ., Linköping, Sweden, 1993.

[11] Kernighan, B. & Ritchie, D. *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[12] Koskinen, J. *HyperSoft: Static Program Analyzer, Program Data Base and Access Structure Generator Components*. Computer Science and Information Systems Reports, Working Paper WP-35, Dept of Computer Science and Information Systems, University of Jyväskylä, Jyväskylä, Finland, 60 p., 1996.

[13] Koskinen, J., Paakki, J. & Salminen, A. "Program text as hypertext - using program dependences for transient linking". *SEKE'94, 6th Int. Conf. Software Engineering and Knowledge Engineering*, Jurmala, Latvia, pp. 209-216, 1994.

[14] Krasner, G. & Pope, S. "A cookbook for using the model-view-controller interface paradigm in Smalltalk-80". *Journal of Object-Oriented Programming* 1, 3 (Aug./Sept. 1988).

[15] Lyle, J., *Evaluating Variations on Program Slicing for Debugging*. Ph.D. thesis, Univ. of Maryland, ML, USA, 1984.

[16] Nieminen, M. *HyperSoft järjestelmän käyttöliittymä ja sen kehittäminen (HyperSoft System: The User Interface)*, Master's thesis (in Finnish), Department of Computer Science and Information Systems, University of Jyväskylä, Jyväskylä, Finland, 1996.

[17] *AnaGram™ - User's Guide*. Parsifal Software, Wayland, MA, USA, 1993.

[18] Pressman, R. *Software Engineering - A Practitioner's Approach*, 3rd ed. McGraw-Hill, 1992.

[19] Paakki, J., Salminen, A. & Koskinen, J. "Automated hypertext support for software maintenance". Submitted for publication.

[20] Reps, T. & Teitelbaum, T. *The Synthesizer Generator, a System for Constructing Language-Based Editors*, Springer-Verlag, 1989.

[21] Salminen, A., Koskinen, J. & Paakki, J. "HyperSoft: an environment for hypertextual software maintenance", *NWPER'94, Nordic Workshop on Programming Environment Research.* Lund, Sweden, pp. 25-37, 1994.

[22] Salminen, A. & Watters, C. "A two-level structure for textual databases to support hypertext access". *J. American Society for Information Science* 43, 6 (July 1992), pp. 432-447.

[23] Smith, K. *PAT: An Interactive Fortran Parallelizing Assistant Tool.* Ph.D. thesis, Georgia Inst. of Technology, GA, USA, 1988.

[24] Waddle, V. "Production trees: a compact representation of parsed programs". *ACM TOPLAS* 12, 1 (Jan. 1990), pp. 61-83.

[25] Wilde, N., Chapman, A. & Richardson, R. "The extensible dependency analysis tool set: a knowledge base for understanding industrial software". *J. Software Engineering and Knowledge Engineering* 4, 4 (Dec. 1994).

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

IV

AUTOMATED HYPERTEXT SUPPORT FOR SOFTWARE MAINTENANCE

Paakki, J., Salminen, A. & Koskinen, J. 1996.
The Computer Journal 39 (7), 577-597.

(C) 1996 The British Computer Soc. Reproduced with permission.

Automated Hypertext Support for Software Maintenance

JUKKA PAAKKI[1], AIRI SALMINEN[2] AND JUSSI KOSKINEN[2]

[1]Department of Computer Science, PO Box 26, FIN - 00014 University of Helsinki
[2]Department of Computer Science and Information Systems, University of Jyväskylä, PO Box 35, FIN - 40351 Jyväskylä, Finland
Email: koskinen@cs.jyu.fi

A model called HyperSoft is presented, which can be used for viewing programs as hypertext. The main goal in developing the model has been to offer a framework for new program browsing tools to support the maintenance of legacy software in particular. The model consists of four layers: source code as such, its syntactic structure, hypertextual access structures based on the source code and its syntax, and the user interface for viewing and manipulating the source code and the access structures. The access structures are based on a general relational model of program dependencies. Both the hypertextual software model and the program dependency model are language independent and provide for a systematic and automated way of representing programs as different kinds of dependency graphs. The models are implemented in a program browsing tool which analyses C programs and automatically generates relevant hypertextual representations for them, according to requests of the maintainer.

Received July 14, 1995; revised October 28, 1996

1. INTRODUCTION

Software maintenance captures all those tasks in software development that are needed after the software has been released for the first time. Maintenance has become the most expensive phase of software engineering, due to the large number of old legacy systems still evolving and to the immature state of software reuse. Systematic and automated approaches to software maintenance are thus of great practical importance, as proposed for example in [1].

Hypertext integrates text with non-linear navigation capabilities. This makes it possible to read a textual representation flexibly in an arbitrary order by following links between related nodes for text fragments. Therefore, an access structure based on hypertext can be used as an aid both for program understanding and for systematic maintenance, provided that the nodes and links of the access structure over the program are properly selected to support the comprehension and maintenance activities.

Viewing programs as hypertext is especially interesting because program text fragments and their relationships to each other have been extensively studied within programming language research, and because there are many methods and tools for automatic recognition of the fragments and their relationships. These relationships are often called program dependencies. For example EDATS [2] is an extensible dependency analysis tool set recognizing a number of different dependencies between entities in object-oriented programs (dependencies like 'isParameterOf', 'isReadBy', 'isLocalOf', 'isOfType', 'sendsMessage', 'superClassOf'). Useful dependencies also include definition—use relationships [3] and calling dependencies. Calling dependencies are often represented in the form of call graphs, which can be formed automatically [4, 5]. Call graphs typically contain nodes representing functions of the program, and directed arcs between the nodes representing the existence of function call(s). Program slicing means the extraction of relevant statements from the source programs into a slice. Slicing is typically based on data and control flow analysis. Techniques for forming the slices are represented, for example, in [6, 7, 8, 9].

Thus, there exist numerous methods and tools offering support for linking program parts. In principle, these methods may be used for the automatic conversion of program text to hypertext. In practice, however, it is not always clear what the program parts are in different cases of dependencies, how they can be linked and what benefits the linking could offer to a person reading the program text. The links should be used to support effective navigation of the source code. In case of large programs, it is far from clear what effective navigation support means.

In the HyperSoft project our aim is to develop an environment in which we are able to study and evaluate different hypertextual access structures for program text. Instead of *ad hoc* solutions, we want to express the hypertext features based on a clear model where the navigation interface is clearly separated from the hypertextual access structure, and the access structure from the hierarchic program structure. In this paper we are going to describe our model for viewing programs as hypertext, and to discuss what kind of hypertextual access structures may be created in terms of well-known program dependencies. We are not looking for

a static set of nodes and links but different possibilities for dynamic specifications of structures to support hypertext access. The HyperSoft method was originally introduced in [10], and the architecture of the HyperSoft system in [11].

The background and motivation of our work is given in Section 2 followed by a short overview of the HyperSoft model in Section 3. The four layers of the model are described in Section 4. Another general model, for program dependencies, is presented in Section 5. Section 6 introduces a tool based on our general models. Finally, the main contributions of the approach are summarized in Section 7.

2. MOTIVATION AND RELATED WORK

The HyperSoft system will be targeted to industrial legacy systems that are currently under extensive non-automated maintenance. As a preliminary case we have studied the maintenance of a large distributed administrative system. The system has the following characteristics of a typical legacy system.

The system has been developed in a period of 15 years; the system has several customized versions which all have to be maintained all the time; the system is large, consisting of about 14,000 program files and several millions of source code lines; the system has been implemented using several languages (mostly C and SQL) and a number of different programming styles; the system has a large number of devoted customers, and a complete re-design is out of the question; the documentation is for the most parts outdated; the system is assigned a dedicated maintenance group of about 10 programmers, thus implying significant cost.

Software maintenance has often been divided into categories based on the nature of the related maintenance requests, see for example [12]. Considering the maintenance activities applied to the sample system, it was found that most of the activities belong to the adaptive class of maintenance reflecting changes in the system environment. The system provides, for example, complete management of personnel and salary information for an organization. The information is strictly based on existing laws and regulations, and must therefore always be correct and up to date. Hence, most of the maintenance effort is laid on introducing regularly appearing official re-regulations into the system. Another frequently occurring maintenance request is to provide a more friendly interface or better reports to the users. Such activities belong to the perfective maintenance category. The term corrective maintenance refers to corrections of errors, while the term preventive maintenance has been coined for those modifications that raise the quality of the software and thus make it easier to maintain in the future. In the evaluation of the sample system it was found that corrective maintenance is rather rare since the general behaviour of the system is quite stable. On the other hand, many programming resources are needed for preventing the maintaining modifications causing new errors in the software.

In software hypertext systems, hypertextual access structures often connect various program documents to each other and to the source code. Examples of such systems are the tools reported in [13, 14]. In the case of legacy software, however, there probably is no documentation or the documentation is outdated. This emphasizes the problems of localizing the relevant program parts and of understanding their dependencies. Soloway and his co-workers have carried out studies with professional programmers engaged in a maintenance task [15]. They observed that the programmers had difficulty in understanding delocalized plans—that is, pieces of code that are conceptually related but physically located in different regions of the program. The subjects often did not realize that there were more pieces fragmented in the plan and hence the enhancements they made often turned out to be incorrect. A system called Whorf [16] was designed to support the programmer in recognizing delocalized plans. Whorf is a program visualization tool where different windows are generated, showing different representations and views of the program.

Whorf as well as some other program browsing and visualization tools, like DynamicDesign [17], PUNS [18], CARE [19] and EDATS [2], identify program dependencies which are (or can be) provided in the HyperSoft environment as well. Several systems are able to show call and class hierarchies, for example the class browsers for Smalltalk [20] and for C++ [21]. It has, however, been noticed that representing well-known program dependencies as separate windows—either graphically or textually—may lead to techniques which are not used by the programmers. Lakhotia [22] reports some experiments on program maintenance and concludes that tools displaying call graphs have not been very useful. Instead, he suggests capabilities for database queries or hypertext-oriented browsing.

In the HyperSoft project our aim is to combine query capabilities with hypertextual navigation capabilities; a query creates a new access structure which the programmer may then navigate on the source code. The HyperSoft model provides a basis for developing and testing different hypertextual access structures and navigation techniques. With the clear separation of different layers we are able to evaluate the interface and the access structures separately in the HyperSoft system. Also the access structures are autonomous, and can be generated and studied independently.

3. AN OVERVIEW OF THE HYPERSOFT MODEL

The HyperSoft model divides a software system into four layers: the interface layer, the access structure layer, the syntactic structure layer, and the source code layer, as illustrated in Figure 1. The HyperSoft layers have a correspondence to the layers of the Dexter model [23]. The layers of the Dexter model are the run-time layer, the storage layer and the within-component layer. The Dexter run-time layer as well as the HyperSoft interface layer deal with the presentation of the hypertext and user interaction. The Dexter storage layer models the basic essence of the hypertext in the same way as the HyperSoft access structure layer, containing the nodes and links. The Dexter

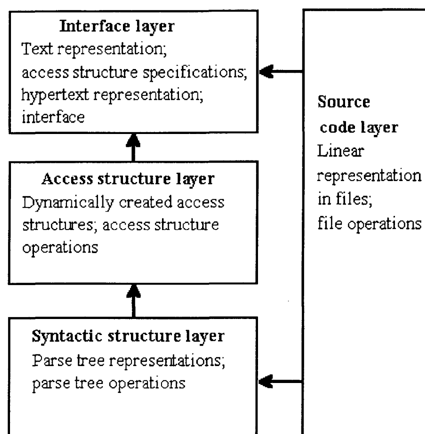


FIGURE 1. Layers of the HyperSoft model.

within-component layer is concerned with the contents and structure inside the nodes. In the HyperSoft model, there are two layers corresponding to the Dexter within-component layer: the syntactic structure layer describes the hierarchic structure of the whole program text and the source code layer contains the source code files.

The purpose of developing the Dexter model has been to capture the important abstractions found in a wide range of existing and future hypertext systems [23]. The HyperSoft model instead is an attempt to capture the important abstractions found in program text and combine them to the abstractions concerning dynamic hypertext. The similarities and differences between the HyperSoft model and the Dexter model will be discussed more in Section 4.

The HyperSoft model is derived from a hypertext model defined for structured text in general, meaning text defined by a grammar [24]. In the HyperSoft model, the data of a hypertextual software system consists of a set of files, each containing a piece of linear program text. This original text is considered at the source code layer. The linear text contains the program in the form the original programmer has authored it. From the viewpoint of program understanding, an important structure in program text is its hierarchic structure, defined by the grammar of the programming language. The maintainer of the program has to understand the components in the hierarchic structure. In the HyperSoft model, this structure is expressed at the syntactic structure layer as a parse tree with respect to the grammar. The syntactic structure layer also contains other compiler-oriented data structures describing the program's components, most notably a symbol table. Imposing a hierarchic structure on hypertext is a method extensively used for solving disorientation problems, see for example [25, 26]. In the HyperSoft model, a hypertextual access structure is always defined over a hierarchic structure.

Since the dynamic nature of hypertext is essential to the HyperSoft model, the access structure layer is regarded as a dynamic component of the system. During program reading, the programmer may create new access structures. The access structure layer contains a set of different access structures which the user is able to operate on. Each of the access structures may be regarded either as a set of nodes or as a set of nodes and links. The nodes of the access structure are always nodes of the parse tree, each representing a part in the hierarchic text structure. An access structure describes a specific kind of dependency between program parts. The dependency may be as simple as a common property of the parts; for example, an access structure may consist of the output statements of the program. On the other hand, the dependency may also be a more complicated relationship expressed as a graph, such as a slice or a call graph.

The interface layer allows the user of the system to display source code on windows, specify access structures, manipulate access structures, and navigate in the current access structure. The dynamic creation of access structures, based on the current information requests of the maintainer, is our way to reduce both the disorientation and the cognitive overhead problems. In many systems, cognitive overhead occurs in the process of reading hypertext which tends to present too many choices about which links to follow and which not to follow [27]. In our approach the generic user interface supports different access structures, and appropriate browsing strategies [28] are designed for the specific access structure categories. The number of possible links available is minimized by showing in a window only those which support the specific information needs. Another practical decision is to present the source program in its original textual form, so as to map the nodes of an access structure directly to the program components subject to maintenance.

4. PROGRAMS AS HYPERTEXT

In the previous section we gave an overview of the four layers of the HyperSoft model, illustrated in Figure 1. The figure also showed how the data is transferred between the layers. In this section we will address each of the layers and the data transfer between them in more detail.

4.1. Source code

The basis of the text handled in a software system consists of a set of source code files. Hence any software system needs capabilities to read and update text in the files. In the HyperSoft model, the source code files as well as the capabilities for handling them are included in the source code layer. Central questions related to this layer include:

- How is the source code stored on disk, using either file systems or database systems?
- What are the mechanisms to retrieve the source code from disk?

In hypertextual program reading, the functions retrieving pieces of the source code are needed at two other layers. For creating the hierarchic parse tree, the syntactic structure layer needs to analyse the original source code. It retrieves the text from the source code layer, as shown in Figure 1. Also, when the text is displayed on windows by the interface layer, the text is retrieved from the original files. Thus there is data transfer from the source code layer to the interface layer as well.

4.2. Syntactic structure

Program text is a case of a currently quite common type of electronic text, i.e., text defined by a context-free grammar. For example, all SGML documents are defined by a context-free grammar [29]. The basic notions concerning grammars and parse trees may be found for example in [30]. The notions of the syntactic structure layer in HyperSoft are taken from a general grammar-based text model described in [24, 31]. The central questions which have to be answered at this level in order to form the hypertext include the following:

- What is the information that should be statically (permanently) stored in the program database in order to make the hypertext generation possible and efficient?
- What is the method of creating the program database?
- What is the storage form of the program database information?

The grammar of a programming language specifies the character strings accepted as programs of the language, and the hierarchic structure of a program is expressed in its parse tree representation. In the language compiler, the meaning of program text is expressed by associating semantic specifications with the elements of the hierarchic structure. To be able to understand the program, the maintainer has to be able to identify the elements in the structure defined by the grammar. In the HyperSoft model, the syntactic structure layer contains the parse tree and the symbol table representations of a program, the capabilities for creating them, and the operations for retrieving information from them.

As a sample program throughout the paper we will consider the C program shown in Figure 2. Its parse tree is sketched in Figure 3. The parse tree is represented only partially, so that the non-terminals whose content is represented as triangles (and the associated program text) are not further elaborated in the figure. The associated source code text is written in boldface.

The parse tree is created by analysing the source code shown in Figure 2 with respect to a context-free grammar with the following productions (only the relevant productions are listed):

```
int g1=1,g2=2,g3=3;

int main(void) {
/* This imaginary SPC is used to */
/* illustrate the generation and use of */
/* HyperSoft access structures */
  int in=0,out=0;
  printf("Input:");scanf("%d",&in);
  out=f1(in)+f2(in);
  printf("\nOutput=%d",out);
  return 0;
}
int f1(int p1) {
  return p1-g3+f4(p1,g1,g2);
}

int f2(int p1) {
  int g1=g2+p1;
  return g1;
}

int f3(int *p1,int *p2) {
  int t=*p1;*p1=*p2;*p2=t;
  return(*p1>*p2);
}
```

FIGURE 2. A sample C program.

(1) program → translation-unit eof
(2) translation-unit → external-declaration | translation-unit external-declaration
(3) external-declaration → declaration | function-definition
(4) declaration → ...
(5) function-definition → declaration-specifiers declarator compound-statement
(6) declaration-specifiers → type-specifier
(7) type-specifier → **'int'** | **'void'** | ...
(8) declarator → direct-declarator | ...
(9) direct-declarator → identifier | direct-declarator '(' parameter-type-list ')'
(10) identifier → ...
(11) parameter-type-list → type-specifier | ...
(12) compound-statement → '{' declaration-list statement-list '}'
(13) declaration-list → ...
(14) statement-list → statement | statement-list statement
(15) statement → expression-statement | ...
(16) expression-statement → ';' | expression ';'
(17) expression → assignment-expression | ...
(18) assignment-expression → unary-expression assignment-operator assignment-expression | ...

Non-terminal symbols are indicated with alphabetic names, terminal symbols are embedded within a pair of apostrophes ('), → separates the left-hand side of a production from its right-hand side, and alternatives are indicated by the symbol |. The grammar is not further refined for the non-terminals that are not relevant to the examples.

The parse trees handled in software systems usually follow the abstract syntax of the program text; not the concrete syntax. The delimiters needed for analysing textual elements, not carrying any semantic information, are not needed in the parse tree. For example, in Figure 3 the braces surrounding a compound statement can be excluded from the tree. Thus the actual grammar defining the parse tree is

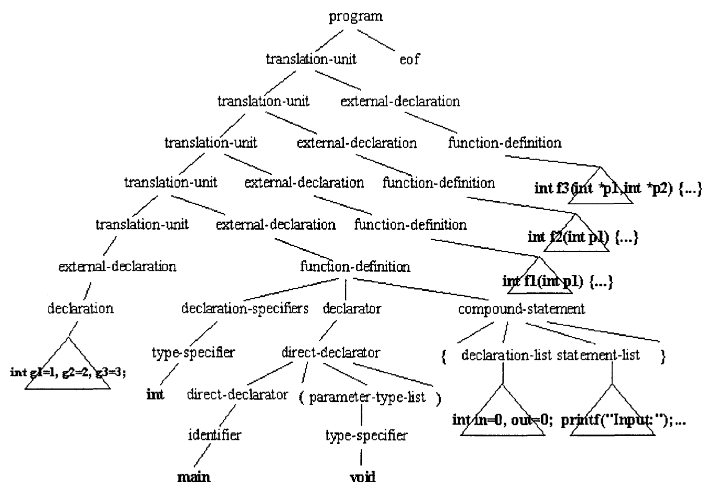


FIGURE 3. A partial parse tree for the program in Figure 2.

usually a transformation of the grammar used for analysing the source code. For example, in the grammar describing an abstract form of the parse tree in Figure 3, production (12) of the original grammar above would be replaced by the production

(12) compound-statement → declaration-list statement-list

Grammar transformations and corresponding text transformations are discussed for example in [30]. We suppose that the parse tree X at the syntactic layer of HyperSoft either describes the concrete syntax, or is created such that any subtree in X always has a unique counterpart in the parse tree X' that completely describes the concrete syntax corresponding to the original grammar. Thus for any subtree in X we are able to specify a corresponding substring in the original source code. The substring consists of the terminal symbol labels in the subtree of X'.

Each non-terminal of a grammar represents a set of textual entities. Therefore, a non-terminal of the grammar is called a text type. In the C grammar there are, for example, text types *program*, *external-declaration*, and *function-definition*. Given any program, each of the non-terminals stands for a set of text parts in the program. In the parse tree, the parts are represented by nodes labelled by non-terminal symbols (and the respective subtrees). Since each part in the parse tree should correspond to an identifiable substring in the source code, a node is regarded as a part only if it is not the single child of its parent. The node labels indicate types of parts such that the label of a single child renames a part.

HyperSoft's syntactic structure layer together with its source code layer correspond to the within-component layer of the Dexter model. The within-component layer is concerned with the content and structure of the components in the hypertext network. A node might be, for example, an SGML document or a program text. The HyperSoft model starts the building of a hypertextual access structure from a program which may be very large (millions of lines, thousands of files). The files and their handling are described at the source code layer. The analysis of the files and the creation of the syntactic structure may be incremental. The abstract model, however, is based on the idea that the parse tree at the syntactic structure layer describes the whole program. The hypertextual access structure is then defined on top of the syntactic structure. This provides us the flexibility we need for defining different kinds of access structures for one program.

4.3. Access structure

The access structure layer contains a set of access structures, algorithms for creating them, and operations on them. For creating the access structures, the access structure layer retrieves data from the parse tree of the syntactic structure layer. Potentially useful access structures for a program text, corresponding to different categories of program dependencies, will be discussed in Section 5. The questions related to this layer include the following:

- What are the useful access structures?
- What is the type of seed for each access structure type (variable, function, syntactic type, or something else) and the way to define it (using a reference, complex expressions, a query language)?

```
int g1=1,g2=2,g3=3;

int main(void) {
/* This imaginary SPC is used to */
/* illustrate the generation and use of */
/* HyperSoft access structures */
  int in=0,out=0;
  printf("Input:");scanf("%d",&in);
  out=f1(in)+f2(in);
  printf("\nOutput=%d",out);
  return 0;
}
int f1(int p1) {
  return p1-g3+f4(p1,g1,g2);
}

int f2(int p1) {
  int g1=g2+p1;
  return g1;
}

int f3(int *p1,int *p2) {
  int t=*p1;*p1=*p2;*p2=t;
  return(*p1>*p2);
}
```

FIGURE 4. A simple access structure.

- Should different node and link types be used within a single access structure type?
- Would it be useful to combine the access structures and how could this be done?
- Should it be possible to store the access structures for later use?

For browsing purposes hypertext is often modelled as a directed graph, i.e. a pair (Z, E) where Z is a set of elements called nodes and E is a set of node pairs called links. The nodes of a link are called the start node and the destination node. In HyperSoft, the access structure the end user is navigating is also a directed graph. The access structure is called a navigation structure. The nodes of the graph as well as the nodes in all access structures are parts in the hierarchic text structure, in other words, nodes in the parse tree of the syntactic structure layer.

Figure 4 shows an access structure for the sample program of Figures 2 and 3. The navigation structure is denoted on top of the source code text. A part appearing as a hypertext node is indicated by a frame around the value of the part in the source code. Links are indicated by arrows. The navigation structure consists of the parts of type *declaration* or *assignment-expression* in the program, linked according to their preorder in the parse tree. The purpose of this navigation structure is to capture all those elements in the program where a variable can be given a value.

The user creates an access structure for specific information needs. The new structure may be a navigation structure with links. On the other hand, the user may also specify an access structure simply as a set of parts of the program text. Considering an access structure as a set allows the application of hypergraph-based hypertext models represented in [24, 32], which offer operations on node sets. For example, considering two access structures as sets it is possible to create their union, intersection or difference. A navigation structure can be derived from a set of nodes by specifying the way links are created between the nodes. A simple way to specify the links is to define an order among the nodes. The order may be based, for example, on the preorder of the nodes in the parse tree. If the text corresponding to the nodes is located in the same file then the order is the same as the textual order in the file. If the parse tree has been created from a program consisting of a set of files then a link may connect parts whose values in the source code are located in different files. The navigation structure shown in Figure 4 might be created in the following steps:

1. create an access structure consisting of the parts of type *declaration*,
2. create an access structure consisting of the parts of type *assignment-expression*,
3. create the union of the two access structures,
4. create the links between the nodes according to their preorder in the parse tree.

The access structure layer of the HyperSoft model corresponds to the storage layer of the Dexter model; in both cases the nodes and links are expressed at this level. In the Dexter model the term node is used sometimes but the fundamental entity and basic unit of addressability in the storage layer is a component. In HyperSoft, the fundamental entity is a program part, i.e. a node in a parse tree. In the Dexter model a component is either an atomic component, a link or a composite entity made from other components. Since the within-component layer is intentionally not elaborated within the Dexter model, the model needs a special mechanism for the interface between the storage layer and the within-component layer. This mechanism is called anchoring. The HyperSoft model is specially designed for structured text, i.e. text defined by a grammar. The syntactic structure layer contains the program text as a hierarchy of parts such that each part is uniquely addressable. Therefore no special mechanism is needed for handling the interface between the two layers. The hypertext nodes are always program text parts, and a link is a relationship between parts. The way nodes and links are expressed on the screen is determined at the interface layer.

4.4. Interface

The interface layer of HyperSoft allows the display of the source code in windows, the specification of access structures, the manipulation of the access structures and the navigation in the current access structure. In principle, the source code is supposed to be displayed as it has been written. In addition to the text itself, the nodes and links have to be visualized somehow. A node of an access structure is represented on the screen by the value of the part, i.e., the source code substring corresponding to the part in the parse tree. The following are examples of the central questions which have to be answered at the interface layer:

- What kind of windows are created?
- What is the technique for specifying specific access structures?
- How are the nodes of the current access structure shown on the screen?
- How are the links expressed on the screen?
- How is navigation achieved?

Concrete examples of the interface layer will be given in Section 6 where we present different navigation structures as currently provided in the HyperSoft prototype.

5. RELATIONAL DEPENDENCY MODEL

The core of accessing software in the HyperSoft model is to extract potentially useful dependencies automatically from the program at the access structure layer. These dependencies are then evolved into navigation structures and provided to the user in the form of hypertext at the interface layer.

The central problem in this kind of approach is to select the useful dependencies to be provided to a maintainer. The research of software analysis and compilation techniques has invented a large number of program dependencies, all more or less relevant in their application area, e.g. [12, 33]. An investigation into well-known program dependencies that have significance especially in the context of software maintenance is given in [10].

To capture the essence of program dependencies, we present a classification of them in this section. Our classification is based on considering dependencies as relations between text parts; recall from Section 4 that each text part stands for a specific syntactic element in the program and for a node in its parse tree. We will discuss just the main principles of regarding program dependencies as relations; a more detailed characterization is given in [34].

As usual in modelling, the classification is abstract and general in the sense that it does not directly express the concrete particular program dependencies but instead their common characteristics and relationships as dependency categories. Hence, in object-oriented terminology, a category corresponds to a class, while each actual dependency corresponds to an object (an instance of its category). Finally, a hypertextual access structure is derived from the dependency instances using standard relational primitives.

Our classification introduces different categories of dependencies. General (super)categories are refined into subcategories on several levels according to certain criteria. The categories can be characterized in terms of their relational properties, such as (R1) reflexivity, (R2) symmetry, (R3) transitivity, and (R4) antisymmetry which are defined as follows:

(R1) xRx for each part x in the relation R;
(R2) $xRy \Rightarrow yRx$;
(R3) $xRy \wedge yRz \Rightarrow xRz$;
(R4) There are no parts x and y, $x \neq y$, such that xRy and yRx.

Besides expressing the essential differences between the dependency categories, these relational characteristics also have significance in suggesting both how to extract the corresponding access structures from the program and how to explicitly present them at the interface layer of HyperSoft.

5.1. Classification

Our classification of program dependencies is shown in Figure 5[1]. The classification is given in the OMT notation [35] where the refinement of a supercategory into subcategories is expressed with a triangle. Each category is specified by giving its name and properties. (For simplicity, we do not make a notational distinction between data properties and algorithmic properties.) Each subcategory inherits the properties of its supercategories.

Since we express a (binary) dependency as a hypertextual link, each dependency involves a start node and a destination node. For different kind of dependencies, the nodes may be of different text type. This is modelled in the classification by the properties Start types and Destination types, respectively, that are inherited from the root category Dependency to all the subcategories. These properties are related to navigation structures and will be discussed in Section 5.2.

Each dependency instance also involves a specific algorithm to find all the pairs of program parts that are in the particular dependency relation. This is modelled by the property Algorithm. Let $a\ R\ b$ denote that parts a and b are in the dependency relation R. Then Algorithm of the dependency instance R can be exploited to automatically find all such pairs (a, b) from the program and, as a consequence, to build the access structure corresponding to R.

5.1.1. Matching dependencies

At the top level of the classification the dependencies are divided into Matching and Subordination ones. Intuitively, a relation $R1$ belongs to the category Matching if $a\ R1\ b$ implies that the parts a and b are somehow similar in their syntax or semantics, while a relation $R2$ belonging to the category Subordination means $c\ R2\ d$ to imply that part c dominates or has control over part d. The Matching relations are reflexive (R1), symmetric (R2) and transitive (R3) (hence, equivalence relations).

For a Matching dependency $a\ R\ b$, the similarity of a and b can be defined on either a lexical, syntactic or semantic basis. If R is Lexical, then a and b have a similar pattern of pure text; in other words, a and b can be defined with the same regular Pattern expression over characters. The involved property Algorithm then finds all those parts from the program that are textually similar, as proposed e.g. in [36]. For instance, a Lexical relation might include all the occurrences of names starting with the string 'foo': 'f' 'o' 'o' *any** where any stands for any letter or digit.

Property Pattern for a relation R in subcategory Syntactic specifies the syntactic shape of the parts. In other words,

[1] For simplicity, we have omitted some categories that are not essential for the main subject of this paper.

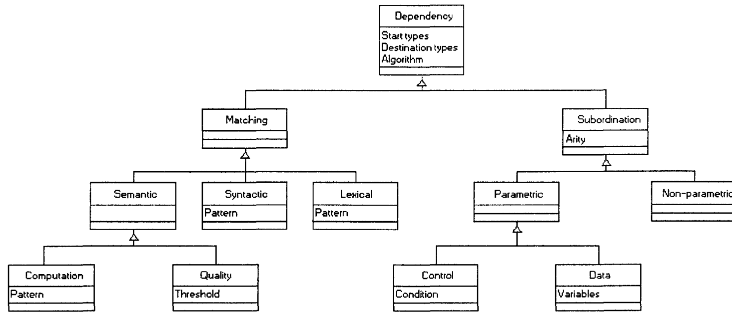


FIGURE 5. Classification of program dependencies.

a R b implies that the syntax of parts *a* and *b* is defined with the same (abstract) context-free grammar, and that the subtrees for *a* and *b* are isomorphic in a parse tree. Property Algorithm for a Syntactic category typically applies pattern matching in (parse) trees, as suggested e.g. in [37, 38]. For instance, a Syntactic relation might extract from the parse tree all the parts of the same Syntactic type, as was illustrated in Figure 4.

Two parts *a* and *b* in a Semantic relation either express the same computation (subcategory Computation) or two program structures with the same software engineering quality (subcategory Quality). In the former case, the semantic property Pattern may stand e.g. for a data-flow computation [7] or a specific algorithm (a cliché) [39]. For instance, a Computation relation might capture all the different sorting algorithms used in the software. Two parts are in a Quality relation if they have the same Threshold value characterizing some quality factor. Cohesion and coupling, for instance, are two well-known approximations for the quality of modules [12] and cyclomatic complexity [40] is a standard metrics for computational and mental complexity of programs. Thus, a Quality relation might e.g. hold between subprograms that are too complex for a sound maintenance, being at the Threshold level 10 or higher in their cyclomatic complexity.

The similarity of the parts in a Matching relation can be graphically illustrated with a special spatial arrangement of them at the interface layer. For instance, an affinity browser [41] has been proposed for visualizing a relative Syntactic or Semantic similarity of the classes in an object-oriented program.

5.1.2. Subordination dependencies

As characterized above, one part somehow dominates the other in a Subordination relation. For instance, *a R b* might mean that part *a* contains part *b* as one of its components (in the case of structured types), that *a* is a superclass of *b* (in the case of classes of an object-oriented program), or that *a* calls *b* (in the case of functions). The Subordination relations are transitive (R3) and usually antisymmetric (R4) as well. However, the flexibility of functions in most programming languages makes, for instance, the calling relationship non-antisymmetric in the general case: there may well exist a mutual calling dependency (*a Calls b, b Calls a*) between two functions *a* and *b* of the same program. Hence, the non-antisymmetric domination dependencies (such as Calls) belong to the category Subordination, whereas the antisymmetric ones (such as Contains and IsSuperclassOf) are located in its subcategories.

Unlike a Matching dependency, a Subordination dependency is not symmetric. Therefore it always has a meaningful inverse dependency relation that can be generated in a straightforward manner from the original one. For instance, the inverse of *a IsSuperclassOf b* is *b IsSubclassOf a*, and the inverse of *a Calls b* is *b IsCalledBy a*. The inversion property of Subordination dependencies is utilized in the HyperSoft prototype for some central navigation structures (see Section 6).

In general, a (non-equivalence) relation *a R b* is either of type *1-to-1, 1-to-n, m-to-1* or *m-to-n*, depending on the possible arity of *a* and *b*. In the context of HyperSoft the 'arity' means how many different parts for *a* and *b* there are such that *a R b* holds. For example, a dependency *R* is of arity *1-to-1*, if there is a single part *a* and a single part *b* in *a R b*, and of arity *1-to-n* if *a* is unique but there may be several different *b*s in *a R b*. The arity of a dependency is represented in our model by the property Arity associated with the Subordination category.

Notice that the inverse of a *1-to-n* dependency belongs to the *m-to-1* category and, accordingly, the inverse of an *m-to-1* dependency is of arity *1-to-n*. The inverse of a *1-to-1* dependency is also *1-to-1*, and the inverse of an *m-to-n* dependency remains in the arity type *m-to-n*.

Subordination is specialized into two direct antisymmetric subcategories, Parametric and Non-parametric. A Parametric dependency *R* entails some additional information to make *a R b* hold, while there is no such parameter for the Non-parametric relations.

For example, the *IsSuperclassOf* dependency mentioned above, mapping a class with its subclasses in an object-oriented program, is of category Non-parametric with arity *1-to-n* in the case of single inheritance. Therefore the inverse dependency, *IsSubclassOf*, is Non-parametric with arity *m-to-1*. The Subordination dependency Calls and its inverse *IsCalledBy* are usually of arity *m-to-n*; this shows the fact that a function may call several functions and may itself be called from several functions. Hence, a conventional call graph usually takes the form of a general graph where a node may be connected to several other nodes in both directions. The same holds for multiple inheritance in object-oriented programs: in that case both the dependency *IsSuperlassOf* and its inverse *IsSubclassOf* are of arity *m-to-n* rather than of arity *1-to-n* and *m-to-1* as in the case of single inheritance.

5.1.3. *Control and data dependencies*

When a program is executed, it basically involves merely control (determining the applied statements and their relative invoking order) and data (capturing the computed values). In our model, these fundamental concepts appear as dependency categories Control and Data, respectively. The dependencies in these categories are associated with parameter information, as expressed by their common supercategory Parametric.

The property Condition of Control denotes a Boolean expression c whose value shall be True in order to realize an instance of the dependency during program execution. In other words, a Control relation $a\ R\ b$ with Condition c means that the part b will be executed immediately after the part a only if the involved condition c yields True when executing the program. Notice that an unconditional flow of execution from a to b is not represented by a Control dependency but rather by a Non-parametric one. Hence, in a conventional control-flow graph each pair of successive parts is either in relation Non-parametric of arity *1-to-1* (statement sequences), Non-parametric of arity *m-to-1* (unconditional loops), or Control of arity *1-to-n* or *m-to-n* (conditional statements and controlled iterations). Such a many-faced graph can be characterized as a hypertextual navigation structure by combining the individual elementary dependencies (here: Non-parametric */1-to-1*, Non-parametric */m-to-1*, Control */1-to-n*, Control */m-to-n*) into a compound dependency (see Section 5.2 below).

The property Variables represents the set of variables that move data from part a into part b for a Data dependency $a\ R\ b$. Usually, an instance of such a dependency expresses an immediate flow of data from statement a to statement b during the program's execution. Notice that in this sense a data flow from a to b always implies that a and b are in a (transitive) Control relation as well.

Since the Parametric relations R stand for possible execution flows within a program, they are, besides transitive and antisymmetric, also irreflexive:

(R5) There is no part x such that xRx.

Notice especially that a direct loop from a statement into itself is always an unconditional jump and thus not of category Control or Data but rather of category Non-parametric with arity *m-to-1*.

5.2. Dependencies as navigation structures

Having set up the dependency model, we can precisely define what a navigation (access) structure means in HyperSoft. First, the basic dependencies presented above are generalized into a joint compound dependency:

Let $R_1, R_2 \ldots, R_n (n \geq 1)$ be concrete dependencies, that is, instances of the dependency categories given in Figure 5. A compound dependency R_c is defined as a relational expression over the elementary dependencies:

$$R_c = R_1 \circ R_2 \circ \ldots \circ R_n$$

where each occurrence of the symbol $\circ$ stands for one of the set operations $\cup$ (union), $\cap$ (intersection), (difference)[2].

As a set expression, a compound dependency defines the nodes for an access structure. To obtain a hypertextual navigation structure as a graph, the directed links have to be defined as well. For the antisymmetric Subordination dependencies R_s the linking is obvious. For each relation $a\,R_s\,b, a$ will be the start node and b the destination node; hence, there will be a directed arc from a to b in the navigation structure. For the symmetric Matching dependencies R_m the direction is not that obvious. In the HyperSoft prototype, the usual choice is to apply the textual order of the nodes: For a relation $a\,R_m\,b, a$ will be the start node and b the destination node if a precedes b in the preorder over the parse tree; otherwise b will be the start node and a the destination node[3].

In the simplest case a compound dependency consists of a single elementary dependency ($n = 1$) which directly yields the corresponding navigation structure. Often, however, the navigation structure is a hybrid one and formed in terms of several elementary program dependencies. For instance, the navigation structure shown in Figure 4 is spanned by the compound Syntactic dependency *IsOfType* (*declaration*) ∪ *IsOfType* (*assignment-expression*).

Usually a hypertextual navigation structure has a specific program part r that the user has selected as the root when dynamically specifying the access structure in which she or he is interested. As an example, the mandatory slicing criterion (usually a variable occurrence) is the root of a conventional forward or backward slice. There might be an implicit root even in cases when the user has not explicitly defined one. For instance, applying the pre-order for linking has introduced a root node in the navigation structure of Figure 4. For a user-specified criterion, a navigation structure is the reflexive transitive closure of a compound dependency relation R_c with respect to the specified root part r: The navigation structure contains all those parts for which the relation $r\,R_c^*$ holds. The initial node a_0 in

[2]Negation ($\neg$) is not applied since it is not useful in this context.

[3]Note, however, that the HyperSoft model deos not fix the linking direction for symmetric relations.

the navigation structure is r, and the directed graph $a_0 \rightarrow a_1 \rightarrow a_2 \ldots$ is constructed from the chain of dependencies $r = a_0 R_c a_1 R_c a_2 R_c \ldots$

For instance, if R_c is the *IsSuperclassOf* relation and r represents the root of an object-oriented class hierarchy, then the navigation structure corresponding to R_c comprises all the direct and indirect subclasses of the class r. Thus our dependency classification in Figure 5 can be regarded as a hypertextual navigation structure over dependency categories, with Dependency as the root. Combining a Subordination dependency with its inverse makes it possible to traverse the navigation structure in both directions. For instance, having R_c = *IsSuperclassOf* ∪ *IsSubclassOf* makes it possible to trace the intricate 'yoyo' phenomenon in object-oriented programs where control moves up and down the class hierarchy along inherited operations and self references [42].

The special relational properties of the underlying elementary and compound program dependencies affect, besides the navigation structures at the access structure layer of HyperSoft, also their appearance at the interface layer. For instance, one solution to the disorientation problem of large hypertexts is to associate each node in the navigation structure with a direct backtrack link to the root node r. Another reflection is to have several alternative hypertext links both from a start node for dependencies of arity *1-to-n* as well as to a destination node for dependencies of arity *m-to-1*. For dependencies of arity *m-to-n*, both start nodes and destination nodes may have alternative links. Hence, the nodes in a navigation structure over an object-oriented class hierarchy with multiple inheritance would have alternatives both as incoming and outgoing links.

In addition to the notion of a root, other useful concepts can also be defined in terms of the relational properties of our dependency classification, examples being cycles, strongly connected components and the distance between two parts. We do not discuss these concepts here in more detail but refer instead to [34].

Since our hypertextual dependency model is relational, it could well be supported by a special query language, for example in the style provided in EDATS [2]. Such a relational query language is not included in the HyperSoft prototype where the dynamic specifications are given as direct indications on the source code. The development of a query language is one of the possible research directions in the future, as well as e.g. the design of a pattern definition language for dependencies in the Matching category.

In addition to the HyperSoft classification presented in this section, a number of other dependency classifications have been proposed, for instance in [2, 19, 41]. The main contribution of our approach is to have a general classification of possible program dependencies, whereas the other ones can be seen as particular classifications in a concrete sense. In other words, the classifications mentioned above can be characterized as being defined over instances of our universal classification of dependency categories. Another general approach to program dependencies is provided in PegaSys [43]. The main difference to our formalization is that PegaSys is based on a logical calculus whereas HyperSoft applies a relational model.

6. THE HYPERSOFT SYSTEM

A prototype maintenance support tool based on the HyperSoft model has been constructed at the University of Jyväskylä in an ongoing project, started in 1994 and funded mainly by the Technology Development Centre of Finland (TEKES). The project is guided and partially financed by an industrial steering group having representatives from the four largest software houses in Finland: KT-Tietokeskus, Nokia, TT-Tieto/Valtionjärjestelmät and TT-Tieto/Kuntajärjestelmät (formerly VTKK). The design and implementation decisions concerning the system have been discussed and agreed in the steering group. A significant number of the maintenance problems in the partner enterprises are related to legacy systems written in C [44]. Hence C was chosen as the first language to be supported in the system. The system supports the examination of complete C programs. The needs of the partner enterprises have also determined the selection of PCs as the platform and Microsoft Windows as the operating system under which the user interface operates. In Section 6.1 we will first shortly characterize the access structures chosen for the prototype. Then the use of the system, its implementation, and its usability evaluation will be discussed in Sections 6.2, 6.3 and 6.4 respectively.

6.1. Current access structures

In the beginning of the HyperSoft system design, several potential access structures were introduced to the industrial steering group. Of the structures, five were regarded as the most appropriate for attacking the maintenance problems of the participating companies. In the following each of the chosen structures is briefly described; their use in the system will be described in Section 6.2. In the current system, all five access structures are created and stored as navigation structures. In other words the links intended for browsing are included in the access structures. So far no general access structure manipulation capabilities are provided in the system.

An Occurrence list (set) is an access structure which contains the occurrences of a specified symbol. Thus there exists a Semantic Matching dependency among the parts. The set may contain links between the elements. The nodes within the structure are currently ordered based on their preorder in the underlying parse tree. Also, more complex dependencies could be formed, such as Subordination between the definition and use parts. The symbol may stand for a variable, function or macro. Both the start and destination nodes bounded within this structure are of type *identifier* [44]. This structure, as well as the others, could be combined with other access structures to produce more confined or extended compound navigation structures.

An occurrence list is a useful access structure in many different kinds of maintenance situations, by providing

starting points for further browsing. The occurrences of familiar symbol names serve as beacons [15] to the attempts of gaining general comprehension over the program text. The conventional text search is often not sufficiently accurate, especially in the case of overloaded short symbol names, like i. In addition to general comprehension support an occurrence list can be used, for example, to find the occurrences of a variable whose value should be changed (adaptive maintenance) or to rename poorly named symbols (preventive maintenance).

Forward and backward calling structures contain parts of the dependency chains traditionally represented in call graphs. Each of the structures is created for a function chosen by the user. As noted in Section 5, in terms of our dependency classification forward and backward calling access structures contain Subordination dependencies. As an access structure pair, forward and backward calling structures (as well as forward and backward slices described below) are an example of a dependency relation and its inverse relation.

When using systematic comprehension strategies the maintainer tries to gather sufficient information about the program structure and behaviour before making changes to it. This information gathering is often based on tracing the control and data flows of the program. Because program execution follows the control flow and the functional decomposition splits it, there is a need to find the associated function calls and implementations. The often followed top–down comprehension strategy necessitates finding the function implementations (and the associated comments). The automatic methods of showing the call graphs help in this task. In HyperSoft, the forward calling structure is formed only for the function(s) that the maintainer considers relevant to the current maintenance task. Similar to an occurrence list, this structure is useful in a wide range of maintenance situations. It is important to evaluate the possible effects of the intended changes on the functions which use the modified function (ripple-effect analysis), especially when making changes to a lower-level function of a large system. Also, when identifying a usage of a global variable, it would be useful to be able to trace the control flow back to the points where the variable has previously been used. These types of information requests can be satisfied by backward calling structures, as well as by backward slices, described below.

Backward slicing is the more conventional form of slicing introduced by Weiser [45]. Some of the main ideas behind the forward slicing were introduced in [46] and the terminology was systematically first used by Horwitz *et al.* [6]. Forward and backward slices provide a view of a program based on the extraction of somehow relevant statements during tracing of the program's control and data flow. Since control flow can be traced to two directions, it is possible to form both backward and forward slices. Since slices involve both data flow and (dominating) control flow, backward and forward slices are in our dependency model produced from compound dependencies of the form $D \cup C$, where D is of category Data and C is of category Control.

The slicing structures of HyperSoft support variable-based slicing and statement-level linkaging. The slicing criterion is always taken as the root of the corresponding navigation structure.

One main approach to find the programming errors is to trace the control flow of a program backwards from the point where the erroneous (output) value has been identified. Backward slicing is specially tailored for this purpose. A backward slice consists of the statements which may have effect on the slicing criterion, i.e. a symbol of interest, thus helping to locate the cause of the error. In the style of Weiser [45], an intraprocedural backward slicing structure is constructed at the access structure layer of HyperSoft by iteratively solving data-flow equations. Each statement is analysed (without approximations) in both types of slicing in HyperSoft.

Forward slicing has shown its value in two most intricate problems of software maintenance: side-effect analysis of modifications and optimized regression testing. Forward slicing structures of HyperSoft reveal the program parts that can be affected by the modifications, thus helping the user to focus side-effect analysis on those relevant program parts only. A forward slice also shows the magnitude of the effects of a planned change, thus capturing those modules that must be regression tested in connection with maintenance. Our implementation correctly solves the well-known 'calling context' problem [6], which is a problem both in backward and forward slicing. Since we want to give the user the option to choose between preciseness of the slice and its effective formation, no approximations of the interprocedural information are necessarily used.

6.2. The use of the HyperSoft system

When using HyperSoft, the user opens relevant files on the screen, and uses either the mouse or the keyboard to browse the program text, to select a program part to serve as the root of an access structure, and to make a request for an access structure generation. Each source file is displayed in its own window. Program parts belonging to the current access structure are emphasized by using either reverse colour or boundary boxes. Different shades are used to separate start nodes from which zero, one or several links originate as well as those that have already been traversed. The links may be graphically visualized as arrows. The user navigates through the access structure either by clicking the emphasized parts of the program text or by pressing the forward and backward buttons or keys. There is a special button for the root node of an access structure.

6.2.1. *Occurrence lists*

Global variables are considered quite problematic in software maintenance. Figure 6 shows an example occurrence list for the global variable *g1* declared in the module *hsoft1.c* of a sample C program. The user has first selected a *g1* occurrence by double clicking it within the program text, and then the occurrence list access structure from the pop-up menu of the alternative access structures.

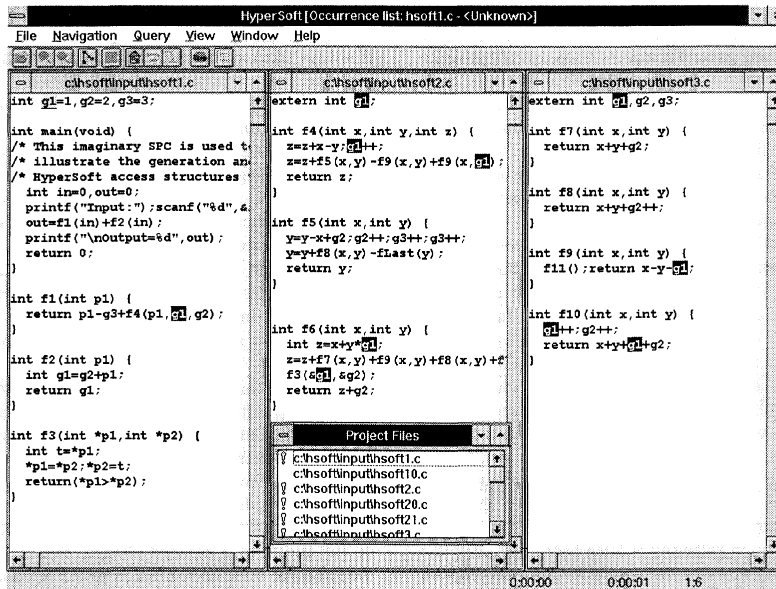


FIGURE 6. Occurrence list.

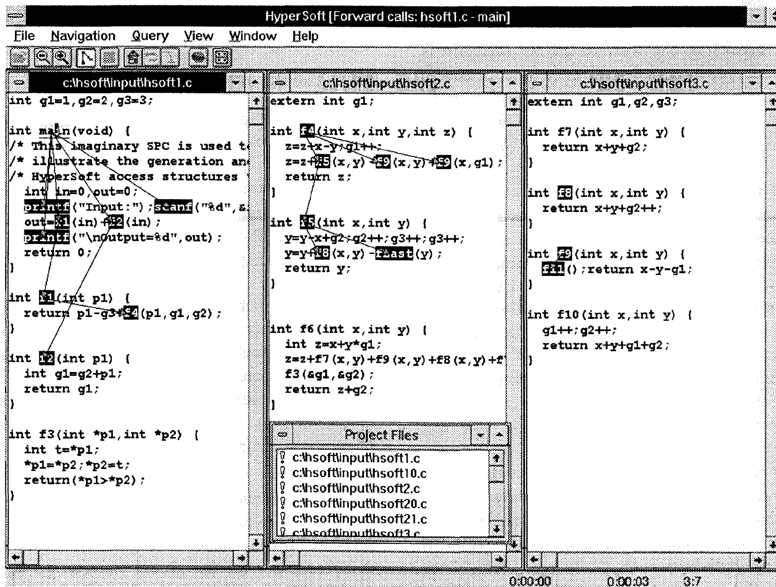


FIGURE 7. Forward calling structure.

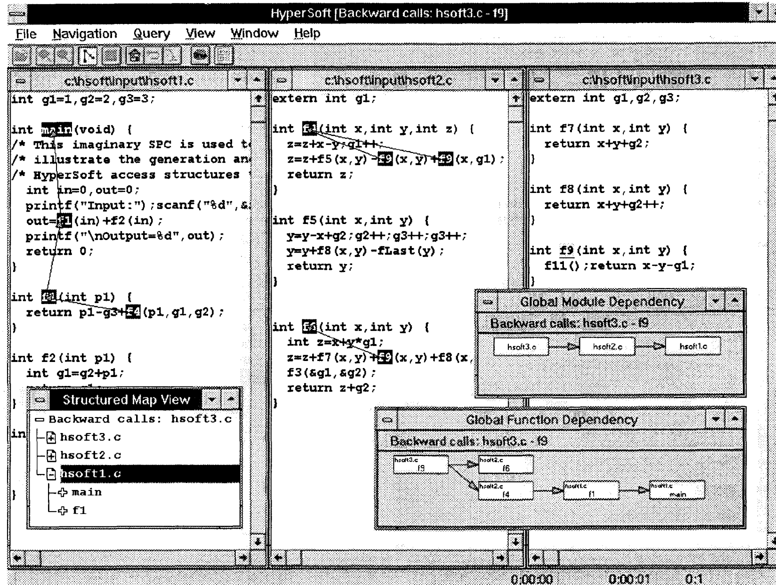


FIGURE 8. Backward calling structure.

This has activated the generation of the access structure which is then shown in the windows. In this case the user has decided not to get the links explicitly on the screen. The analysed program files are listed in the Project Files window and can be opened via it. Files containing parts of the current access structure are emphasized with exclamation marks.

6.2.2. *Forward calling structures*

Figure 7 depicts a situation in which the user needs to find out which functions a certain function uses (forward calling dependencies), thus grasping a general functional view over the program. The generation of the access structure has in this case been initiated from the *main* function, which has therefore become the root of the navigation structure. The linkage is formed such that there are links from a relevant function implementation to the function calls it contains. These function calls in turn are linked to the corresponding implementations. This process is repeated so that a tree consisting of the function calls and implementations related to the original function is formed (in the described manner).

HyperSoft shows the links outgoing from a start node as arrows immediately when the mouse cursor is moved on an anchor, a special piece of program text that represents the node. These arrows can also be activated as one kind of bookmark, helping to keep the orientation. If the user decides, all the arrows can be triggered on. By this, the system forms an explicit dependency graph over the program text as shown in the figure. The arrows may exceed the size of the visible window area, in which case the user may follow them by scrolling the window.

If there is exactly one link from a start node, the destination node is reached by double-clicking the anchor of the start node. If there are several links, the alternative links are displayed in a pop-up menu. If the activated link leads into another module, the corresponding window is either opened or fetched at the foreground. For intramodular function calls the link may also be represented graphically as an arrow.

6.2.3. *Backward calling structures*

The access structure depicted in Figure 8 shows where the function *f9* can be called from (backward calling dependencies). Any relevant function implementation node is linked to the corresponding function calls. These in turn are linked to their context function. The whole process is then repeated like in the case of forward calling dependencies.

In order to support the overall comprehension of (large) programs, some special outline capabilities have been implemented. In HyperSoft, there are (hyper)text views, map views, and graphical views complementing each other, all appearing in Figure 8. A structured map view, resembling the representation style of conventional code browsers, lists in a hierarchic fashion the nodes belonging to a certain access structure so that these can be quickly checked out. Dependency levels within the map view can be opened and

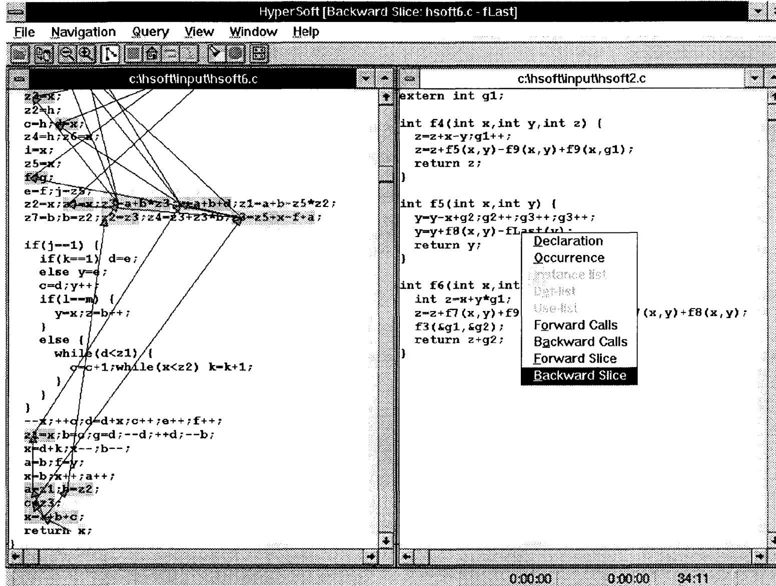


FIGURE 9. Backward slicing structure.

closed using the '+' and '–' buttons. A map view shows the related program text in a separate window as well as making it possible to directly move into any part within the access structure. Global module dependency views and global function dependency views also support program comprehension by graphically outlining the access structure on the abstract module and function level, respectively. There may be multiple outline views simultaneously opened, which makes the parallel investigation of access structures possible. Components appearing in the graphical views are linked to the original program text.

6.2.4. *Backward slices*

Figure 9 illustrates a situation in which the user needs to find out where the (erroneous) return value x of the function *fLast* has been produced. Only those statements, that might have a data-flow effect on the slicing criterion (variable x) in the return statement, are included in the access structure. Thus the user is provided with an exact view of the factors relevant to the current corrective maintenance situation.

6.2.5. *Forward slices*

Figure 10 shows two intraprocedural forward slices for the function *fLast*. On the left, the user is interested in where the value of the variable c is (indirectly) used. Slicing is started from the declaration of this variable. On the right, the user is interested in the ways a modification of the variable *z3* at the first assignment statement of it might affect the program. The arrows illustrate the progress of data flow. Because the number of links easily becomes too large and since control dependencies are in most cases not as important as data dependencies, the control dominance on data flow is not explicitly included in the access structure. The forward-slicing access structures can be used both to estimate the effects of a proposed change and to systematically reduce the resultant side effects. The figure also illustrates how multiple access structures can be compared on the screen in parallel. This is a useful feature when making major changes to the source code.

Figure 11 shows a situation in which the user is interested in the magnitude of the effects that a modification of the variable *in* in the *main* function would have to the rest of the program. The access structure expands into three program files, *hsoft1.c, hsoft2.c* and *hsoft3.c*. Intramodular interprocedural links (that is, links between two procedures within the same module) are graphically represented as arrows. The figure also shows Debug window, which is used to represent various information to the user during the access structure generations.

6.3. Implementation of HyperSoft

The main components of the HyperSoft prototype are the syntactic program analyser (SPA), the access structure generator (ASG), and the graphical user interface (GUI), as illustrated in Figure 12. These main components correspond to the syntactic structure layer, the access structure layer,

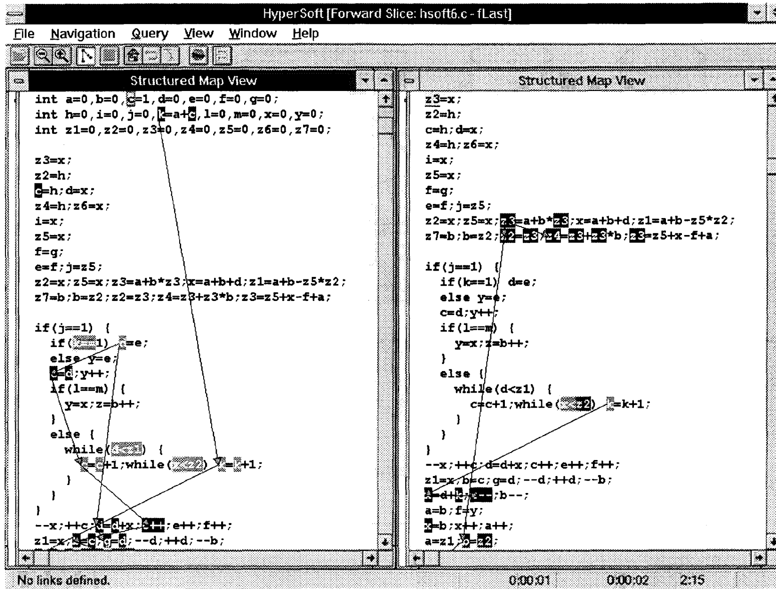


FIGURE 10. Forward slicing structures.

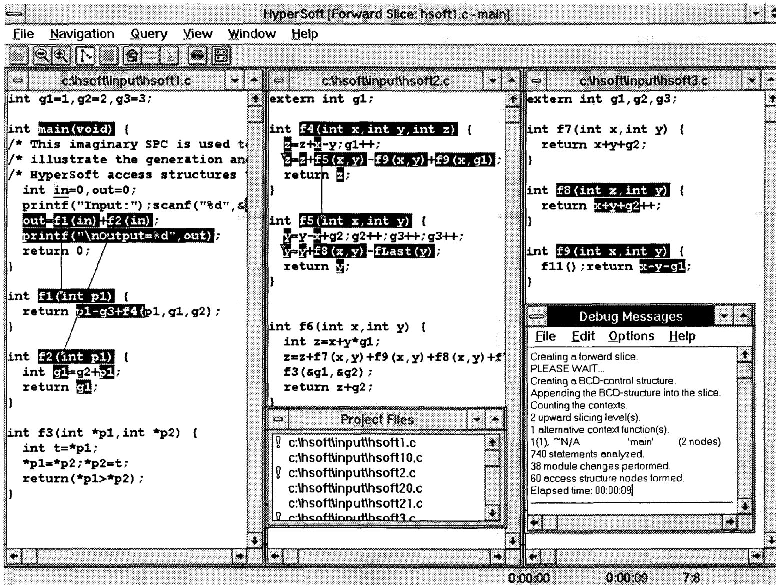


FIGURE 11. Interprocedural forward slice.

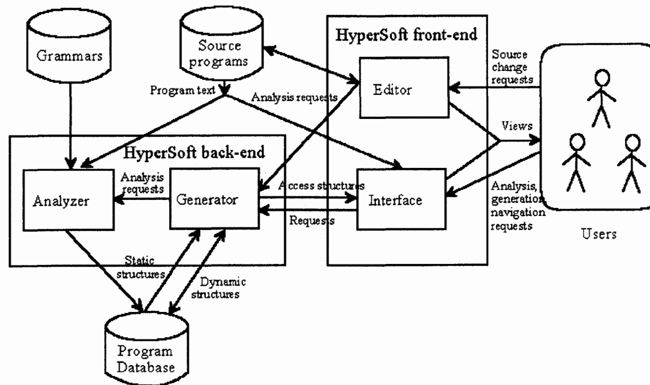


FIGURE 12. General architecture of the HyperSoft system.

TABLE 1. Required disk space and time related to the static analysis.

Process/abstraction mode	Relative tree size	Size of the program database/ source programs	Static analysis speed (LOC/s)
1. Pure parsing	N/A	N/A	490
2. Complete parse tree	1.00	50	170
3. Tree for slicing	0.25	20–25	180
4. Tree for simple access structures	0.12	8–12	90

TABLE 2. Test projects.

Project	Size (kbyte)	Static analysis time (min/s)	Static program database size (Mbyte)
1. Simple project (10 files)	6.1	0.05	0.168
2. Chess program	58.0	0.16	1.233
main.c	16.2	0.04	0.201
opening.c	5.3	0.03	0.125
dialog.c	11.4	0.04	0.203
eval.c	17.9	0.05	0.321
try.c	7.2	0.04	0.178
3. File management system	240.4	0.58	3.751
main.c	58.7	0.14	0.654
setup.c	14.0	0.09	0.377
file.c	51.3	0.14	0.714
key.c	20.7	0.06	0.283
view.c	49.1	0.14	0.695
screen.c	46.6	0.14	0.759

and the interface layer of the HyperSoft model, respectively. The implementation of the back-end components (SPA and ASG) is described in more detail in [47].

Given the subject software as input, SPA produces a program database, which is used by the ASG. GUI in turn represents the access structures created by the ASG to the user. In addition, HyperSoft is currently integrated to the programmer's file editor (PFE). The integration approach makes it possible, in principle, to use also other integrated text editors. After the user has made changes to one module through the editor, the program database is updated to reflect these changes. The prototype runs on PCs under MS Windows and supports the ANSI-C language. The objective of the first phase of the HyperSoft project (9/94–12/95) was the production of a working prototype demonstrating the ideas of transient hypertext. During the second phase (1/96–12/96) the system has been refined, optimized and extended. Our test configuration consisted of a Pentium 66 MHz, 16/428 MB micro, with MS-Windows 3.1. The SPA was compiled as a DOS-program with 32-bit Gnu-C compiler and other components with the 16-bit Borland C/C++ 4.5 compiler as a Windows program.

6.3.1. *Source code*

The original program text is fetched from mass memory for the analysis by the SPA and for the representation by the GUI. The source program collection contains the source files for which the program database has been or will be produced. Also other source modules may be viewed through GUI, but no access structure generations are allowed for them. The modules are stored on disk as conventional DOS files and the textual information is manipulated by standard C functions.

6.3.2. *Syntactic program analyser and program database*

The SPA creates the program database for the source programs that need to be (re)analysed. This is a preliminary action preceding the access structure generations. The speed of the static program analysis and program database formation is currently between 90 and 180 lines of code/s (see Table 1). Included header files are not taken into account, since currently access structures cannot be formed from them (unless they are analysed apart from their main files). For example, in case of project 2 (see Table 2) there would be about 2.7 times more code if standard header inclusions were also counted. The analysis time is also dependent on how elaborate parse tree restructurings and prunings are performed. On average approximately 3–5 times more time is needed to form the static structures than pure parsing would take. Note that the analysis times for the individual files in Table 2 are measured related to the incremental updating of the program database after changes are made to the sources.

The analyser is implemented using the AnaGram parser generator [48] employing C++ as the implementation language. Despite the term, the program 'database' is not currently implemented as a real database but instead as a collection of DOS files. The database consists of an abstract syntax tree and a local symbol table for each module. Moreover, there is a global symbol table gathering global data. An abstract syntax tree describes the syntactic hierarchic structure of the source program parts related to a module. The tree also contains the information needed to map the nodes into the symbol tables and into the original program text.

Abstract syntax trees constitute approximately 88% of the current static program database. Position, type and file information is needed for each relevant syntax tree node. Also linkage to both directions (to the child and parent nodes) has been implemented to support the creation of slicing structures. It is possible to generate variant versions of the program database depending on the need to generate various access structures. These variants and their disk space consumption are described in Table 1, related to the projects 2 (about 2,700 LOC) and 3 (about 11,000 LOC) described in Table 2. Disk space consumption varies to some extent depending on the properties (e.g. complexity, amount of comments) of the analysed programs. The variant which supports all of the currently implemented access structures takes disk space approximately 20–25 times the size of the original C sources. The ratio is of the same magnitude as, for example, in SAMS, a tool for corrective maintenance [1].

The ratio could, in principle, be further reduced by e.g. using efficient compression methods or tools, see [49]. However, the employment of the appropriate variant version of the program database is a more desirable solution because the compression and decompression would reduce the speed of access structure generations. Most of the static information is needed only when creating the slicing structures. For occurrence lists only the nodes which are of type *identifier* are relevant. The calling dependency structures require information about the function scopes, in addition to the identifier information.

6.3.3. *Access structure generator*

The access structure generator receives requests for access structure generation from the user interface, generates the structure and returns a pointer to it. In order to support incremental modifications of the source programs, we have decided to gather all of the intermodular information within the global symbol table. For efficiency reasons this table has to be kept within RAM during the session. The size of the global symbol table related to the largest test project (see Table 2, project 3) was 270 kbyte. In the worst case, this table and the local program database information (759 kbyte) of the largest module had to be kept simultaneously within RAM, occupying about 1 Mbyte. This is a quite managable outcome. Access structures as such take only a marginal portion of memory: $44 + 32\ m + 42\ n + 4\ l$ bytes, where m is the number of modules, n is the number of nodes within the access structure, and l is the average number of links.

Table 3 represents access structure generation times for the projects 2 and 3. Results for the occurrence lists and

TABLE 3. Access structure generations.

Type of the access structure	Formed nodes		Needed module swaps		Generation time (min/s)	
	Proj. 2	Proj. 3	Proj. 2	Proj. 3	Proj. 2	Proj. 3
Occurrence lists for local variables	8	12	1	1	< 0.01	< 0.01
Occurrence lists for global variables	51	24	5	3	0.06	0.10
Worst case forward call graph	280	340	63	22	0.51	1.36
Worst case backward call graph	74	189	15	40	0.16	3.00
Intraprocedural backward slices	19	4	0	0	0.01	0.02

TABLE 4. Interprocedural forward slicing—example structure generations related to project 2.

#	Upward calling levels	Analysed (total) contexts	Formed nodes	Needed module swaps	Needed time (min/s)
1	1	1(1)	609	46	1.13
2	1	1(1)	369	74	1.35
3	2	1(1)	11	74	1.13
4	7	1(254)	127	18	0.21
5	7	254(254)	127	144	5.27
6	4	2(5)	186	7	0.13
7	3	3(3)	11	5	0.05
8	3	3(3)	557	59	1.26
9	7	52(610)	193	68	1.22
10	7	610(610)	193	360	8.36

backward slices are counted as averages of 10 typical cases and for the calling structures for the largest (worst case) structures. The module swap column in Table 3 (and in Table 4) represents the number of the needed recreations of the local program database elements (abstract syntax tree and local symbol table). The creation of the occurrence lists is based on the information received from the symbol tables. During the formation of the calling structures also parts of the parse trees are traversed.

Table 4 represents 10 cases of the interprocedural forward slices formed for the project 2. The number of upward calling levels in the table represents the longest path to *main* function from the starting point of the slicing. Each function occurrence within any calling path to *main* is counted as a separate context (function). Slices are formed based on iterative solving of data-flow equations and preorder traversals of the relevant subtrees of the syntax trees. This means that all the possible function calling paths from the starting point of the slicing to the *main* function have to be checked. Therefore, it is not possible to form complete slices for realistically large programs in all cases within a reasonable time-span.

Intermodular function calls necessitate the retrieval and recreation of the local program database information (if it cannot be kept within the available RAM). Therefore, the compression of the syntax trees is useful and will directly have an effect both on space and time efficiency of the system. Slices could be formed more effectively if structures like program dependence graphs [7] would be used as a basis for them.

The user is given the option of selecting how many (ascending and descending) calling levels are taken into the slicing analysis, thus making it possible to trade off between the preciseness of the slice and fast response time. In Table 4 cases 5 and 10 represent the worst case situations, where slicing is started from the bottom of the calling hierarchy. Cases 4, 6 and 9 represent partial slices formed for the most immediate upward calling level.

HyperSoft also supports partial multiprocessing which makes it possible to, for example, browse the source code and to navigate through the already formed access structures during the formation of the new access structures. A time-estimate for the formation of the access structure is given at the beginning of its formation. The estimate can be used to decide whether it is sensible to form the structure.

The time needed to form an access structure is heavily dependent on many factors, like the size of the programs, the number of the relevant program files, the existing intermodular relations, the point from which the access structure generation has been initiated and (in the case of interprocedural slicing) on the nature of the existing calling dependencies and the use of global variables. The number of formed nodes has only a marginal effect on the time consumption.

6.3.4. *Graphical user interface*

The interface runs under MS Windows and is implemented with Borland's C++ class libraries. The interface is generic,

meaning that its operations are designed to be as independent of the target language as possible. Special attention has been paid to separate the GUI and the ASG components, so that they can be developed separately. If, for example, new access structures are implemented or the system is ported to other environments, the modifications needed on the system will be rather small.

6.4. Evaluation of the system

The first prototype version of HyperSoft was industrially evaluated during summer 1995 with small test programs; see Table 2, project 1. The information gathered during the first evaluation phase consisted mostly of free-form comments concerning the usefulness of the implemented HyperSoft capabilities and the possible ways to improve them. The gained experiences guided the further development of the system. Eight persons within the steering companies participated in the evaluation. Another session of evaluation was performed during summer 1996, containing also larger programs; see Table 2, projects 2 and 3. The results given in the table are based on using the normal variant of the program database, supporting all of the currently implemented access structures. Three professional software maintainers from the partner enterprises, selected by the steering committee, participated in this latter evaluation. The largest test project was a real file management system consisting of 6 files and 47 non-standard header files. All the projects could be processed easily within 16 Mbyte of RAM.

During the latter evaluation the information was gathered by using a questionnaire containing 22 questions. Most of the questions concerned the perceived quality of the functionalities that HyperSoft provides. Due to the small number of persons involved, the results obtained should be interpreted with some caution. The general impression of the idea of transient hypertext over programs has been positive during both evaluation periods. Especially the optional graphical visualization of hypertextual links as arrows on top of the original source code has been considered as a good idea. All the current access structures have been considered useful and the set of implemented access structures sufficient for supporting the maintenance situations that the respondents are facing. The efficiency of the access structure generations was considered adequate for the occurrence lists and for the calling structures. Complete interprocedural slicing obviously cannot be performed interactively for large programs, but the possibility of forming partial slices was still considered useful.

The amount of the formed linkage within the current access structures is quite extensive. In the case of interprocedural slices, the capability of showing the linkage only partially (related to the current node or by showing only the interprocedural links) is useful. Especially in the case of long statements the variable level linkage (as an alternative to the statement level linkage) is also justified. The formation of permanent, static linkages would not be practical except possibly for function calls. An increase in the amount of static links would make the support for incremental modifications of source files less flexible. Features that had been hoped for and were implemented during the second phase of the project included: (a) seamless integration of the SPA with the rest of the system, (b) support for on-line incremental modifications to source code, (c) specification of sets of modules (sub-projects) as basis for the access structure generation and (d) the level-wise formation of interprocedural slices (mentioned in Subsection 6.3).

7. CONCLUSIONS

We have presented a language-independent HyperSoft model of representing programs as hypertext. The model is layered according to abstracting a program into source code, syntactic structure, access structures and user interface. On one hand, the model is founded on the well-known principles of language processing, and on the other hand the general concepts of hypertext. Thus, the model effectively combines two mature areas into a flexible software engineering discipline.

We have also presented a general and language-independent model of hypertextual access structures as program dependencies. The model is based on treating program dependencies as relations between program parts. The dependencies have been classified according to their fundamental characteristics, taking into account both the relational properties of the dependencies and the way they appear in programs.

In addition to representing software abstractly in terms of hypertext, the HyperSoft model has also practical significance by suggesting how systematically to construct automated tools for software engineering. The layering of the model has been directly used as the basis for the reusable architecture of a prototype for software maintenance. By having a clean separation between the main components of the prototype we can independently implement and study different kinds of access structures and user interface solutions.

With respect to usability, the main contributions of the HyperSoft system are dynamic specification of the access structures needed by the user, automatic generation of the access structures and graphical representation of the corresponding hypertextual navigation structures on top of the original source code. We think that dynamic specification and automated generation are absolutely essential facilities when viewing (large) programs as hypertext; a statically fixed or manually produced selection of access structures would be both too restricting and too laborious for practical applications. The dynamic facilities also attack the well-known hypertext problem of cognitive overhead.

The style of representing the navigation structures at the user interface of HyperSoft is quite similar to that used in most hypertext systems, thus making it easy to adopt. Embedding the hypertextual structures directly on top of the source code makes it easy to map them together, thus following the 'direct manipulation' metaphor

of user interfaces. Notice that this solution is in contrast with conventional program browsers where the abstract (hypertextual) representation of the program is physically separated from its code. Our solution to have the whole original source code integrated with the hypertextual representation also helps to keep track of the context during program reading, thus solving the common disorientation problems of large hypertexts.

Most efficiency problems within the HyperSoft system relate to interprocedural slices, which are the most complex of the access structures. Most existing slicing tools, like those reported in [50, 51], are also research prototypes with some problems in their efficiency. In our approach, the main factor having an effect on the efficiency of generating slices (and other access structures as well) is the content of the abstract syntax trees. Possible ways to promote the slicing performance include: usage of intermediate program representations which are specially tailored for slicing [7, 8, 52], batch processing when necessary, usage of subsets of modules and usage of flexible slicing criteria.

HyperSoft applies entirely static analysis, whereas most other slicers use also dynamic run-time information of the program. The reason for our choice is that the main intended application area of HyperSoft is software maintenance where the modifications on the software must be general and not tied to any particular test case. The only notable exception is corrective maintenance where the software has to be debugged with respect to errors found in one specific execution. Notice, however, that static analysis facilitates debugging even though static analysis usually is less precise than dynamic analysis.

The kind of model and tool we have presented is probably most useful when maintaining legacy systems that are too large for being manually managed and that lack a decent documentation. Since another common feature of legacy systems is a multi-language implementation, the maintaining tool should support several languages. An explicit separation of the components in HyperSoft makes it quite straightforward to extend the system with new languages. One of the future directions of developing the HyperSoft system will be to extend its selection of source languages. At the moment, there is an on-going spin-off project to support the (embedded) database language SQL.

For languages based on different programming paradigms, the set of useful access structures may slightly vary. For instance, object-oriented programs expose dependencies not appearing in purely imperative programs. Currently, when most of the legacy software is written in languages like Cobol and C, the need for hypertextual browsing capabilities of object-oriented programs concerns the engineering of new software rather than the maintenance of old software. To solve maintenance problems of the future, the access structures supporting the hypertextual reading of object-oriented programs should be investigated more thoroughly.

ACKNOWLEDGEMENTS

The active project participation of the industrial steering group has greatly affected the interface solutions in the HyperSoft system. The user interface of HyperSoft has been implemented by Mika Nieminen. The comments of the anonymous referees have been helpful in improving the presentation of the paper.

REFERENCES

[1] Jambor-Sadeghi, K., Ketabchi, M. A., Chue, J. and Ghiassi, M. (1994) A systematic approach to corrective maintenance. *Comput. J.*, **37**, 764–778.

[2] Wilde, N., Chapman, A. and Richardson, R. (1994) The extensible dependency analysis tool set: a knowledge base for understanding industrial software. *Int. J. Software Engng Knowl. Engng*, **4**, 521–534.

[3] Harrold, M. and Soffa, M. (1990) Computation of interprocedural definition and use dependencies. In *Proc. IEEE Comput. Soc. 1990 Int. Conf. on Computer Languages*, New Orleans, LA, pp. 297–306.

[4] Ryder, B. (1979) Constructing the call graph of a program. *IEEE Trans. Software Engineering*, **5**, 216–225.

[5] Murphy, G., Notkin, D. and Lan, E. (1996) An empirical study of static call graph extractors. In: *Proc. 18th Int. Conf. on Software Engineering*, Berlin, Germany. IEEE Computer Soc. Press, pp. 90–99.

[6] Horwitz, S., Reps, T. and Binkley, D. (1990) Interprocedural slicing using dependence graphs. *ACM Trans. Progr. Langs Syst.*, **12**, 26–60.

[7] Horwitz, S. and Reps, T. (1992) The use of program dependence graphs in software engineering. In *Proc. 14th Int. Conf. on Software Engineering*, Melbourne, Australia. IEEE Computer Soc. Press, pp. 392–410.

[8] Kamkar, M. (1993) *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. Ph.D. thesis, Linköping Studies in Science and Technology Dissertations No. 297, Department of Computer and Information Science, Linköping University, Sweden.

[9] Tip, F. (1995) A survey of program slicing techniques. *J. Program. Lang.*, **3**, 121–189.

[10] Koskinen, J., Paakki, J. and Salminen, A. (1994) Program text as hypertext: using program dependences for transient linking. In *Proc. 6th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'94)*, Jurmala, Latvia. Knowledge Systems Institute, pp. 209–216.

[11] Salminen, A., Koskinen, J. and Paakki, J. (1994) HyperSoft: an environment for hypertextual software maintenance. In *Proc. Nordic Workshop on Programming Environment Research (NWPER'94)*, Lund, Sweden. Report LU-CS-TR: 94-127, Department of Computer Science, Lund University, pp. 25–37.

[12] Pressman, R. S. (1992) *Software Engineering—A Practitioner's Approach*, 3rd ed. McGraw-Hill, New York.

[13] Garg, P. K. and Scacchi, W. (1990) A hypertext system to manage software lifecycle documents. *IEEE Software*, May, 90–98.

[14] Cybulski, J. L. and Reed, K. (1992) A hypertext-based software-engineering environment. *IEEE Software*, March, 62–68.

[15] Letovsky, S. and Soloway, E. (1986) Delocalized plans and program comprehension. *IEEE Software*, May, 41–49.

[16] Brade, K., Guzdial, M., Steckel, M. and Soloway, E. (1994) Whorf: a hypertext tool for software maintenance. *Int. J. Software Engng Knowl. Engng*, **4**, 1–16.

[17] Bigelow, J. (1988) Hypertext and CASE. *IEEE Software*, March, 23–27.

[18] Cleveland, L. (1989) A program understanding support environment. *IBM Systems J.*, **28**, 324–344.

[19] Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P. and Tulula, P. (1993) CARE: an environment for understanding and re-engineering C programs. In *Proc. Conf. on Software Maintenance*, Montreal, Canada. IEEE Computer Society Press, pp. 130–139.

[20] Goldberg, A. (1984) *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA.

[21] SunPro (1992) SparcWorks (v. 2.0) Browsing Source Code. SunPro, Sun Microsystems, Inc. Business, Mountain View, CA.

[22] Lakhotia, A. (1993) Understanding someone else's code: analysis of experiences. *J. Systems Software*, **23**, 269–275.

[23] Halasz, F. and Schwartz, M. (1994) The Dexter hypertext reference model. Commun. *ACM*, **37**, 29–39.

[24] Salminen, A. and Watters, C. (1992) A two-level structure for textual databases to support hypertext access. *J. Am. Soc. Info. Sci.*, **43**, 432–447.

[25] Akscyn, R. M., McCracken, D. L. and Yoder, E. A. (1988) KMS: a distributed hypermedia system for managing knowledge in organizations. *Commun. ACM*, **31**, 820–835.

[26] Rivlin, E., Botafogo, R. and Shneiderman, B. (1994) Navigating in hyperspace: designing a structure-based toolbox. *Commun. ACM*, **37**, 87–96.

[27] Conklin, J. (1987) Hypertext: an introduction and survey. *Computer (IEEE)*, **20**, 17–41.

[28] Shneiderman, B., Shafer, P., Roland, S. and Weldon, L. (1986) Display strategies for program browsing—concepts and experiment. *IEEE Software*, May, 7–15.

[29] Goldfarb, C. F. (1990) In Rubinsky, Y. (ed.), *The SGML Handbook.* Oxford University Press, Oxford.

[30] Aho, A. V. and Ullman, J. D. (1972) *The Theory of Parsing, Translation and Compiling.* Prentice Hall, Englewood Cliffs, NJ.

[31] Salminen, A., Tague-Sutcliffe, J. and McClellan, C. (1995) From text to hypertext by indexing. *ACM Trans. Infor. Syst.*, **13**, 69–99.

[32] Watters, C. and Shepherd, M. A. (1990) A transient hypergraph-based model for data access. *ACM Trans. Infor. Syst.*, **8**, 77–102.

[33] Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers - Principles, Techniques and Tools.* Addison-Wesley, Reading, MA.

[34] Paakki, J., Koskinen, J. and Salminen, A. (1996) From relational program dependencies to hypertextual access structures. *Nordic J. Comput.*, submitted.

[35] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ.

[36] Baker, B. S. (1993) A theory of parameterized pattern matching: algorithms and applications. In *Proc. 25th ACM Symp. on Theory of Computing*, San Diego, CA. ACM Press, pp. 71–80.

[37] Yang, W. (1991) Identifying syntactic differences between two programs. *Software Pract. Exper.*, **21**, 739–755.

[38] Kilpeläinen, P. and Mannila, H. (1994) Query primitives for tree-structured data. In *Proc. 5th Annual Symp. on Combinatorial Pattern Matching (CPM'94)*, Asilomar, CA. LNCS 807, Springer-Verlag, pp. 213–225.

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

[40] McCabe, T. J. (1976) A complexity measure. *IEEE Trans. Software Engng*, **2**, 308–320.

[41] Gibbs, S., Tsichritzis, D., Casais, E., Nierstrasz, O. and Pintado, X. (1990) Class management for software communities. *Commun. ACM*, **33**, 90–103.

[42] Taenzer, D., Ganti, M. and Podar, S. (1989) Object-oriented software reuse: the yoyo problem. *J. Object-Oriented Program.*, **2**, 30–35.

[43] Moriconi, M. and Hare, D. F. (1986) The PegaSys system: pictures as formal documentation of large programs. *ACM Trans. Program. Lang. Syst.*, **8**, 524–546.

[44] Kernighan, B. and Ritchie, D. (1988) *The C Programming Language*, 2nd ed. Prentice Hall, Englewood Cliffs, NJ.

[45] Weiser, M. (1984) Program slicing. *IEEE Trans. Software Engng*, **10**, 352–357.

[46] Bergeretti, J. -F. and Carre, B. (1985) Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Systems*, **7**, 37–61.

[47] Koskinen, J. (1996) Creating transient hypertextual access structures for C programs. In *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*, Herzliya, Israel. IEEE Computer Society Press, pp. 56–65.

[48] Parsifal Software (1993) *AnaGramTM—User's Guide*. Parsifal Software, Wayland, MA.

[49] Katajainen, J. and Mäkinen, E. (1990) Tree compression and optimization with applications. *Int. J. Foundations Comput. Sci.*, **1**, 425–447.

[50] Samadzadeh, M. and Wichaipanitch, W. (1993) An interactive debugging tool for C based on dynamic slicing and dicing. In *Proc. 21st Annual Computer Science Conf.*, Indianapolis, IN. ACM Press, pp. 30–37.

[51] Hoffner, T., Kamkar, M. and Fritzson, P. (1995) Evaluation of program slicing tools. In *Proc. 2nd Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, St Malo, France. IRISA-CNRS.

[52] Binkley, D. W. and Gallagher, K. B. (1996) Program slicing. In *Adv. Comput.*, **43**, 1–50.

V

FROM RELATIONAL PROGRAM DEPENDENCIES TO HYPERTEXTUAL ACCESS STRUCTURES

Paakki, J., Koskinen, J. & Salminen, A. 1997.
Nordic Journal of Computing 4 (1), 3-36.

Reproduced with permission.

Nordic Journal of Computing 4(1997), 3–36.

FROM RELATIONAL PROGRAM DEPENDENCIES TO HYPERTEXTUAL ACCESS STRUCTURES

JUKKA PAAKKI
Department of Computer Science
P.O. Box 26, FIN - 00014 University of Helsinki, Finland
paakki@cs.helsinki.fi

JUSSI KOSKINEN   AIRI SALMINEN

Department of Computer Science and Information Systems
University of Jyväskylä
P.O. Box 35, FIN - 40351 Jyväskylä, Finland
{koskinen,airi}@cs.jyu.fi

Abstract. Several important aspects of software systems can be expressed as dependencies between their components. A special class of dependencies concentrates on the program text and captures the technical structure and behavior of the target system. The central characteristic making such program dependencies valuable in software engineering environments is that they can be automatically extracted from the program by applying well-known methods of programming language implementation. We present a model of program dependencies by considering them as relations between program elements. Moreover, we show how dependency relations form the basis of producing a graph-like hypertextual representation of programs for a programming environment. Having a general and well-defined model of program dependencies as a foundation makes it easier to systematically construct and integrate language-based tools. As an example application, we present a hypertextual tool which is founded on our relational dependency model and which can be used to maintain programs written in the programming language C.

CR Classification: G.2.2, D.2.2, D.2.5, H.5.1

Key words: program dependencies, hypertext, software maintenance, reverse engineering

1. Introduction

Modern software systems are large and complicated. A software-engineering lifecycle is typically confronted with the problems of team work, heterogeneous operating platforms, constantly changing requirements, and a long chain of different development phases. To completely cover a system for further advancement and maintenance, its documentation should record the whole development history as well as all the technical design and implementation decisions. Managing such a huge amount of dispersed, yet interrelated information is impossible in practice without automated means.

Received August 18, 1996; accepted December 5, 1996.

While documentation is most essential for both using and modifying a software system, an unfortunate fact is that a complete and up-to-date documentation is usually not available. This is most common for *legacy systems* [23] having a long history with numerous software releases on different hardware and software architectures, alternating design and implementation styles, and various development and maintenance teams. For such systems, the only reliable documentation is the source code, which often makes it necessary to apply special *reverse-engineering* or *re-engineering* techniques [1] for obtaining, e.g., an approximated system design to support further maintenance. Notice that managing the internals of source code is useful not only for legacy systems but for any software system that is subject to repeated modifications; after all, the final target of software modification is always the source code.

Re(verse)-engineering and other program analysis techniques are based on finding the relevant code fragments and their relationships. Being formally defined artifacts, these *program dependencies* can be *automatically* extracted from the software which is a significant advantage over documentation-level software representations that usually must be produced manually. Program dependencies are relationships holding between the parts of programs and can be determined from a program's text, see for example [38]. Well-known and widely used examples of program dependencies are the dependencies included into the definition-use chains and call graphs. Many of such dependencies have originally been developed for the needs of compiler construction (see e.g., [4]) from where they have been adopted into more general software engineering tools, such as debuggers and maintenance assistants. Having a unified software representation in terms of fine-grained program components and their interdependencies makes it also easier to integrate the different language tools into a seamless environment for collaborative (or "concurrent") software development (see, e.g., [13] and [30]).

One technical solution suggested for managing the external and internal documentation of software systems is *hypertext*, or more generally, *hypermedia* (e.g., [14], [3], [17], [15]). Hypertext and hypermedia make it possible to link together pieces of related information and to navigate in the resulting information space. A versatile hypertextual software engineering environment typically connects together items in software requirements, designs, source code, test material, etc., making it easier to map a certain element to other elements somehow related to it. For instance, there might be a hypertext link from a module in the source code to other modules using it, to a chapter in the requirements document describing the module's external functionality, to a chapter in the design document defining the module's internal structure, and to the test document describing how the module has been tested. All this information related to the module is useful when, for some reason, modifying it.

HyperSoft is an automated tool supporting the understanding and maintenance of software. HyperSoft is based on representing the internal source code of a software system as hypertext, concentrating on program dependen-

cies that are useful in typical maintenance tasks. Instead of ad hoc solutions, HyperSoft provides the hypertextual features in terms of a clear model that makes a separation between the program, its internal dependencies, and the user interface. The organization of the system follows the model and makes it possible to flexibly explore different kinds of solutions by modifying just one component of the system. The HyperSoft method and the possibilities of using program dependencies in software maintenance were originally introduced in [27], and the technical architecture of the HyperSoft system was presented in [44]. Notice that HyperSoft concentrates solely on the program code and does not provide links to any external documentation.

In this paper we study the nature of program dependencies. In order to get a solid framework for (hypertextual) programming environments, we have classified the potential dependencies in terms of their *relational characteristics*. Hence, in our model of software a program dependency is considered as a binary relation between two program components. The model is general and not tied to the special solution in HyperSoft which is merely introduced as a constructive example of applying the approach in practice.

Our relational model provides a systematic base for selecting and implementing the program dependencies in software tools. By precisely stating the inherent characteristics of the various dependencies, the model also makes it easier to design the external representation of the dependencies in a graphical user interface. Finally, the model makes it possible to analyze programming environments with respect to how extensive and coherent support they provide for program dependencies.

The paper is organized as follows. First, the connection between programs and hypertext is introduced in Section 2 and the relational properties of program dependencies in Section 3. Then, a relational classification of program dependencies is presented in Section 4. Section 5 discusses how program dependencies can be joined into compound dependencies and evolved into hypertextual access structures. Section 6 addresses the related and useful graph theoretical concepts. Section 7 introduces HyperSoft as a tool based on the dependency model. Finally, the paper is concluded and related works are discussed in Section 8.

2. Programs and hypertext

Software engineers must read programs in connection with many daily duties, such as maintenance, debugging, code reviews, and testing. As is well known [43, 9], the task of understanding the meaning of a program is mentally quite demanding due to acquiring large amounts of domain-specific knowledge. The problem is made even harder by the fact that usually the knowledge in demand is not physically centralized in one module but spread throughout the program as a “delocalized plan” [29].

The technical means attacking the program comprehension problem can be roughly divided into two categories: One can apply the techniques of

artificial intelligence for finding semantic *chunks* or algorithmic *clichés* from the program (e.g., [51]), or one can use compiler-oriented analysis algorithms for constructing a characteristic dependency *graph* over the program (e.g., [21]). Such higher-level program representations reduce the engineer's search space by hiding the uninteresting details and thus make it easier for the engineer to concentrate on the relevant aspects only.

Whatever the technique, the essence in constructing a high-level program representation is to extract the useful pieces of the program and to find out how they are related. When expressing relations over textual information (such as program code), a convenient and popular technique is hypertext. Hypertext integrates text with nonlinear navigation capabilities, making it possible to study the program fragments in an arbitrary order by following links between them. Thus, the central idea of hypertext conforms well with the dispersed nature of delocalized plans, chunks, clichés, and dependency graphs.

Hypertext is often modeled as a directed graph, that is, as a pair (N, L) where N is a set of *nodes* and L is a set of ordered node pairs called *links*. We will here use the terminology that we have used in our previous papers [42, 27, 44, 26]. The links as well as the nodes may be associated with a label. The nodes of a link are called the *start node* and the *destination node*. Having graphs as a mathematical model in the background makes it possible to apply the advanced concepts of graph theory on hypertext, as will be demonstrated in this paper. In our approach a hypertextual representation for a program is a directed graph constructed in terms of one or more program dependencies. The graph is called an *access structure*. Some node of the access structure (usually the one which is formed first) may have special status, and is called the *root* (or *home*) *node*.

The central problem when creating hypertext is how to select the nodes and the links to properly support the application. While being quite hard for arbitrary text, the problem can be tackled in software engineering by utilizing the special characteristics of *program text*: Most notably, each program has a (context-free) *grammar* which formally defines its structure as a (parse) *tree* [2]. The parse tree contains all the elements of the program hierarchically organized, thus providing a natural basis for inducing hypertextual access structures for the program. [1]

Our technique of transforming a program into an access structure is based on the context-free grammar for the programming language and on the parse tree for the program. Each nonterminal of the grammar represents a set of textual entities (its instances) in the program. Therefore, in our terminology a nonterminal in the grammar is called a (text or program) *type* and each of its instances in the program is called a (text or program) *part*. In a parse tree, each node for a text part is labelled by the corresponding text type

[1] Notice that even ordinary text may in some special cases have a formal structure enhancing its automatic processability. For example, all SGML documents are defined by a context-free grammar [18].

(nonterminal symbol), and the leaves in the subtree for the node, concatenated from left to right, form the textual representation of the part. This representation is called the *value* of the part. Notice that since a context-free grammar typically includes a number of unit productions of the form $A \rightarrow B$ where A and B are nonterminals, a value may stand for several text types (here A and B). In order to make the part - value mapping unique, a node in the parse tree is considered a part only if it is not the single child of its parent. For the same reason, and for being able to visualize each part to the user as a character string, we restrict our grammars such that the production of empty strings is not allowed. This formulation of grammar-based text follows the model originally presented in [42].

An access structure over a program text shows dependencies between program parts. The dependencies are binary relations among parts and may thus be expressed as hypertextual links. Each access structure node stands for a program component which is represented: (1) as one or more nonterminal symbols in the grammar for the programming language and (2) as a piece of code in the program. Accordingly, a node has both (1) one or more types and (2) a value. In the following we will use the terms "node" and "part" interchangebly.

As an example, Fig. 1 depicts a simple access structure over a C program. Nodes are indicated as frames around their value and links are indicated as directed arrows.

Suppose that the program in Fig. 1 conforms to the following context-free grammar. Nonterminal symbols are given as alphabetic names, terminal symbols are embedded within a pair of apostrophes ('), → separates the left-hand side of a production from its right-hand side, and alternatives are separated by the symbol |. Only the relevant productions are given.

```
program → translation-unit
translation-unit external-declaration |
    translation-unit external-declaration
external-declaration → declaration | function-definition
declaration → init-declaration | ...
init-declaration → type-specifier identifier '='
    initial-value ';'
type-specifier → 'int' | 'void' | ...
identifier → ...
initial-value → ...
function-definition → type-specifier identifier
    compound-statement
compound-statement → '{' declaration-list statement-list '}'
declaration-list → declaration | declaration-list declaration
statement-list → statement | statement-list statement
statement → assignment ';' | ...
assignment → identifier '=' expression
expression → ...
```

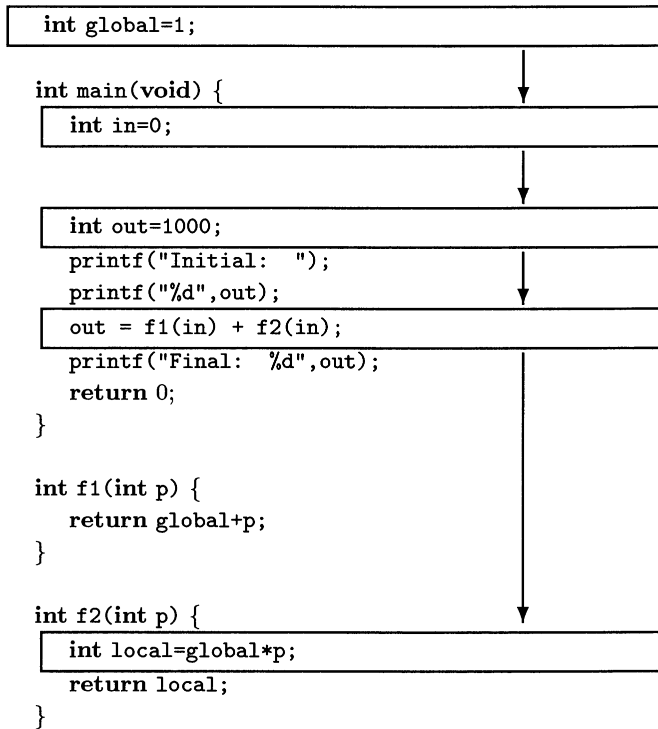


Fig. 1: A sample access structure.

Now the parts in the access structure in Fig. 1 are of type *init-declaration* or *assignment*, linked to each other according to their preorder in the parse tree. Hence, the access structure captures all those parts of the program where a variable can be given a value. In this case the parts can be automatically found from the underlying parse tree by simply consulting the label of its nodes, but for more advanced cases the process is more complicated. In general, the relevant nodes are found by applying a part-selection algorithm during a traversal over the parse tree.

This access structure might be created roughly in the following steps. A formal account will be given in Section 5.

(1) Construct the *init-declaration* relation by finding all the parts of type *init-declaration* in the underlying parse tree.

(2) Construct the *assignment* relation by finding all the parts of type *assignment* in the parse tree.

(3) Form the union of the *init-declaration* and *assignment* relations.

(4) Induce a *preorder* relation over the parts in *init-declaration* ∪ *assignment*.

(5) Develop the *preorder* relation into a directed graph by taking the parts as nodes and the binary relations between the parts as links.

3. Relational properties of program dependencies

In this section we introduce the common relational properties of program dependencies. These properties have influence on the way that program text should be represented as hypertext and they can be used as a basis to classify the program dependencies as is done in the following section. Their main implications to the formation of the hypertextual access structures are represented here and are explored in more detail in the following Sections 4-7.

Program dependencies can be characterized in terms of (P1) *reflexivity,* (P2) *symmetry,* (P3) *transitivity,* (P4) *irreflexivity,* (P5) *antisymmetry,* and (P6) *intransitivity,* which are defined as follows. R denotes a dependency relation in a set S of parts, and x, y, and z are parts in the set S.

$$
\begin{array}{ll}
(P1) & \forall\, x \in S : x\ R\ x; \\
(P2) & \forall\, x, y \in S : x\ R\ y \Rightarrow y\ R\ x; \\
(P3) & \forall\, x, y, z \in S : (x\ R\ y) \wedge (y\ R\ z) \Rightarrow (x\ R\ z); \\
(P4) & \forall\, x \in S : \neg\,(x\ R\ x); \\
(P5) & \forall\, x, y \in S : (x\ R\ y) \wedge (y\ R\ x) \Rightarrow (x = y); \\
(P6) & \forall\, x, y, z \in S : (x\ R\ y) \wedge (y\ R\ z) \Rightarrow \neg\,(x\ R\ z).
\end{array}
$$

As described in Section 2, hypertextual access structures can be formed rather straightforwardly by bounding together program parts (and the corresponding program text) based on various program dependencies existing between them. In order to avoid navigational problems due to exceedingly dense or uninformative linkage, some linking conventions, however, need to be applied. The following conventions can be applied and are applied, for example, in HyperSoft:

- Reflexivity. In hypertext, links are used by a reader to move from a currently visible node to another node. A link from a node to itself is not useful in hypertext. Thus indication of reflexive relations $x\ R\ x$ by hypertextual links is not needed.
- Symmetry. In hypertext two nodes may be connected by bidirectional links to allow return to previous node. On the other hand, most hypertextual systems offer automatically a backtracking capability in the user interface. Thus symmetric relationships may be represented with uni-directional links. Navigation to reverse direction (backtracking) becomes possible after a link has been used to move to its destination node.
- Transitivity. In transitive relations, we can separate the direct and indirect dependencies. Indicating all of them by links easily leads to an exceedingly dense linkage. Restricting the linkage to the direct dependencies simplifies the hypertextual structure. Then only the reachability of a node from another node by link traversal indicates the (indirect) dependency between the nodes.

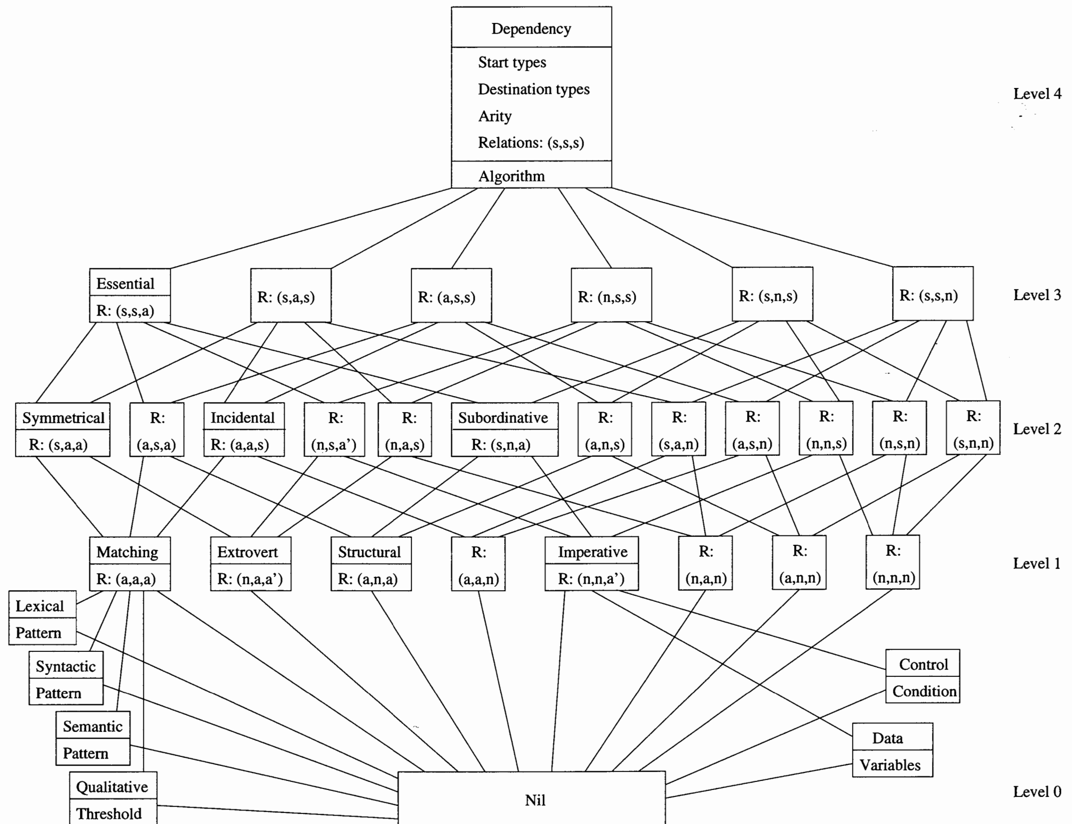


Fig. 2: Classification of program dependencies.

4. Relational classification of program dependencies

Our classification of program dependencies is based on considering them as relations between program parts. Recall from Section 2 that each part stands for a specific element in the program text and for a node in the parse tree of the program. The classification is shown in Fig. 2 as a lattice. It is represented in order to capture the essence of program dependencies and to establish a basis for systematically discussing the major dependencies and the possibilities to consider them as a foundation of forming hypertextual access structures over the program text.

In addition to classifying the concrete elementary program dependencies, the lattice also supports the classification of compound dependencies (described in Section 5) and the corresponding hypertextual access structures. Elementary program dependencies are listed in [47], [11], [28], and [38]. The sensibility of the combined structures can be evaluated based on the types of the elementary dependencies. The classification could also be used to analyze programming environments based on the categories of program dependencies that are included. It also enhances the possibilities to represent

profoundly similar program constructs in a consistent way at the interface level of the support environments.

As usual in modelling, our classification is abstract and general in the sense that it does not directly express the concrete particular program dependencies but instead their common characteristics as *dependency categories*. Hence, in object-oriented terminology, a category corresponds to a *class*, while each actual program dependency corresponds to an *object* (an instance of its category). The central properties of a program dependency are specified by its category, and each category typically includes several program dependencies with similar properties. General (super)categories are refined into subcategories on several levels according to certain criteria, usually manifesting the relational properties described in the previous section (the most important dependency subcategories are further specialized based on other criteria also). Lower level subcategories inherit the properties of the supercategories. The properties can, however, also be overridden (redefined) in the subcategories when necessary. Since the classification hierarchy is a lattice, multiple inheritance is the common case.

Because our classification is abstract and total, all the represented dependency categories are not necessarily (at least currently) of practical importance. The most important categories are named in the figure. Most practical program dependencies belong to the leaf categories of the classification (Level 1), but even the higher-level categories hold some dependencies, which are commonly cited in the programming literature and provided in software tools. In this section we concentrate on the explicitly named dependency categories.

The categories hold *properties*, that is, *attributes* and *methods*. In the following we will describe the properties specified for the top category – *Dependency*. Since we are dealing with pairs of program parts, the dependencies are considered as binary relations. Since a (binary) dependency is expressed in our model as a hypertextual link, each dependency involves a start node and a destination node. For different kinds of dependencies the nodes may be of different text type, and a certain dependency may involve several text types. This is modelled in the classification by the properties *Start types* and *Destination types*, respectively. Programming environments (related to procedural languages) most typically support the investigation of program dependencies between components like: files, functions, variables, types, and macros [11], which typically have a corresponding syntactical type in the grammar of the language.

The arity (cardinality) of a dependency is represented in the model by the property *Arity*. In general, a relation $a\ R\ b$ is either of type *1-to-1*, *1-to-n*, *m-to-1* or *m-to-n*, depending on the possible arity of a and b. In this context an "arity" means how many different parts for a and b there are such that $a\ R\ b$ holds. For example, a dependency R is of arity *1-to-1* if there is a single part a and a single part b in $a\ R\ b$, and R is of arity *1-to-n* if a is unique but there may be several different b's in $a\ R\ b$.

The attribute $Relations(degree_{reflexivity}, degree_{symmetry}, degree_{transitivity})$ specifies whether all (a), some (s) or none (n) of the dependencies within a certain category are reflexive, symmetric, and transitive, respectively. For example the value of the attribute *R: (s,n,a)* of the category *Subordinative* denotes that some of the program dependencies belonging to that category are reflexive (and some are not), none of the dependencies are symmetric (i.e. all of them are antisymmetric) and all the dependencies are transitive. The value of the *Relations* attribute is overridden in each more specific subcategory. The top category of the hierarchy, at Level 4, is the most indetermined case where all the relational properties and their opposites hold for some concrete dependencies. At Level 3 only two properties may vary freely and at Level 2 only one property. At Level 1 (the leaf category level) each of the three relational properties hold either for all or for none of the dependencies within a category. The purpose of the bottom category *Nil* is explained in Section 5 (definition 5.2).

Each dependency involves a specific algorithm to find all the particular program parts. This is modelled by the method *Algorithm*, associated with the top category and inherited to all the descendant categories. Let $a\ R\ b$ denote that parts a and b are in the dependency relation R. Then all such pairs (a, b) may be retrieved from the program by applying the *Algorithm* of the dependency instance R.

4.1 Essential dependencies, R: (s,s,a)

Most of the common, useful program dependencies belong to some of the subcategories of the abstract category named *Essential.* All the program dependencies within this category (and within its descendants) follow transitivity. While the *Essential* category itself is rather general by just requiring transitivity, it still captures some frequently used program dependencies and structures representing these dependencies. Most notably, the general form of well-known *call graphs* can be allocated in this category: A conventional call graph for a program contains its subroutines (procedures and functions) as nodes and the calling relationships as links. In other words, there is a link from subroutine a to subroutine b if a calls b (directly) in the program. In our model, this is represented by a binary relation a *Calls* b. The relation is transitive since a *Calls* b and b *Calls* c means that the subroutine a calls the subroutine c indirectly via b.

Besides transitivity, such a notion of call graph does not have the other relational properties in a general case: for recursive subroutines (a calls itself) reflexivity holds (a *Calls* a), whereas for non-recursive subroutines it does not hold. For subroutines mutually calling each other, symmetry holds (a *Calls* b, b *Calls* a), whereas it does not hold for other pairs of subroutines. Since all these different cases can appear in the same program, the total dependency is neither (ir)reflexive nor (anti)symmetric. Since, for example, in Fortran, recursion is not allowed, the call graph of a Fortran program could be placed (as a more specific case) into the category *Imperative*,

R: (n,n,a') (the transitivity property *a'* will be explained in Section 4.5). The call graph of a program written in a (hypothetical) programming language where direct recursion (causing a reflexive relation) is forbidden but indirect recursion allowed would consequently belong to the category *R: (n,s,a')*.

A "call graph" is actually quite overloaded as a term and appears in a surprisingly large number of different forms in software tools [32]. These alternative forms typically aggregate more information about subroutines than just the calling relationship between subroutine names, for instance parameters and their data flow. In HyperSoft the access structures for call graphs link together subroutine calls and definitions, and actually combine two different dependencies. As a result, the calling access structures in HyperSoft belong to the category *Subordinative, R: (s,n,a)* rather than to *Essential* (see Section 7).

4.2 Incidental dependencies, R: (a,a,s)

While most practical program dependencies are transitive, this is not always the case. Consider for instance the relation *Shares*, where *a Shares b* holds whenever the subroutines *a* and *b* use a common global variable. Now it might be that *a Shares b* because both *a* and *b* use the variable *x*, while *b Shares c* because both *b* and *c* use the variable *y*. But then (assuming that *a* and *c* do not use other global variables) *a Shares c* does not hold, and the relation is not transitive. In the general case total intransitivity does not hold either, because three other subroutines of the same program, say *d*, *e*, and *f*, might well use the same variable *z*.

Such dependencies without definite (in)transitive properties are captured in the category *Incidental*. The dependencies in this category are reflexive and symmetric, and can therefore be characterized as *compatibility relations*. A consequence of nontransitivity is that an *Incidental* dependency classifies the program parts into nondisjoint compatibility classes that can be provided special support in a (hypertextual) programming environment. For instance, the most tightly coupled block of subroutines could be handed over to a software engineer by generating a compatibility classification over the *Shares* dependency, and by extracting the class with the largest number of elements.

4.3 Symmetrical dependencies, R: (s,a,a)

Intuitively, a relation *R* belongs to the category *Symmetrical* if *a R b* implies that the parts *a* and *b* are somehow similar in their syntax or semantics. The *Symmetrical* category is abstract in the sense that its role is just to express the general symmetry of dependencies actually falling into categories lower in the hierarchy. The most prominent descendant category of *Symmetrical* is *Matching, R: (a,a,a)*.

4.4 Matching dependencies, R: (a,a,a)

The dependencies in the *Matching* category are reflexive, symmetric, and transitive, and thus correspond to the special class of *equivalence relations*. The *Matching* category is further specialized into the subcategories *Lexical*, *Syntactic*, *Semantic*, and *Qualitative*. For a *Matching* dependency $a\ R\ b$, the similarity or equivalence of a and b can be induced either on lexical, syntactic, or semantic basis. Intuitively, this division corresponds to classifying the programming language issues into "lexical", "syntactic" and "semantic" ones, each involving specific definition and implementation techniques of different power.

If R is *Lexical*, then a and b have a similar textual representation; in other words, a and b can be defined with the same regular *Pattern* expression over characters. The involved *Dependency* method *Algorithm* then finds all those parts from the program that are textually similar, as proposed e.g. in [6]. For instance, a *Lexical* relation might include all the occurrences of names starting with the string "foo":

```
'f' 'o' 'o' any*
```

where `any` stands for any letter or digit. Then the relation would include, among others, the name-parts with value `foo`, `foolish`, and `food2much`.

The property *Pattern* for a relation R in the subcategory *Syntactic* specifies the syntactic shape of the parts. In other words, $a\ R\ b$ implies that the syntax of parts a and b is defined with the same context-free grammar, and that the subtrees for a and b are isomorphic in a parse tree. The method *Algorithm* for a *Syntactic* category typically applies pattern matching in (parse) trees, as suggested e.g. in [52] and in [24]. For instance, a *Syntactic* program dependency might extract from the parse tree all the parts of the same program type, as was illustrated in Fig. 1.

The context-free grammar of the programming language can also be extended to hold information about the more stylistic aspects of the program, typical examples being layout (indentation, empty lines, etc.) and comments. These kind of components cannot be executed and are "white space" in the dynamic respect. However, for user-centered software tools the layout and comments are absolutely necessary and must be included in the internal representation of the program (for special techniques, see e.g. [10]). We therefore assume that the context-free grammar of the language and the underlying parse tree of the program include even these kind of "stylistic" components, which can be managed in the same manner as the more ordinary ones.

The parts in a *Semantic* relation express the same computation or a similar behavior when running the program. Now the property *Pattern* may stand e.g. for a data-flow computation [21], a specific algorithm [40], or fuzzy functionality [16]. For instance, a *Semantic* relation might capture all the different sorting algorithms or all the non-reachable regions ("dead code") in the program or the different ways, aliases, that are used to refer to

a specific memory location, thus inducing a *SharesMemory* relation. Since semantic issues of programming languages are much more complex than lexical and syntactic ones, this category calls for powerful definition formalisms for its properties. The *Semantic Patterns* and *Algorithms* may be founded on purely static techniques with no actual execution of the program (such as attribute grammars [25]) or on more dynamic ones with an approximated execution (such as abstract interpretation [12]).

Two parts are in a *Qualitative* relation if they have the same *Threshold* value for some quality factor of software engineering, such as testability, maintainability, understandability, or portability. *Cohesion* and *coupling*, for example, are two well-known approximations for the quality of modules [39], and *cyclomatic complexity* [31] is a standard metric for the computational and mental complexity of programs. For instance, a *Qualitative* dependency might hold between subprograms that are too complex for a sound maintenance, being at the *Threshold* level 10 or higher in their cyclomatic complexity. Notice that the analysis of software quality can be based either on lexical, syntactic or semantic properties. For instance, cyclomatic complexity of loops can be roughly approximated by counting their nesting level on purely syntactic basis, while a more precise measure can be obtained by computing on semantic basis even the number of loop iterations.

4.5 Extrovert dependencies, R: (n,a,a')

The *Extrovert* program dependencies are symmetric, transitive, and irreflexive. In terms of the relational properties (*P1*) - (*P6*) such a combination is quite peculiar. Suppose that R is an *Extrovert* dependency such that $a\ R\ b$ holds ($a \neq b$ since R is irreflexive). Symmetry implies that $b\ R\ a$ holds as well and transitivity yields $(a\ R\ b) \wedge (b\ R\ a) \Rightarrow (a\ R\ a)$, which would be a contradiction with respect to irreflexivity. In order to make the relational characterization of *Extrovert* sensible, we modify the requirement of transitivity as follows:

$$(P3') \qquad \forall\, x, y, z \in S, x \neq z : (x\ R\ y) \wedge (y\ R\ z) \Rightarrow (x\ R\ z).$$

This kind of modified transitivity is indicated by the *a'* in Fig. 2 and is used also in the category *Imperative* (Section 4.8) as well as in the common superclass of these two categories. Even after this modification the *Extrovert* category is rather esoteric and contains program dependencies of marginal interest only. An example is the rendezvous synchronization mechanism of concurrent processes, as applied, e.g., on Ada tasks: a *Rendezvous* b holds if the task-parts a and b are synchronized by a call-accept pair of statements. The dependency is clearly symmetric, and also transitive since b may be further synchronized with a task c during its rendezvous with task a, making both b *Rendezvous* c and a *Rendezvous* c hold. For precluding deadlock, a task must not synchronize with itself, making the *Rendezvous* dependency

irreflexive.[2] Another example is the sibling relationship between classes in an object-oriented hierarchy: Two classes are siblings if they are different subclasses of the same superclass. For instance, *Matching* and *Extrovert* are siblings in the category hierarchy of Fig. 2 by having *Symmetrical* as their common supercategory.

4.6 Subordinative dependencies, R: (s,n,a)

For *Subordinative* relations, *a R b* implies that part *a* somehow dominates or has control over part *b*. Since one part dominates the other, these relations are antisymmetric. For instance, part *a* may contain part *b* as one of its components (in the case of structured types), or *a* might be a superclass of *b* (in the case of classes of an object-oriented program), or the expression-part *a* might control the execution of the statement-part *b* (in the case of conditional statements). Also, the general *IsUsedToCompute* relation belongs to this category, since the relation can be either irreflexive, as in the case of `x=y;` or reflexive as in the case of `x=x+y;` The *IsUsedToConstruct* relation holds between a definition and another structure that uses that definition while defining itself. For example in C: `typedef TYPE1 int;` defines that `TYPE1` corresponds to the integer type. This definition can further be used e.g. in: `typedef TYPE2 TYPE1.` So, here exists the following relations: `int` *IsUsedToConstruct* `TYPE1` and `TYPE1` *IsUsedToConstruct* `TYPE2.`

Because a *Subordinative* dependency is antisymmetric, it always has a meaningful *inverse dependency* relation that can be obtained in a straightforward manner from the original one. For instance, the inverse of *a IsSuperclassOf b* is *b IsSubclassOf a* (for object-oriented class hierarchies), and the inverse of *a Controls b* is *b IsControlledBy a* (for general domination). The inversion property of *Subordinative* dependencies is utilized in the HyperSoft tool for some central access structures (see Section 7). Notice that the inverse of a dependency with *Arity 1-to-n* belongs to an *m-to-1* category and, accordingly, the inverse of an *m-to-1* dependency belongs to a *1-to-n* category. The inverse of a *1-to-1* dependency is also of *Arity 1-to-1*, and the inverse of an *m-to-n* dependency remains in the *Arity* class *m-to-n*.

Subordinative is specialized into two subcategories, *Structural* (reflexive) and *Imperative* (irreflexive) which will be discussed in the following Sections 4.7 and 4.8. Most *Subordinative* dependencies fall into the specialized subcategories, but in some cases the general category has to be applied. Consider, for instance, the modelling of unconditional jumps: *a GoesTo b* holds if *a* is a "goto" statement whose target is the statement *b*. Now *GoesTo* is clearly transitive and antisymmetric. In most cases the relation is irreflexive as well but in the following (rather anomalous) case reflexivity holds instead, making *GoesTo* thus belong to the general *Subordinative* category:

```
Loop:  goto Loop;
```

[2] Notice, however, that (reflexive) deadlocking may not be explicitly prohibited in the definition of a programming language (such as Ada).

4.7 *Structural dependencies, R (a,n,a)*

The dependency category *Subordinative* is refined into *Structural* (reflexive) and *Imperative* (irreflexive). Since the *Structural* program dependencies are transitive, antisymmetric, and reflexive, each of them induces a *weak partial order* over its parts which form a *partially ordered set.* Partial orders are particularly useful for modelling data types with a number of intuitive representations.

For example, the *IsSuperclassOf* dependency mentioned in Section 4.6, mapping a class with its subclasses in an object-oriented program, is of category *Structural* with *Arity 1-to-n* in the case of single inheritance. Therefore the inverse dependency, *IsSubclassOf*, is *Structural* with *Arity m-to-1.* In the case of multiple inheritance both the dependency *IsSuperlassOf* and its inverse *IsSubclassOf* are of *Arity m-to-n.* The arity of these program dependencies directs their (graphical) representation: for single inheritance the natural choice is to illustrate a class hierarchy as a tree, whereas for multiple inheritance a general lattice has to be used [19].

Another example of a *Structural* dependency is the *Includes* relation, i.e. the one between a file and an included (header) file. Depending on the exact definition of the relations *IsSuperClassOf, IsSubClassOf,* and *Includes* they could also be considered as being irreflexive or nonreflexive. Note that these different interpretations could easily be taken into consideration by placing the relations in the categories *Imperative, R: (n,n,a')* or *Subordinative, R: (s,n,a),* respectively. Note also as a curiosity, that the relation *IsDirectSuperClassOf* would fall into the (most esoteric) category *R: (n,n,n).*

4.8 *Imperative dependencies, R (n,n,a')*

When a program is executed, it basically involves merely control (determining the applied statements and their relative invoking order) and data (capturing the processed values). In our model, these fundamental concepts appear as the *Imperative* subcategories *Control* and *Data*, respectively. Since these dependencies involve a directed flow of control and data, they are irreflexive by nature, defined by their common supercategory. Being irreflexive, antisymmetric, and transitive, the *Imperative* program dependencies induce a *strict partial order* as a relation. For the same practical reasons as discussed for the *Extrovert* category (Section 4.5), irreflexivity makes it necessary to have relational transitivity in the modified form of (P3').

The property *Condition* of *Control* denotes a Boolean expression c whose value shall be *True* in order to realize an instance of the dependency during the program's execution. In other words, a *Control* relation $a\ R\ b$ with *Condition* c means that the part b will be executed immediately after the part a only if the involved condition c yields *True* when executing the program. The property *Variables* represents the set of variables that move data from part a into part b for a *Data* dependency $a\ R\ b$. Usually a dependency in the *Data* category expresses a direct flow of data from a statement a to a

statement b during the program's execution. Notice that in this sense a data flow from a to b always implies that a and b are in a (transitive) *Control* relation as well.

The investigation of the concrete dependencies belonging to this category is the most typical target of automatic support within programming environments. The *Data* dependencies could further be classified based on various criteria, e.g. based on the type of the structures between which the relation holds or on the semantics of the data-flow. For example, the following subtypes can be distinguished (see e.g. [47]):

- *IsMovedTo*, the relation between two variables, when data value is directly passed into another variable, but not altered, e.g. in the case of `x=y;` but not in the case of `x=y+1;`
- *ParameterIn*, the relation between the actual and formal parameter in function calls.
- *ParameterOut*, conversely, the relation between the formal and actual parameter.
- *IsReturned*, the relation between the expression that is used to calculate and pass the value of the function and the variable into which it is assigned at the calling level.
- *IsSubScriptOf*, the relation between the index of an array and the array, e.g. in statement `A[k]=x;`

The most well-known examples of structures containing *Data* dependencies are the widely used *data-flow graphs* and *program slices* [49] which will be analyzed in more detail in Sections 5 and 7.

5. Compound dependencies as a basis for hypertextual access structures

Many useful program representations can be expressed as a compound dependency created from elementary dependencies. For example, in a general *control-flow graph* each pair of successive statement-parts is either in an *Imperative* dependency of *Arity 1-to-1* (statement sequences), *Subordinative* of *Arity m-to-1* (unconditional jumps), *Control* of *Arity 1-to-n* (conditional statements), or *Control* of *Arity m-to-n* (controlled iterations). This kind of a hybrid composition can be developed by joining the individual elementary dependencies (here: *Imperative / 1-to-1*, *Subordinative / m-to-1*, *Control / 1-to-n*, *Control / m-to-n*) into a whole. Such a compound dependency is defined as follows:

DEFINITION 5.1. *Let $R_1, R_2, ..., R_n$ $(n \geq 2)$ be concrete program dependencies, that is, instances of the dependency categories shown in Fig. 2. A compound dependency R_c is a relational expression over the elementary dependencies:*

$$R_c = R_1 \bullet R_2 \bullet ... \bullet R_n$$

where each occurrence of the symbol • stands for one of the operations $\cup$ *(union),* $\cap$ *(intersection),* $\setminus$ *(difference), and* $|$ *(restriction). Union, intersection, and difference are ordinary set operations over elements of type* $a\ R_i\ b$. *The restriction* $R_i \mid R_{i+1}$ *yields those elements* $a\ R_i\ b$ *of the dependency* R_i *whose parts* a *and* b *occur also as parts in the dependency* R_{i+1}.[3] *For grouping, subexpressions of* R_c *may be enclosed in parentheses.*

When reaching for a condensed and connected access structure, it is most useful to apply the restriction operation in the underlying dependency specification. For instance, the access structure of Fig. 1 is defined by the expression *PreOrder* | (*IsOfType*(*init-declaration*) $\cup$ *IsOfType*(*assignment*)), where *PreOrder* (being irreflexive, antisymmetric, and transitive) denotes the total preorder relation of the parts in the parse tree.

As another example, the above mentioned control-flow dependency (graph) *CFG* is defined by the expression

$$CFG = Sequence \cup Jump \cup Conditional \cup Iteration$$

where *Sequence* is of category *Imperative* (*Arity 1-to-1*), *Jump* is of category *Subordinative* (*m-to-1*), *Conditional* is of category *Control* (*1-to-n*), and *Iteration* is of category *Control* (*m-to-n*).

Each program dependency R, be it elementary or compound, defines a directed graph consisting of a set of nodes and a set of links. The nodes are the parts in the dependency, and the links are the pairs (a,b) for which $a\ R\ b$ holds. Let us consider the following program fragment where S1, S2, ..., S10 are statements, and E1, E2, E3 are control predicates (Boolean expressions):

```
S1; S2;
if E1 then begin
  S3; S4; S5; S6
end else begin
  S7;
  while E2 do begin
    S8;
    if E3 then S9 else S10
  end
end
```

When considering the fragment's control flow as defined by the compound dependency *CFG*, the directed graph in Fig. 3 is obtained. The nodes are expressed in this and in the following graphs as rectangles and the links as arrows from start nodes to destination nodes. Note that this way of

[3] Union is typically applied for combining specialized and narrow dependencies into more general and extensive ones, intersection is used for finding program parts with several interesting properties, and difference is used for pruning of marginal cases. By restriction a given dependency is bounded to include the parts specified by another dependency.

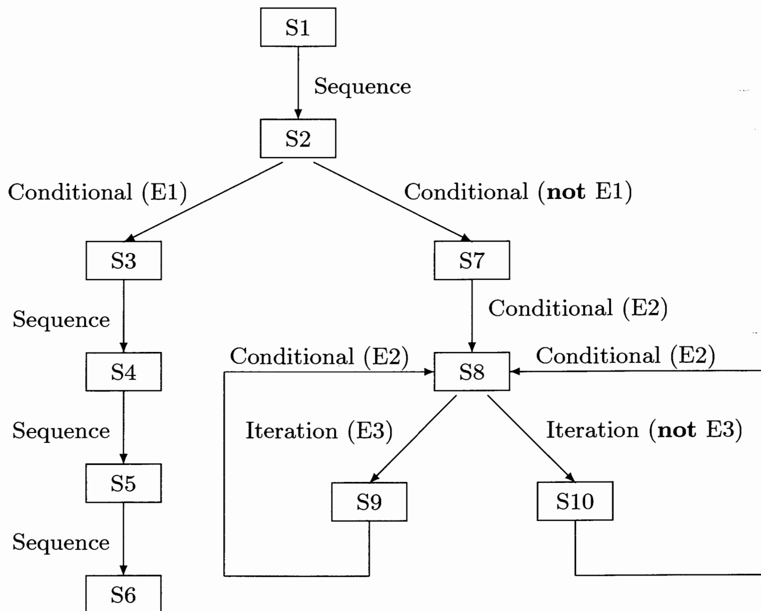


Fig. 3: Control-flow dependency graph.

representing the structures here is selected (among other reasons) because of its compactness as opposite to the representation of actual hypertext (as represented at the interface level of the support environments). The interface level representation of hypertext is discussed in Section 7.

Each link is labelled by the name of the elementary program dependency between its nodes. For the *Control* dependencies the *Condition* property is given in parentheses. In this case the nodes stand for the program statements only (here `S1`, ..., `S10`); another choice would be to include also the conditions (here `E1`, `E2`, `E3`) as nodes.

When focusing on the problematic case of conditional branching, a more specialized graph can be generated by the expression

$$BFG = CFG \setminus Sequence \setminus Jump.$$

The resulting reduced graph for the program fragment is shown in Fig. 4.

We call the directed graph specified by a program dependency an *access structure*. The links of an access structure may be labelled or unlabelled. For instance, the graph in Fig. 3 is the access structure specified by the (compound) dependency *CFG*, and the graph in Fig. 4 is the access structure specified by the (compound) dependency *BFG*.

For usability, it is crucial to select a suitable dependency specifying the access structure. Undisciplined dependency expressions quite easily lead to graphs that are too dense and messy for a user trying to study the under-

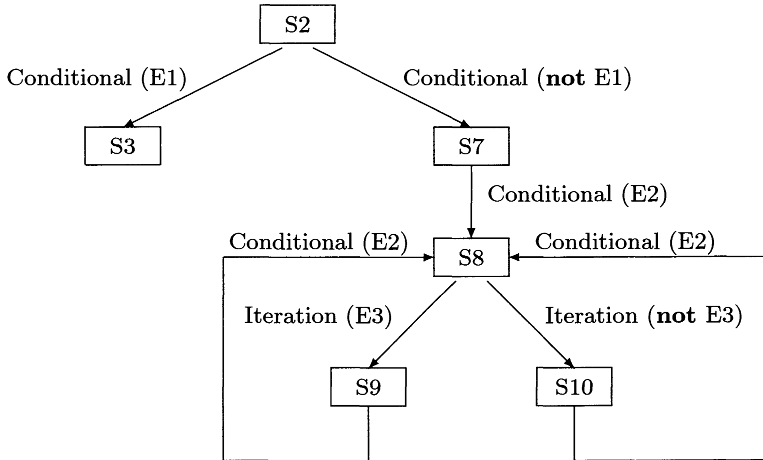


Fig. 4: Branching-flow dependency graph.

lying program. For instance, consider the sample program in Fig. 1. Suppose that an access structure over the program would be specified by the compound dependency *IsOfType*(*init-declaration*) ∪ *IsOfType*(*assignment*), collecting all the variable declarations and assignment statements of the program. The *IsOfType* dependency belongs to the category *Syntactic* and is reflexive, symmetric, and transitive, thus being an equivalence relation. The compound dependency would therefore give rise to the access structure shown in Fig. 5. Symmetric dependencies are depicted by dual-headed arrows. Only the program parts of the compound dependency are included in the graph.

This example gathers many of the problems of forming hypertextual access structures based on program dependencies. By applying the rules given

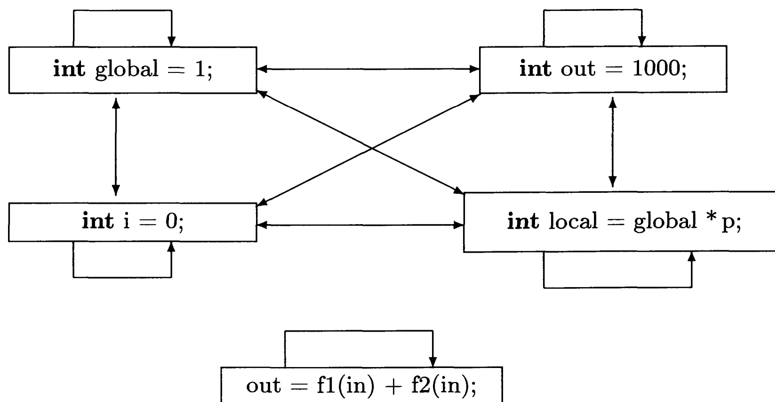


Fig. 5: Undisciplined access structure.

in Section 3 most of the problems can be avoided. For structures based on equivalence relations an additional problem is that the order between the nodes is not determined by the relation itself. The standard solution in HyperSoft is to regard them as sets ordered in terms of their preorder in the parse tree of the program. For the equivalence relations the order often is not crucial/important and if compound dependencies containing also other than equivalence relations are formed, the order of the nodes may be partly determined by the other components of the compound dependency. The ordering problem also emerges, in general case, while forming access structures based on compound dependencies which produce disjoint subgraphs. In that case the separate graphs can be joined by forming an additional root node having links to the first nodes (in preorder) of the separate subgraphs.

In the example of Fig. 1, the elementary dependencies *IsOfType(init-declaration)* and *IsOfType(assignment)* are of the same category *Syntactic*, which obviously is the category of the union dependency *IsOfType(init-declaration)* $\cup$ *IsOfType(assignment)* as well. But what if the elementary dependencies are in different categories, as in the control-flow example of Fig. 3?

The problem of categorizing a compound dependency and its access structure is analogous to the issue of type rules in programming languages: The dependency expression $R_c = R_1 \bullet R_2 \bullet ... \bullet R_n$ is valid if the categories of the dependencies R_i $(i = 1, ..., n)$ are "compatible", in which case the "result category" of the expression is also the category of R_c. In our case all the dependencies are considered "compatible" as they reside in the same hierarchy of categories. The category rule for compound dependencies is given below. The evaluation order of the (binary) operations in the dependency expression is from left to right unless changed with parentheses. Thus, the operation $R_{n-1} \bullet R_n$ yields the category of R_c.

DEFINITION 5.2. *Let $R_c = R_1 \bullet R_2$ be a compound dependency, let C_1 and C_2 denote the category of R_1 and R_2, respectively, and let C_c denote the category of R_c.*

1. *If $R_c = R_1 \cup R_2$, then C_c is the lowest common ancestor of C_1 and C_2 in the lattice of Fig. 2.*
2. *Let $R_c = R_1 \cap R_2$. If $C_1 = C_2$, then $C_c = C_1$. Otherwise $C_c = C_i$, where C_i is a descendant of C_j in the lattice of Fig. 2 $(i = 1,2; j = 1,2; i \neq j)$. If neither of these holds, then $C_c = Nil$ where Nil is a common virtual subcategory of all the leaf categories in Fig. 2.*
3. *If $R_c = R_1 \backslash R_2$, then $C_c = C_1$.*
4. *If $R_c = R_1 \mid R_2$, then $C_c = C_1$.*

For a union of two program dependencies (case 1), it is natural to have their closest joining specifier as the category of the compound dependency. For difference (case 3), the category of the first dependency represents the properties that will be retained when removing a set of relational elements.

For restriction (case 4), the first dependency dominates by neglecting those program parts that are not included in it; therefore its category is the category of the compound dependency as well. Intersection of two different dependencies (case 2) is the most complicated case. The sensible way of applying intersection is over two dependencies that have something in common; in other words when one of C_1 and C_2 is an ancestor of the other (or when they are the same). In that case it is possible for the resulting compound dependency to be nonempty by including program parts in the more specialized category. In an extreme case the dependencies R_1 and R_2 are unrelated, and their intersection necessarily yields an empty set. This possibility is captured in the lattice by the technical category *Nil* whose only purpose is to provide a sound interpretation even for such peculiar combinations of program dependencies.

For example, the compound dependencies *CFG* and *BFG* discussed above belong to the category *Subordinative*, and the dependency *PreOrder* | (*IsOfType*(*init-declaration*) ∪ *IsOfType*(*assignment*)) underlying the access structure in Fig. 1 is of category *Imperative*. Notice that our definition of compound categories is conservative and in some cases just an approximation. A more precise categorization would be reached by analyzing the set of relational elements in R_c instead of analyzing the categories of R_1 and R_2, similar to "dynamic typing" of programming languages. Then, for instance, the *CFG* in Fig. 3 would be of category *Imperative* and the *BFG* in Fig. 4 of category *Control*.

6. Graph theoretical characterizations

Being (directed) graphs, the hypertextual access structures can be characterized by applying the well-known concepts of graph theory. These characteristics can be used for deriving additional information for the user about the access structures, making it easier for her to study them. This derived information may be directly associated with the access structures, or it may be provided externally in the user interface of the programming environment. User interface issues, such as window management, anchoring styles, and dialogue policies, are not discussed in this paper; for a general account of interaction in hypertext, refer, e.g., to [35]. In the following we give some examples of graph-theoretic concepts that are useful in the context of hypertextual access structures. A more extensive list can be deduced from standard literature on graph theory, such as [7].

A *path* is a finite sequence of links where the destination node of each link is the start node of the next link (if any). There is a path from node m to node n if m is the start node of the first link and n is the destination node of the last link in some path. The number of links in the path is called its *length*.[4] If there are paths from node m to node n, then the minimum length of such paths is called the *distance* between m and n. For instance,

[4] Usually a node is defined to have a path of length 0 to itself.

the distance between the first node "**int** `global=1`;" and the last node "**int** `local=global*p`;" in the access structure of Fig. 1 is 4 (the length of the only path between the nodes). Now an access structure might provide a direct crosscut from a node to all the nodes at a given distance from it. For instance, the user might reach the last node of Fig. 1 directly from the first node by a crosscut of distance 4.

Often an access structure contains a specific program part r that the user has selected as the *root* when specifying the access structure she is interested in. For instance, the mandatory *slicing criterion* (usually a variable occurrence in the target program) is the root of a conventional slice. The length of the path between the root and a specific node can be used to restrict the access structure (and its formation) while it is being constructed. This strategy is used in HyperSoft to form partial slices (see Section 7). There might be an implicit root even in cases when the user has not explicitly defined one. For instance, the preorder dependency has introduced the root node "**int** `global=1`;" in the access structure of Fig. 1. For a user-specified criterion, an access structure is the *reflexive transitive closure* of a compound dependency relation R_c with respect to the specified root part r. The access structure contains all those parts for which the relation $r\ R_c^*$ holds. The first node a_0 in the structure is r, and the directed graph $a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow$ ... is constructed from the chain of dependencies $r = a_0\ R_c\ a_1\ R_c\ a_2\ R_c$ ...

Even though in the general case it is not useful to form hypertextual links describing indirect transitivity, in some cases a hypertextual structure showing whether there exists a certain transitive dependency between two nodes can be useful. For example in slicing (see Section 7) it would be faster to determine whether there exists a data or control flow dependency between two distant nodes (or nodes within certain larger contexts, e.g. modules) than to form a complete slice. Likewise, in the case of access structures based on compound dependencies it could be useful to be able to determine whether a certain node can be reached from another node by following links corresponding to certain kind of dependencies.

In terms of closures, another characterization for the existence of a path between a node m and a node n is that the relation $m\ R_c^*\ n$ holds. Accordingly, the distance between m and n is the minimum number d such that $m\ R_c^d\ n$ holds. Given a (compound) dependency R_c for an access structure, a (root) node r, and a distance e, the *local e-neighborhood* of r contains all the nodes n for which $r\ R_c^d\ n$ holds such that $d \leq e$. The concept of local neighborhood is useful in the case of large access structures: Instead of trying to manage the whole access structure at once, one can concentrate on a narrow local e-neighborhood of some suitable range e and study only that. Conversely, the *global e-neighborhood* of r contains all the nodes m for which $r\ R_c^d\ m$ holds such that $d > e$. Besides capturing the distant and often uninteresting connections, a global neighborhood can be utilized for finding anomalous parts of a program. For instance, it is well known that deep loops are problematic both for program comprehension and for code performance. If a loop of depth, say 5, is considered anomalous, then all of

them can be extracted as a global 5-neighborhood of loop statements with R_c being the containment relation (of category *Structural*).

A node with no incoming links is called an *initial* node of the access structure, and a node with no outgoing links is called a *final* node. These can be easily found: Given a dependency R_c for the access structure, i is an initial node if $n\ R_c\ i$ does not hold for any node n ($n \neq i$), and f is a final node if $f\ R_c\ m$ does not hold for any node m ($m \neq f$). Since initial and final nodes make up the borderline between the program parts under analysis and the rest of the program, they should have a special representation in the user interface.

Another class of nodes subject to special linking in the access structure and a special representation in the user interface are the cyclic ones. A path that originates and ends in the same node is called a *cycle*, and all the nodes residing in a cycle are called *cyclic*. In relational terminology, a node n is cyclic if $n\ R_c^+\ n$ holds for the given dependency relation R_c. Having each cyclic node explicitly visualized in the user interface makes it easier for the user to avoid repeatedly traversing over the same sequence of program parts. Also, the notion of cycles may be utilized for discovering intricate programming idioms in the code, such as indirect recursion. For example, in the HyperSoft system recursive calling dependencies are identified based on the already formed access structure and a single link closing the cycle is formed.

Dependencies R_c of arity *1-to-n* and *m-to-n* introduce branches in the hypertextual access structure. The degree of branching can be measured by the *outdegree* giving the number of links starting from a node, and by the *indegree* giving the number of links leading to a node. In other words, the outdegree of a program part a is the number of parts b for which $a\ R_c\ b$ holds, and the indegree of a part d is the number of parts c for which $c\ R_c\ d$ holds. The sum of the outdegree and the indegree of a node is called its *total degree*. For an isolated program part the total degree is 0, and for each part in a function-like access structure the outdegree is 1. Note that since the access structures in HyperSoft are formed so that node pairs are always bound together, the process does not introduce disconnected subgraphs. However, parts of the structure may be isolated in the sense that the total degree related to a certain elementary dependency of a node may be 0.

The concepts discussed above are demonstrated in the access structure of Fig. 6. A is an initial node with indegree 0, G is a final node with outdegree 0, and H is an isolated node with total degree 0. The other nodes have both indegree and outdegree greater than 0; for instance, both indegree and outdegree of B is 2. The nodes B, C, D, and E are cyclic. The (branching) nodes B, C, and F have several destinations. This is illustrated with links of different style (type [45]). There is an infinite number of paths from A to G (due to cyclic and branching nodes), and their distance is 3. The local 2-neighborhood of A consists of A, B, C, and F, and the global 2-neighborhood of A consists of D, E, and G. Finally, notice that the double

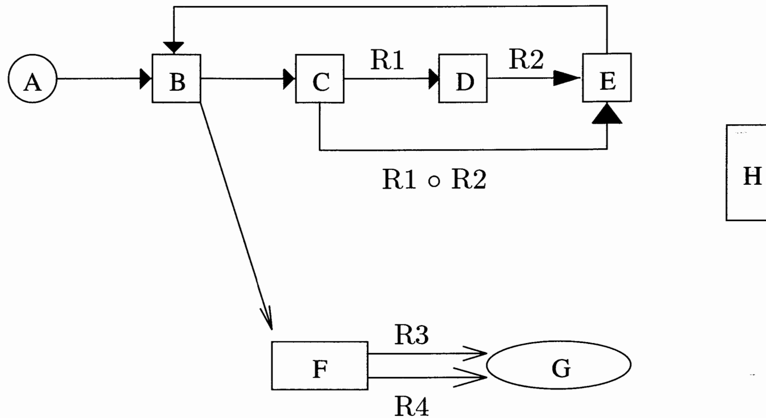


Fig. 6: Rich access structure.

linkage between F and G makes the access structure a *multigraph*, and that there is an induced *composition* relation $R1 \circ R2$ between C and E.

Finally, following the style of conventional software metrics [31, 20], the (graph- theoretic) complexity of hypertext can be measured. These metrics can be used to approximate the readability, usability, and maintainability of access structures and to estimate the effort needed to explore them. Note that since each access structure addresses some aspect of the software, the hypertext metrics in this case strongly correlate with the corresponding software metrics. We omit a further discussion of hypertext metrics and refer instead to [22] where the topic is elaborated in more detail.

7. Access structures in HyperSoft

HyperSoft is an automated assistant for software maintenance, providing facilities for (a) locating the relevant pieces of the software, (b) systematically navigating over it, and (c) modifying it. HyperSoft is technically based on hypertextual access structures as described in Sections 4 and 5, and is thus an example of a software tool where relational program dependencies are in the central role.

HyperSoft is organized into an architecture of four layers, as shown in Fig. 7. The *source code layer* contains the target program code in its original form (as written by a programmer). The *syntactic structure layer* captures the hierarchical structure of the target code as a parse tree and a symbol table. The *access structure layer* provides the facilities for creating access structures by user demand. Finally, the *interface layer* considers the visual representation of the access structures and the interaction with the user.

HyperSoft is implemented in C and C++ and it runs on PC under Microsoft Windows. HyperSoft supports the maintenance of programs written in C (an extension into embedded SQL is currently under development).

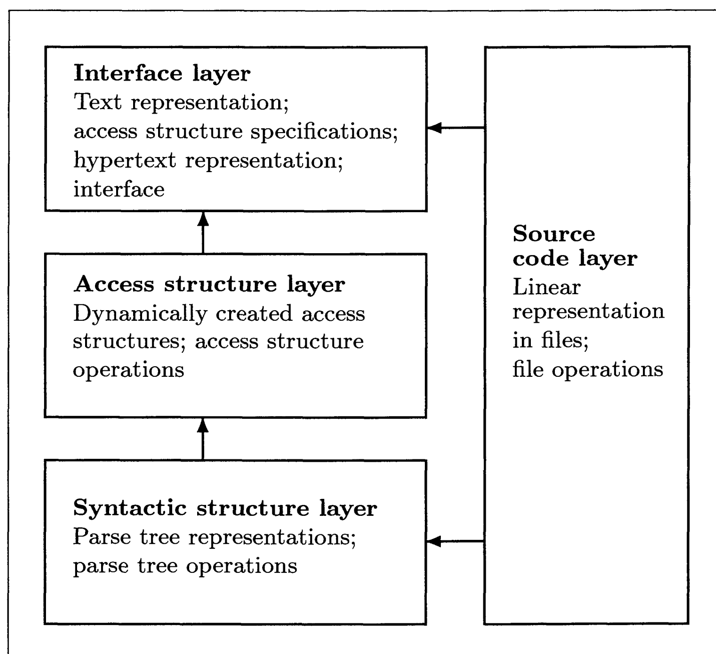


Fig. 7: HyperSoft architecture.

The C parser of the syntactic structure layer has been implemented using the AnaGram metacompiler [37]. The technical architecture of HyperSoft is described in more detail in [44], the implementation of the syntactic structure layer and the access structure layer (including the necessary set of statically and dynamically stored information) in [26], and the design of the interface layer in [34].

The access structures provided by the current version of HyperSoft are *occurrence lists*, *forward* and *backward call graphs*, and *forward* and *backward slices*; a couple of others are being considered for the future versions. The access structure layer contains a generation process for each of the access structures. Each process basically traverses the parse tree, extracts from it the nodes of the access structure (assisted by the information in the symbol table), and generates the linkage of the access structure. The most complicated access structures, forward and backward slices, are produced using iterative solving of data-flow equations during the parse-tree traversal, as originally suggested in [48].[5]

[5] We prefer this solution to the more popular slicing technique of program dependence graphs [21], because the dependence graphs would not be very useful for the generation of the other access structures of HyperSoft.

All the access structures have alternative visual representations in the user interface. The user can study several access structures at the same time in which case each distinct structure is contained in a window of its own. HyperSoft provides facilities for managing projects as collections of program files. Only the files included in the indicated project are subject to analysis, and each project file is shown in its own window.

Occurrence lists

An occurrence contains all the occurrences of a specified symbol (variable, function, type) in the program. There exists a *Semantic Matching* dependency among the program parts (since they cannot be identified purely on syntactic or lexical basis, without taking context into account), which therefore form an equivalence class. The nodes within the access structure are linked as a list based on their preorder in the underlying parse tree.

Calling structures

HyperSoft provides calling dependencies for (an occurrence of) a function f to two directions: the hypertextual structure resembling a forward call graph contains those functions that f calls, whereas the structure corresponding to a backward call graph contains those functions where f is called from. Hence, this pair is an example of a dependency relation and its inverse relation.

Fig. 8 gives an example of the backward calling dependency structure, generated with respect to function `f9` in the program. The access structure extends into three program files, each contained in a window in the background of the figure. These hypertextual windows illustrate the basic representation style of access structures in HyperSoft. The textual content of the nodes belonging to an access structure are emphasized with a reverse color in the windows. The links are (in this case) represented graphically as arrows. Note that the user can browse the program fragments nodewise by selecting the links with the mouse (in case of multiple destination nodes, a pop-up menu is shown). After the selection of a link, the HyperSoft system automatically follows it to the destination node. Links between files are not shown in the hypertextual views as graphical arrows but if such a link is selected, the corresponding text window is either opened or fetched at the foreground.

Fig. 8 also demonstrates alternative dependency representations of access structures which are used to supplement the hypertextual representation. Note that also these views are linked to the source code so that selecting an object within a view causes the corresponding part of program text to be fetched to the active hypertextual view. A *structured map view* (bottom left in the figure) shows the access structure in hierarchical fashion based on the modular and functional decomposition of the program. The other two representations are used to give the user an abstracted view of the access structure: a *global function dependency view* (bottom right) provides

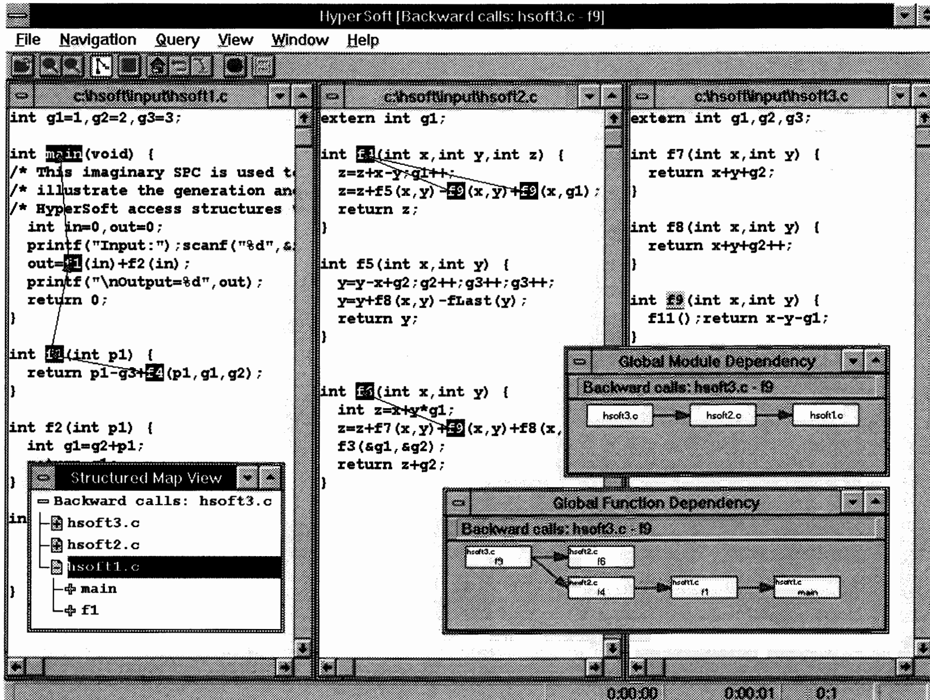


Fig. 8: Backward call graph.

the abstraction in terms of the included functions, and a *global module dependency view* (middle right) in terms of the involved modules (in C: files). From these dependency views it is easy to see that the root function `f9` (in module `hsoft3.c`) is directly called from the functions `f4` and `f6` (in module `hsoft2.c`) and indirectly from the functions `f1` and `main` (in module `hsoft1.c`). These views are formed in the following way: if there exists an access structure node within the larger context (at the selected level of abstraction) an abstract node for that context is created. The node-pairs bound together based on the existing dependencies are checked out and if the destination node is within another context than the start node (and no link between the contexts is already formed), a connecting link is formed.

While the different views stand for the same calling structure, they can be regarded as access structures of different category. The basic call graph on top of the source code links together call statements within function bodies and the enclosing function definitions (their headings). Thus the access structure is induced by the union of the *IsCalledIn* dependency and the *IsDefinedBy* dependency, both of category *Subordinative*. On the other hand, the global function dependency view corresponds to the "conventional" form of call graphs by just containing the calling relationships between functions and therefore belongs to the category *Essential*, as observed in Section 4.1. The same applies to the global module dependency view that captures the intermodular links within the call graph.

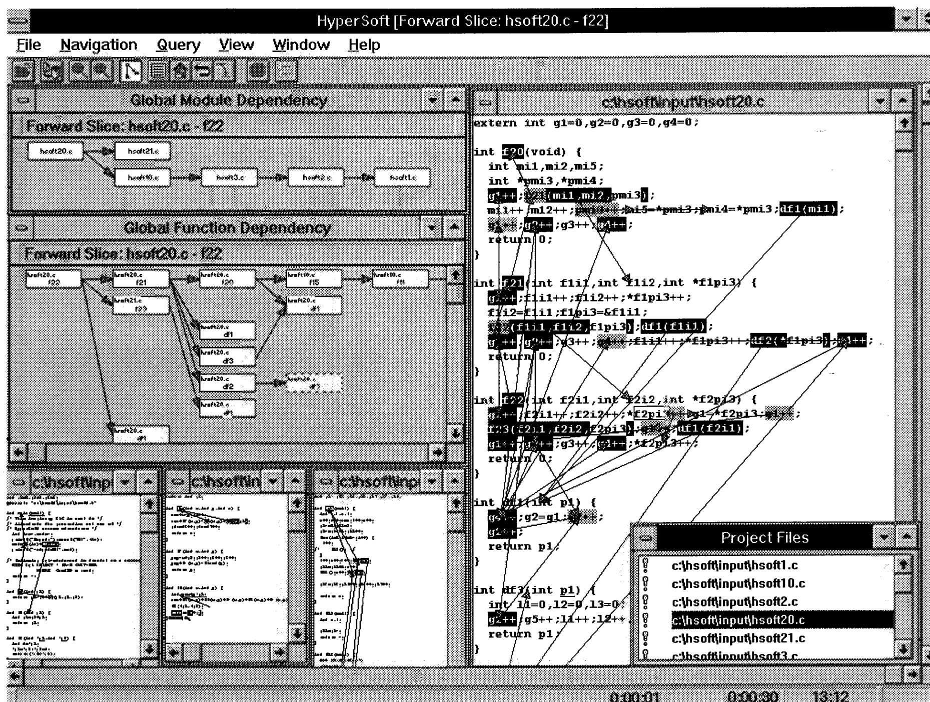


Fig. 9: Complete forward slice.

Slices

Intuitively, a slice with respect to a certain variable occurrence v in the program includes either all those statements that use the value of v, or all those statements that may affect the value of v. In the former case the program fragment is called a forward slice and in the latter case a backward slice, the terms characterizing the direction of data-flow analysis with respect to v. (Backward) slices were originally proposed for debugging [48], and later application areas include program integration, program differencing, impact analysis, testing, and compiler optimization. A survey of program slicing is given in [46].

As mentioned in Section 4.8, a slice combines data flow and control flow. Consequently, the dependency for a slice is of the form *DataFlow* ∪ *ControlFlow*, where *DataFlow* is of category *Data* and *ControlFlow* is of category *Control*. Thus a slice as a whole belongs to the category *Imperative* (cf., Definition 5.2). Within the interprocedural slices, the parameters for which either the relation *ParameterIn* or *ParameterOut* (see Section 4.8) holds so that the passed *Variable* was relevant (at least related to one call of that function) are also appended into the access structure.

Slicing structures can be very large and complicated, as illustrated in Fig. 9 by the hypertextual forward slice with an occurrence of variable `f2pi3` in the function `f22` as the root. The practical value of these kind of large and

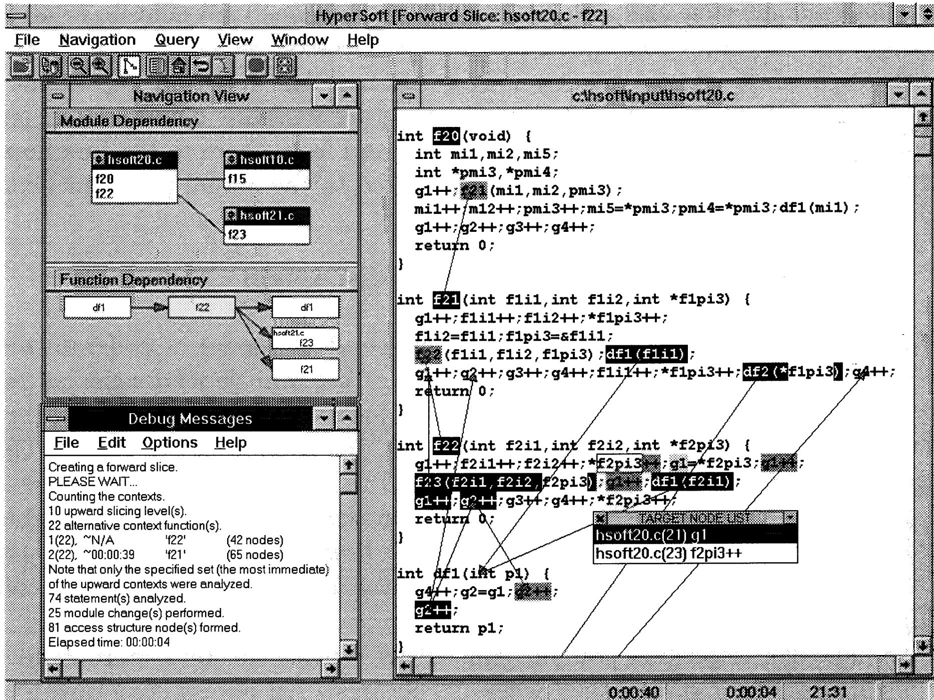


Fig. 10: Partial forward slice.

complex webs is questionable[6], and therefore it is necessary to have additional abstraction capabilities for them. In addition to the global function and global module dependency views, HyperSoft provides a *miniature* over an access structure for grasping an overall view of the structure and its focus. A partial miniature of the example slice is shown bottom left in the figure as windows over three project files.

Even under abstracted views, a complete slice may be incomprehensible for a user. Therefore HyperSoft makes it possible to generate a partial access structure as a local neighborhood of the root node (see Section 6). Fig. 10 contains such a neighborhood for the same forward slice as depicted in Fig. 9. In this case the (interprocedural) slice contains only the relevant statements that reside in functions within one upward-calling level from the function `f22` or within one downward-calling level from those functions.[7] Moreover, only interprocedural links are shown. Notice the simplicity of this partial structure when compared to the complete one in Fig. 9. The *Debug Messages* window contains run-time information of HyperSoft. A *destination (target) node list* provides a menu of alternative links starting from the node which is currently pointed by the mouse cursor. Finally, a (local) *func-*

[6] Especially since the space and time complexity of complete program analysis algorithms (such as slicing) may be prohibitive for large programs [5].

[7] Notice that the drawback of this kind of optimization is the impreciness of the resulting partial access structure.

tion dependency view and a (local) *module dependency* view contain relative navigation information as the immediate predecessors and successors of the function/module currently active. For instance, the function dependency view in the figure tells that the access structure contains direct links from the function `df1` to the active function `f22`, and from there to the functions `df1`, `f23`, and `f21`. Also these views are linked to the actual program text.

8. Related work and discussion

We have presented a general language-independent model of program dependencies. The model is based on treating program dependencies as relations between program parts. The dependencies have been classified according to their fundamental characteristics, taking into account both the relational properties of the dependencies and the way they typically appear in programs. The classification is complete in the sense that all the possible program dependencies can be classified based on it. Most of the currently common practical dependencies belong to the represented *Matching*, and *Imperative* categories.

We have also described how program dependencies can be systematically transformed into hypertextual access structures. An access structure is a directed graph defined by a relational expression over program dependencies, such that the nodes of the graph stand for the program parts in the compound dependency and the links represent the relations between the parts. In addition to the relational properties manifested by the underlying program dependencies, the access structures have structural characteristics that can be analyzed in terms of well-defined concepts of graph theory.

The model can be applied for constructing and evaluating language-based software tools. As an example, we have presented the hypertextual HyperSoft tool for software maintenance. HyperSoft follows the model by automatically extracting from the program the relevant parts and their relationships, and by transforming them into an access structure. The access structures are provided to the user in a graphical user interface. In addition to the default representation as hypertext, HyperSoft visualizes the access structures in a number of alternative abstract forms.

A number of other program dependency models and classifications have been introduced in the literature. For instance, both [11] and [28] describe a conceptual model of programs written in C. The models are based on entities of C (such as variables, constants, data types, and functions) and on their relationships (such as *calls*, *uses*, *defines*, and *includes*). A similar model for object-oriented programs has been presented in [50], emphasizing especially the specific object-oriented entities (such as classes and messages) and relationships (such as *inherits* and *understands*).

These models are more restricted than our model by being based on a particular programming language or paradigm, whereas our model is based on general relational properties of the dependencies with no commitment to

any implementation method. Another essential difference is that the models mentioned above are based on concrete program dependencies, whereas our model is based on their conceptual categories. From this perspective, the other models and classifications can be seen as founded on dependencies that are *instances* of our general dependency categories. The same applies to the extensible and language-independent model presented in [47], where a large number of entities and relationships appearing in typical programs and conventional programming languages is collected.

A more formal approach is presented in [38] by analyzing the general properties of control-flow and data-flow dependencies in programs. Various different forms of these dependencies are discovered and defined in terms of their syntactic and semantic properties. With respect to our classification, the model is restricted to the *Control* and *Data* dependencies only and does not discuss the formal properties of the other categories. Another formalization of control-flow and data-flow aspects is given in [33]. The described system is based on a number of predefined language entities and dependencies that can be used to express control-flow and data-flow properties of a program as formulas in first-order logic. The essential difference to our approach is that a high-level representation of the program is not produced automatically but instead by the user, and the task of the system is to verify that the high-level representation is consistent with the program code in terms of the underlying logical formulas.

Hypertextual language-based tools are described, e.g., in [41], [8], and [53]. As HyperSoft, all these provide high-level views over source code as hypertext. The main difference to HyperSoft is that the models underlying the high-level views are tied to some particular programming language, whereas our dependency model is language-independent (even though HyperSoft applies the model to C).

Since HyperSoft is founded on a relational model of programs, it could well be equipped with a (relational) query language as described in [8] and in [36]. The current way of generating the access structures in HyperSoft is by direct manipulation over the source code using a mouse and a menu of available alternatives. A (textual) query language is one of the possible future extensions for HyperSoft. Other improvements under consideration or implementation include performance optimization, extension into languages other than C, and the support for additional access structures based on our program dependency model.

Acknowledgements

This research is part of the HyperSoft project which is funded by the Technology Development Centre of Finland (TEKES) and by an industrial steering group. The user interface of HyperSoft has been implemented by Mika Nieminen. The comments of the reviewers have been very helpful for improving the style and focus of the paper.

References

[1] *Communications of the ACM* 37, 5, 1994. Special Issue on Reverse Engineering.

[2] AHO, A. V. AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling.* Prentice-Hall, 1972.

[3] AKSCYN, R. M., MCCRACKEN, D. L., AND YODER, E. A. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. *Communications of the ACM* 31, 7, 820-835, 1988.

[4] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[5] ATKINSON, D. C. AND GRISWOLD, W. G. The Design of Whole-Program Analysis Tools. In *Proc. 18th Int. Conference on Software Engineering*, Berlin, Germany, 1996. IEEE Computer Society Press, 16-27, 1996.

[6] BAKER, B. S. A Theory of Parameterized Pattern Matching: Algorithms and Applications. In *Proc. 25th ACM Symposium on Theory of Computing*, San Diego, CA. ACM Press, 71-80, 1993.

[7] BERGE, C. *Graphs and Hypergraphs* (2nd ed.). North-Holland, 1976.

[8] BRADE, K., GUZDIAL, M., STECKEL, M., AND SOLOWAY, E. Whorf: A Hypertext Tool for Software Maintenance. *International Journal of Software Engineering and Knowledge Engineering* 4, 1, 1-16, 1994.

[9] BROOKS, R. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies* 18, 6, 543-554, 1983.

[10] VAN DEN BRAND, M. AND VISSER, E. Generation of Formatters for Context-Free Languages. *ACM Transactions on Software Engineering and Methodology* 5, 1, 1-41, 1996.

[11] CHEN, Y.-F., NISHIMOTO, M. Y., AND RAMAMOORTHY, C. V. The C Information Abstraction System. *IEEE Transactions on Software Engineering* 16, 3, 325-334, 1990.

[12] COUSOT, P. AND COUSOT, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages* (POPL'77), Los Angeles, CA, 1977. ACM Press, 238-252, 1972.

[13] *Computer* 26, 1, 1993. Special Issue on Computer Support for Concurrent Engineering.

[14] CONKLIN, J. Hypertext: An Introduction and Survey. *Computer* 20, 9, 17-41, 1987.

[15] CYBULSKI, J. L. AND REED, K. A Hypertext-Based Software-Engineering Environment. *IEEE Software* 9, 2, 62-68, 1992.

[16] FAUSTLE, S., FUGINI, M. G., AND DAMIANI, E. Retrieval of Reusable Components Using Functional Similarity. *Software - Practice and Experience* 26, 5, 491-530, 1996.

[17] GARG, P. K. AND SCACCHI, W. A Hypertext System to Manage Software Lifecycle Documents. *IEEE Software* 7, 3, 90-98, 1990.

[18] GOLDFARB, C. F. *The SGML Handbook* (Y. Rubinsky, ed.). Oxford University Press, 1990.

[19] GODIN, R. AND MILI, H. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. In *Proc. OOPSLA'93*, Washington, DC. *ACM SIGPLAN Notices* 28, 10, 394-410, 1993.

[20] HALSTEAD M. *Elements of Software Science.* Elsevier, 1977.

[21] HORWITZ, S. AND REPS, T. The Use of Program Dependence Graphs in Software Engineering. In *Proc. 14th Int. Conference on Software Engineering*, Melbourne, Australia. IEEE Computer Society Press, 392-410, 1992.

[22] HATZIMANIKATIS, A., TSALIDIS, C., AND CHRISTODOULAKIS, D. Measuring the Readability and Maintainability of Hyperdocuments. *Journal of Software Maintenance: Research & Practice* 7, 2, 77-90, 1995.

[23] *IEEE Software* 12, 1, 1995. Special Issue on Legacy Systems.

[24] KILPELÄINEN, P. AND MANNILA, H. Query Primitives for Tree-Structured Data. In *Proc. 5th Annual Symposium on Combinatorial Pattern Matching* (CPM'94) Asilo-

mar, CA. LNCS 807, Springer-Verlag, 213-225, 1994.

[25] KNUTH, D. E. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2, 127-145, 1968.

[26] KOSKINEN J. Creating Transient Hypertextual Access Structures for C Programs. In *Proc. 7th Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, 1996. IEEE Computer Society Press, 56-65, 1996.

[27] KOSKINEN, J., PAAKKI, J., AND SALMINEN, A. Program Text as Hypertext: Using Program Dependences for Transient Linking. In *Proc. 6th Int. Conference on Software Engineering and Knowledge Engineering* (SEKE'94), Jurmala, Latvia. Knowledge Systems Institute, 209-216, 1994.

[28] LINOS, P., AUBET, P., DUMAS, L., HELLEBOID, Y., LEJEUNE, P., AND TULULA, P. CARE: An Environment for Understanding and Re-engineering C Programs. In *Proc. 1993 IEEE Conference on Software Maintenance*, Montreal, Canada. IEEE Computer Society Press, 130-139, 1993.

[29] LETOVSKY, S. AND SOLOWAY, E. Delocalized Plans and Program Comprehension. *IEEE Software* 3, 3, 41-49, 1986.

[30] MAGNUSSON, B., ASKLUND, U., AND MINÖR, S. Fine-Grained Revision Control for Collaborative Software Development. In *Proc. 1st ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Los Angeles, CA, 1993. ACM SIGSOFT *Software Engineering Notes* 18, 5, 33-41, 1993.

[31] MCCABE, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering* 2, 4, 308-320, 1976.

[32] MURPHY, G. C., NOTKIN, D., AND LAN, E. S.-C. An Empirical Study of Static Call Graph Extractors. In *Proc. 18th Int. Conference on Software Engineering*, Berlin, Germany, 1996. IEEE Computer Society Press, 90-99, 1996.

[33] MORICONI, M. AND HARE, D. F. The PegaSys System: Pictures as Formal Documentation of Large Programs. *ACM Transactions on Programming Languages and Systems* 8, 4, 524-546, 1986.

[34] NIEMINEN M. Developing the User Interface of the HyperSoft System. Master's thesis (in Finnish), Department of Computer Science and Information Systems, University of Jyväskylä, 1996.

[35] NØRMARK, K. AND ØSTERBYE, K. Rich Hypertext: A Foundation for Improved Interaction Techniques. *International Journal of Human-Computer Studies* 43, 3, 301-321, 1995.

[36] PAUL, S. AND PRAKASH, A. A Query Algebra for Program Databases. IEEE *Transactions on Software Engineering* 22, 3, 202-217, 1996.

[37] Parsifal Software: *AnaGram*™ - *User's Guide*. Parsifal Software, 1993.

[38] PODGURSKI, A. AND CLARKE, L. A. A Formal Model of Program Dependencies and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering* 16, 9, 965-979, 1990.

[39] PRESSMAN, R. S. *Software Engineering - A Practitioner's Approach*, 3rd ed. McGraw-Hill, 1992.

[40] RICH, C. AND WILLS, L. M. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Software* 7, 1, 82-89, 1990.

[41] SAMETINGER, J. AND POMBERGER, G. A Hypertext System for Literate C++ Programming. *Journal of Object-Oriented Programming* 4, 8, 24-29, 1992.

[42] SALMINEN, A. AND WATTERS, C. A Two-Level Structure for Textual Databases to Support Hypertext Access. *Journal of the American Society for Information Science* 43, 6, 432-447, 1992.

[43] SHNEIDERMAN, B. AND MAYER, R. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences* 8, 3, 219-238, 1979.

[44] SALMINEN, A., KOSKINEN, J., AND PAAKKI, J. HyperSoft: An Environment for Hypertextual Software Maintenance. In *Proc. Nordic Workshop on Programming Environment Research* (NWPER'94), Lund, Sweden. Report LU-CS-TR: 94-127, Department of Computer Science, Lund University, 25-37, 1994.

[45] SNAPRUD M. AND KAINDL. H. Types and Inheritance in Hypertext. *International Journal of Human-Computer Studies* 41, 1/2, 223-241, 1994.

[46] TIP, F. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3, 3, 121-189, 1995.

[47] WILDE, N., CHAPMAN, A., AND RICHARDSON, R. The Extensible Dependency Analysis Tool Set: A Knowledge Base for Understanding Industrial Software. *International Journal of Software Engineering and Knowledge Engineering* 4, 4, 521-534, 1994.

[48] WEISER, M. Programmers Use Slices When Debugging. *Communications of the ACM* 25, 7, 446-452, 1982.

[49] WEISER, M. Program Slicing. *IEEE Transactions on Software Engineering* 10, 4, 352-357, 1984.

[50] WILDE, N. AND HUITT, R. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering* 18, 12, 1038-1044, 1992.

[51] WILLS, L. M. Automated Program Recognition: A Feasibility Demonstration. *Artificial Intelligence* 45, 1-2, 113-171, 1990.

[52] YANG, W. Identifying Syntactic Differences between Two Programs. *Software - Practice and Experience* 21, 7, 739-755, 1991.

[53] ØSTERBYE, K. Literate Smalltalk Programming Using Hypertext. *IEEE Transactions on Software Engineering* 21, 2, 138-145, 1995.

VI

HYPERTEXT SUPPORT FOR INFORMATION NEEDS OF SOFTWARE MAINTAINERS

Koskinen, J., Salminen, A. & Paakki, J. 1999. University of Jyväskylä, Jyväskylä, Finland. *Computer Science and Information Systems Reports, Working paper* WP-37. Submitted (Dec. 1999) for publication to *IEEE Transactions on Software Engineering*.

(C) 1999 IEEE. Reproduced with permission.

ABSTRACT

Koskinen, Jussi
Salminen, Airi
Paakki, Jukka
Hypertext Support for Information Needs of Software Maintainers
University of Jyväskylä, Jyväskylä, Finland, 1999
32 pages
Working paper

Professional software maintainers need various kinds of information about the software system they are processing. This information is typically scattered across a variety of sources: human, printed, and digital. Of the digital sources, the most important are program code and documentation. Hypertext is a natural representational form for such fragmented digital information, thanks to its linking facilities. We present a general model and a system for satisfying the information needs of software maintainers by hypertextual access structures. The system provides the maintainer with automatically generated access structures covering a subject software written in the programming language C. While the system restricts the information to static program code only, we show that the general hypertextual model makes it possible to utilize multiple sources satisfying information needs frequently occurring in software maintenance tasks. The information needs underlying our approach are taken from a series of earlier empirical software maintenance studies.

Keywords: hypertext, software maintenance, reverse engineering, program dependencies, program slicing

ACM Computing Reviews Categories:
D.2.2. Software Engineering: Tools and Techniques/
Computer-aided software engineering (CASE), User interfaces
D.2.5. Software Engineering: Testing and Debugging/ *Tracing*
D.2.7. Software Engineering: Distribution and Maintenance/
Corrections, Enhancement
H.3.3. Information Search and Retrieval/ *Retrieval models*
H.5.1. Multimedia Information Systems/
Hypertext navigation and maps

TABLE OF CONTENTS

ABSTRACT 1

TABLE OF CONTENTS 2

1. INTRODUCTION 3

2. SOFTWARE MAINTENANCE 4

2.1. Maintenance tasks 4

2.2. Program comprehension 5

2.3. Potential solutions 5

3. THAS-BASED MAINTENANCE SUPPORT 7

3.1. The HyperSoft model 7

3.2. THAS-based maintenance support environment 8

4. INFORMATION NEEDS OF SOFTWARE MAINTAINERS 10

4.1. Identification of information needs in the studies 10

4.2. Frequent information needs 11

4.3. Information sources 14

5. ACCESS STRUCTURES FOR INFORMATION NEEDS 16

5.1. References 17

5.2. Lists 18

5.3. Sets 19

5.4. Trees 20

5.5. General graphs 23

6. CONCLUSION 28

REFERENCES 29

1. INTRODUCTION

The problems of maintaining large and complicated software systems are prominent and well-known. Due to the increase in system sizes, the vast bulk of old code, and changing requirements, reverse engineering techniques are needed. A combination of reverse-engineering and hypertext techniques can help to overcome some of the maintenance problems commonly encountered. In our approach, hypertext over a target software system is formed automatically. The temporary data structures are called transient hypertextual access structures (THASs). This paper deals with identifying THAS types useful in fulfilling the information needs of professional maintainers.

We have earlier studied the theoretical possibilities of forming different THAS types in detail; see [34]. On the other hand, the practical importance and usefulness of THAS types are also vital aspects. Supporting actual work processes effectively by information technology necessitates that the information needs related to those processes are understood. Often, however, the emphasis is laid on describing the technical possibilities, rather than the user needs. Since it is often not clear which kind of structures are needed by maintainers, we focus in this paper on studying the relation between the information needs of software maintainers and THASs.

THASs can help the maintainer by providing the needed information in a useful and informative way. The formation of a THAS should be sufficiently easy to justify the additional operations and related effort. The benefits come from making the work-process smoother, thus also sparing mental resources. Since different programming languages have different characteristics and information needs are very heterogenous, we shall focus on the programming language C [22] and on the most important information needs as they are represented within the series of empirical studies conducted by von Mayrhauser, Vans and Howe [28,29,30,31].

As a contructive proof-of-concept we shall use HyperSoft [35], a hypertextual tool for the maintenance of software written in C. We shall describe THAS types which cover some of the important information needs of software maintainers. The central research questions, which will be studied in this paper, are as follows.

(1) What are the typical information needs of professional software maintainers?
(2) What are the central characteristics of the information needs which should be taken into account when designing hypertextual support for them?
(3) What kind of THAS types can contain the requested information?
(4) Which of the useful THAS types can be automatically produced?

The paper is organized as follows. First the general problems of software maintenance and their related solutions are briefly surveyed in Section 2. Then the nature and benefits of THAS-based software maintenance support are presented in Section 3. The typical information needs of software maintainers are discussed in Section 4, and THAS types targeted at satisfying some of the most important of those information needs are presented in Section 5. Finally, the results are summarized and directions to further research are outlined in Section 6.

2. SOFTWARE MAINTENANCE

Software maintenance and program comprehension are bound together such that typically program comprehension is a requirement for successfully fulfilling software maintenance tasks. The problems of software maintenance and program comprehension have been attacked in various ways. Since THASs combine the notions of hypertext, reverse engineering and program analysis techniques, only these approaches are briefly surveyed here.

2.1. Maintenance tasks

Software maintenance tasks are characterized by the attributes of their *target system* and the *requirements* for the new behaviour of the target system. The tasks require the source code to be changed so that the system fulfills the new requirements. The user of the support environment is a *software engineer* or *maintainer* maintaining the target system. The maintainer has to comprehend the parts of the program which are relevant to the maintenance task so as to be able to implement the necessary changes correctly without introducing any negative side-effects.

There are many different ways to categorize maintenance tasks. Classifications differ from each other in level of detail. The traditional way, see *e.g.* [37], is to classify the maintenance tasks according to the purpose of the activity into classes of corrective, adaptive, perfective, and preventive maintenance activities. Tilley *et al.* [44] and Arunachalam and Sasso [3] have also created abstract classifications for maintenance tasks. On a more detailed level, software maintenance includes activities such as:

- identification of problems, bugs or requirements,
- reading of comments and documentation,
- planning of required changes,
- chunking of the program,
- generation or revision of hypotheses (concerning the purpose of program components),
- determination of the relevance of program components and their localizations,

- changes to the program,
- manipulation of data, and
- testing the behaviour of the program after changes have been made.

Software maintenance activities as expressed above do not, however, reveal much of the inherent nature of the information needed. It would also be important to study the typical and most time-consuming or problematic elementary tasks and processes which constitute the more complex tasks. Such processes have been revealed in studies which will be discussed in Section 4.

2.2. Program comprehension

Program comprehension is a process intertwined with all maintenance activities. Models for program comprehension have been suggested, for example [8,27,44]. Comprehension is especially problematic when trying to make sense out of so-called *delocalized programming plans* [25] - situations where the code implementing a certain purpose extends beyond the boundaries of functional components - and while maintaining (undocumented) *legacy systems*. *Systematic strategies* of program comprehension can be roughly classified into bottom-up and top-down ones. A *bottom-up strategy* consists of "naming" and conceptualizing the lower-level, elementary components of the program first. Conversely, a *top-down strategy* is based on refining the comprehension by gradually delving into more detailed levels of the program.

The most problematic situations are related to the adaptive maintenance of legacy systems, *cf.* [30], *i.e.* to making extensive modifications to complex software whose documentation is out of date. Making changes without sufficient prior understanding of the relevant program components often introduces side-effects. Such situations will easily lead to problems of corrective maintenance, the other main category of software maintenance. Typical problems related to the corrective maintenance of C programs are analyzed in detail, for example, in [14,15].

2.3. Potential solutions

The techniques most related to the THAS techniques of this paper can be divided into three groups: (1) reverse engineering tools for C, (2) program slicing, and (3) hypertext techniques.

(1) Reverse engineering tools

Reverse engineering means the process of identifying the system's components and their interrelations, and creating representations of the system in another form or at a higher level of abstraction [11]. Established methods of program analysis [2,48] are typically used to produce these higher-level representations.

Many of the existing reverse engineering tools represent the program to the user as program dependencies - see for example [34] - which can be automatically recognized from the source code. For example, control and data flow graphs, and parse trees or abstract syntax trees can be used as a basis for program representations. Reverse engineering systems, such as those represented in [9,10,18,32] typically extract information from a source program as program dependencies between its components and use a program database as an information repository to store the extracted information. Some tools, such as EDATS [49], focus on back-end issues and provide a special query language, while others, such as CARE [26], focus on providing the user with a versatile set of graphical program representations.

(2) Program slicing

Program slicing [47,5] means the extraction of relevant statements from the source program. Slicing helps maintainers to focus their attention on program parts which are somehow relevant to a certain maintenance situation. The focus of interest is indicated by the *slicing criterion*, which typically is a variable occurrence within the program text. Slices are formed on the basis of analysis of the control and data flows of the program, and may be formed either by *static* or by *dynamic analysis* of the program. The information which is needed to form slices is often stored in *program dependency graphs* or in some other similar kind of structures [20,21].

(3) Hypertext techniques

Hypertext [12] is text with nonlinear browsing capabilities, consisting of text fragments called nodes and links connecting these nodes. In most cases hypertext is created manually, although some efforts to form hypertext automatically have been made; see [1]. Automatic generation of hypertext is seldom applied to the documentation of a software system, an exception being [16]. Likewise, the automatic generation in case of program text is rare, exceptions being [7,33,35]. Some information models [41,42,43,46] support the idea of "transient" hypertext formation, thus enabling the formation of THASs.

Software hypertext systems support program maintenance activities by allowing the user to create links between the source code and the related documentation. These systems include those represented in [4,13,39,52]. This kind of linkage, especially between different documents, is generally considered an important form of supporting the maintenance process; see e.g. [28]. Although the capabilities of these systems are clearly useful in providing associations between the source code and the documentation, they are not applicable in the maintenance of legacy systems which either lack documentation or where the documentation is out-of-date.

Most of the hypertext techniques in software engineering environments focus on supporting forward engineering. The system most related to THAS-based maintenance support, and to the HyperSoft system in which THAS-based maintenance support is implemented, is Whorf [7]. Both HyperSoft and Whorf aim to provide automated hypertext support for C language. Furthermore, they both provide a variety of graphical views to supplement the hypertext. The developers of Whorf emphasize the importance of links between different views. Whorf focuses on supporting variable and function cross-references, call graphs and containment. Nørmark and Østerbye [33] have in their HyperPro system especially focused on demonstrating the typing of nodes and links and the internal structures of hypertext nodes. We have aimed at systematically enabling versatile support via a set of THAS types based on our model of hypertext support.

3. THAS-BASED MAINTENANCE SUPPORT

Transient hypertextual access structures (THASs) are automatically formed, temporary data structures enabling hypertextual browsing capabilities over the program text. They are the cornerstone of the HyperSoft model, introduced in [35].

3.1. The HyperSoft model

In the HyperSoft model a THAS is modeled as a directed graph, that is, as a pair *(N, L)* where *N* is a set of nodes and *L* is a set of ordered node pairs called links. Typically, the nodes are parts of specific syntactical types and the link types correspond to program dependencies. The links are formed for enabling unlinear text browsing. We have described and classified program dependencies potentially applicable for creating links in [34].

The HyperSoft approach describes a hypertext support environment in four layers: source code, syntactic structure, access structure, and user interface layers (see Figure 1). It is characteristic of the approach that the access structures are transient (temporary), are generated automatically, the source code is the main input, and maintenance support of legacy systems is the main target area. The syntactic parts of the program text serve as the basis for forming the nodes, program dependencies serve as the basis for forming the links, and both the nodes and the links are generated by automatic analysis. It is important within the appoach to be able to focus on the relevant program parts and on the important dependencies. At the interface layer, graphical representations and abstract views are used to deal with the disorientation and cognitive overhead problems [50] often associated with hypertext systems.

Since legacy software often lacks reliable documentation, the source code has been taken as the primary source of information in the maintenance

environment. The model itself covers documentation as well if it is systematically structured, for example, by using rules regulated by a Document Type Definition of SGML [19]. In that case the linear text of the documentation would be part of the source code layer, the parse tree for documentation would be part of the syntactic structure layer, and the access structures would cover the documentation as well as the source code. In addition, comments [40] embedded within the source code could be structured systematically and thus handled within the model.

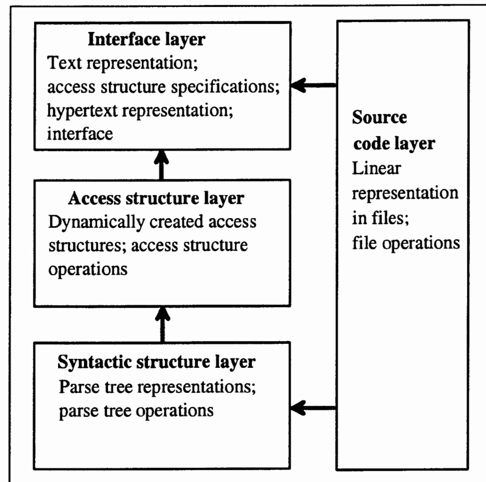


Figure 1. Layers of the HyperSoft model.

3.2. THAS-based maintenance support environment

The THAS-based maintenance support environment HyperSoft [35] has been implemented in the Universities of Jyväskylä and Helsinki and in co-operation with Finnish software houses. Figure 2 shows part of an example THAS within the HyperSoft system. There is a single THAS whose nodes reside in three modules. The three windows contain some of the C source code in the modules. The example THAS is a partial forward call graph initiated from the function identifier `find_moves` (top-left window). The THAS contains a node for each call and implementation of the functions reachable from the `find_moves` function by following function-calling dependencies. The high-lighted text blocks represent the hypertextual nodes and the arrows indicate the links between the nodes.

The example THAS can be browsed in various nonlinear ways by following the links provided by the system. The selection of a node causes the cursor to move to the appropriate target/destination node. In the case of multiple links originating from a node, a pop-up window for making a selection is shown (as in the bottom-left pane of Figure 2). Navigation is supported via various

mechanisms, including the home node link, back-tracking, history lists and abstract and graphical views. The choices regarding applied visual representations and, for example, colors belong to the interface layer of our model (as represented in Figure 1). The way that the user interacts with the system will be further clarified in Section 5. The THAS types currently supported by the HyperSoft system are definition references and occurrence lists for variables, functions and user-defined type names, instance lists of syntactical types, complete and partial forward and backward call graphs, intraprocedural backward slices, and complete and partial interprocedural forward slices.

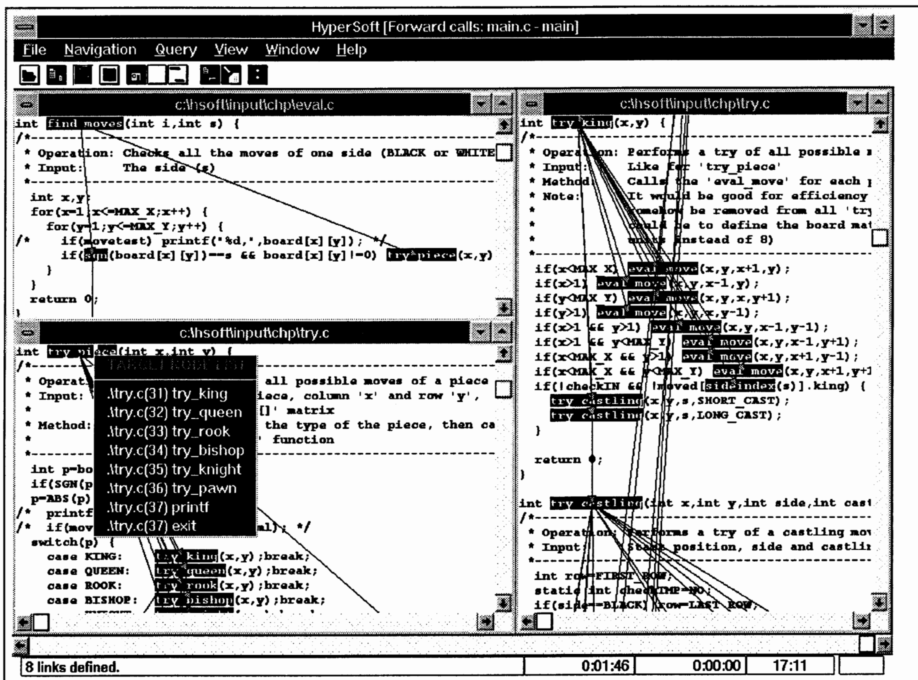


Figure 2. Visualization of a forward call graph in the HyperSoft system.

The usefulness of the THAS types in a maintenance support environment is dependent on their coverage of information needs. Other factors affecting usefulness include the potential accuracy and completeness of the THASs. There also are costs related to forming the THASs, namely issues of technology (time and space complexity) and psychological boundary conditions (cognitive complexity). These issues may make the forming and using of very large and complex THASs inconvenient. In laboratory tests the hypertext support offered by the HyperSoft system was found useful in performing a set of given tasks [23,24].

4. INFORMATION NEEDS OF SOFTWARE MAINTAINERS

Professional software maintenance work has been investigated in a series of empirical studies by von Mayrhauser, Vans, and Howe [28,29,30,31]. These studies have analyzed the general understanding process underlying typical software maintenance activities, as well as the information needs of software maintainers during the performance of their tasks. From the information needs revealed, tool capabilities have been derived that would help them to be more productive. The studies cover the conventional classes of software maintenance quite extensively: a detailed analysis is given for corrective maintenance [29], perfective or enhancing maintenance [31], adaptive maintenance [30], and combinations of these [28]. The studies represent observational field studies of professional software maintainers working on software. Each observation involved a programming session in which the programmer was working with large code consisting of at least 40,000 lines. Sessions were typically two hours long. Thus, one can conclude that these studies constitute a valid and general reference to the maintenance processes that professional software engineers apply when maintaining a production-quality code. They also lay an interesting foundation for analyzing to what extent transient hypertextual access structures in maintenance tools can support maintenance work.

4.1. Identification of information needs in the studies

The series of four above-mentioned studies analyzes and explains the activities of software maintainers, their understanding processes, and their information needs during the performance of their tasks. The purposes of the maintenance tasks were general program understanding, understanding one module, understanding a program bug, fixing a bug, and adding a functionality. The participating maintainers were asked to think aloud while working on the code. The thinking aloud was audio-taped and the tapes were transcribed. The total number of participants in the four studies was 17.

The basis of the protocol analysis of the four studies was the integrative comprehension model [27,28] in which the program comprehension is regarded as a process where the programmer builds and uses three kinds of models: the domain model, the program model, and the situation model. The protocol analysis varied in minor details, but basically consisted of two phases. In the first phase the cognitive activities of the participants and the relationship of these activities to the three models were identified. The second phase identified tasks at a finer level of granularity as well as the information needs for completing the tasks.

Information needs were defined as information and knowledge items that support the successful completion of maintenance tasks. A classification of information needs was developed and the programmer utterances in the protocols

were associated with these categories. For example, the utterance "... because it calls SPOOL-INTO, yeah. So I was at the right place. A long time ago ... " was inferred to belong to the class "List of browsed locations" [31]. The reported data on information needs in [28,29,30,31] is based on the analysis of 13 protocols. These studies do not describe what kinds of tools the subjects used during the programming sessions and what kinds of material the subjects had available. Thus, we do not know whether their utterances were responses to pieces of code on the screen or to other data items.

4.2. Frequent information needs

We collected the information needs reported in the four studies and have summarized the most frequent information needs in Table 1. Needs are ranked according to their total frequency in the studies. The table describes the 24 most frequently occurring information needs in five columns. The first column gives the rank of the information need. The second describes the information as it is described in the source studies. The third states how often the subjects needed the information. The fourth column refers to the sources from which the information could be found in a maintenance environment. The sources will be discussed further in the following subsection. The fifth column refers to the hypertextual access structures we regard as suitable support for finding the needed information. The column will be explained in Section 5.

Some of the information needs are expressed more disambiguously than others. In some cases it is not clear whether the information need means single or multiple pieces of information to be retrieved at any one moment (#1,2,12,19). The specificity of the expressed information needs also varies. Some information needs (#13,16,23) deal with locations only and are thus very general in their nature, whereas some (#2,10,12,17,22) are very precise dealing explicitly with the specific occurrences or values of variables. The information needed in the former case can be provided in many different ways. In relation to many information needs (*e.g.* #4,5) there are simple further specifications, which make the expression more precise; some others (#3,6,21) include even more elaborate descriptions.

It has to be noticed that from Table 1 we cannot draw any direct conclusions about the ranking of programmers' needs for information from external sources. People tend to talk about things they know. In their study [29] concerning corrective maintenance the researchers were surprised that domain concept descriptions were top of the list of needs when they expected finding and removing bugs to require a considerable amount of information at the code level. This unexpected result was explained by the expertise of the subjects: three of the four subjects were domain experts.

Rank	Information need	F	Source				Example id.
			St	Dy	Do	Op	
1	Domain concept descriptions	68	X		X		T1
2	Location and uses of identifiers	54	X				S1
3	Connected domain-program-situation model knowledge	41	X		X	X	-
4	List of browsed locations	33				X	L1
5	List of routines that call a specific routine	29	X				L1,S1,G1
6	A general classification of routines and functions such that if one is understood, the rest in the group are understood	19	X		X		G1
7	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions	18	X	X	X		R1
8	History of past modifications	18	X		X	X	L1
9	Bug behaviour isolated	18	X	X			G2
10	List of executed statements and procedure calls, variable values	17		X			L1,G1
11	Call graph display	16	X				G1
12	Variable definitions, including why necessary and how used, default and expected values	14	X	X	X		R1
13	Location of desired code segment	12	X	X			G2,G3
14	Directory layout/organisation: include files, main file, support files, library files; file structures	12	X				T1
15	Highlighted begin/ends of control blocks	11	X				T2
16	Location of where to put changes	10	X		X	X	G2,G3
17	Conditions under which a branch is taken or not: include variable values	9	X	X			T2,G2,G3
18	Exact location to set breakpoint	7	X	X		X	G2
19	Location and description of library routines and system calls	7	X		X		R1,S1
20	Ripple effect	7	X	X			G3
21	Naming conventions separated by systems or library objects that use them; rules for naming new procedures	7	X		X		-
22	Expected program state, *e.g.* expected variable values when procedure is called	7	X	X	X		G2,G3
23	Good direction to follow given what is already known, possible program segments to examine	7	X	X	X		G2,G3
24	Nesting level of a particular procedure	7	X				T2

Table 1. Frequent information needs.

#	Information need	S1 Gen.	S2 Corr.	S3 Perf.	S4 Ad.
1	Domain concept descriptions		I		I
2	Location and uses of identifiers	IV	III	II	
3	Connected domain-program-situation model knowledge		II	IV	
4	List of browsed locations		IV	I	
5	List of routines that call a specific routine	II	V		
6	A general classification of routines and functions such that if one is understood, the rest in the group are understood			V	
7	Format of data structure plus description of what field is used for in program and application domain, expected field values and definitions	V			
8	History of past modifications				III
10	List of executed statements and procedure calls, variable values				V
12	Variable definitions, including why necessary and how used, default and expected values	I			
14	Directory layout/organisation: include files, main file, support files, library files; file structures				IV
15	Highlighted begin/ends of control blocks	III			
16	Location of where to put changes			III	
20	Location and description of library routines and system calls				II

Table 2. The five most frequent information needs in the four studies.

They probably already had knowledge of the things they were talking about and did not lack all of the domain information they referred to. It is also possible that the subjects as experts recognized pieces of important information which for non-domain experts should be available in the documentation, even though that themselves did not lack this information. In our interpretation the information needs listed in Table 1 do not necessarily indicate lack of information in the case of the subject, but they refer to useful item of information which many potential programmers, given that sort of task, would lack.

The four studies showed, as expected, that the information needs of different people engaged on different maintenance tasks differ. Table 2 summarizes from the information needs given in Table 1 the five most important to emerge in each of the four studies. The table contains six columns. The first gives the total rank in Table 1. The second describes the information. The columns from the third to the sixth show the rank of the information need in the individual studies. The third column refers to the study [28] in which different maintenance tasks were involved (S1). The results concerning that study were based on the analysis of five protocols. The fourth column refers to the study [29] which concerned corrective maintenance (S2), and had four subjects. The fifth column refers to the two-subject study [31] concerning perfective or enhancing

maintenance (S3). And finally, the sixth column refers to the adaptive maintenance study [30] (S4), which also had two subjects. Table 2 shows that the order of importance of information needs clearly varies with different types of tasks. Each of the five most important information needs in Table 1, however, also occurs among the five most important ones in at least two of the four individual studies.

4.3. Information sources

Above, we discussed the information needs of software maintainers as these were identified and expressed in the series of four studies. The information needs were defined as "information and knowledge items that support the successful completion of maintenance tasks". Such support can only exist if the items are available and make sense to the individual in the specific situation. The items may, at least partly, be in the mind of that individual. The remainder should be found from external sources.

The information sources available to a software maintainer outside his or her own head can be classified into three major categories: other persons, digital information available in the computer-aided maintenance environment, and literal information outside the maintenance environment. A goal in the development of computer-aided maintenance environments has been to improve the availability of useful information by the tools of the environment. The digital information sources of a computer-aided maintenance environment can be divided into four major groups. First, the source code and the information made available by the static analysis of the code. Second, information made available by the execution of the code. Third, the software documentation and other literal material available in the environment, as well as the information made available through the analysis of that material. And fourth, the session history. Below, we shall discuss the information needs related to these classes. The sources of the different needs are summarized in the fourth column of Table 1.

4.3.1. The source code

A great deal of the information needs in Table 1 concern program parts of a specific syntactic type: variable definitions (#12), routine or function names (#5,6,11), identifiers (#2), statements within program slices (*e.g.* #9,20), beginnings and ends of control blocks (#15), data structure definitions (#7), branch conditions (#17), and library routines and system calls (#19). In some cases the information need can be satisfied only if the parts are seen in the code context. In other cases the values of the parts can be accessed and listed, or graphically represented out-of-code. For example, the beginnings and ends of control blocks (#15), and location and uses of identifiers (#2) have to be seen in the code context. On the other hand, a list of routines that call a specific routine

(#5) can be given as a separate list. Call graph display (#11) requires showing the calling dependency graphically.

Information about the environment - local or global scope (#12) - is an example of a need which does not involve a specific syntactic structure, but which to fulfill the programmer has to see the code surrounding the current position. The needs for function call count (#5) and count of variable use (#2) concern information which is not directly in the source code text but has to be counted from the occurrences of function calls and variables, respectively. This kind of scope and count information has been regarded as important in the study of [28]. Similarly, the nesting level of a procedure (#24) and metric values are based on additional analyses and calculations. Information about past modifications (#8) may be included in program comments. If the program database contains several versions, modification history can be retrieved from the database.

4.3.2. Code execution

Information about executed statements and procedure calls (#10), and variable values (#10,17,22) can be accessed only if the maintenance environment also includes run-time capabilities. Some information needs may be satisfied only partially via static analysis (#7,17). Finally, the support of many information needs (#9,12,18,20,23) is leveraged beyond the limits of static analysis via the application of dynamic (run-time) analysis of the software.

4.3.3. Documentation and other literal material

Program documentation usually consists of comments in the source code and other related documents. In case of legacy software, a well-known problem is unreliable or lacking documentation. Thus program documentation as an information source is not always available. There will, of course always be some literal material concerning the task always available, such as programming manuals, operation system manuals, and literature about the special domain of the software.

The most frequent information needed in the four studies concerned domain concept descriptions. This information need occurred most often in adaptive and corrective maintenance (see Table 2). As discussed above we do not know how often a need occurrence in one of the four studies concerned a need for a new information item and how often the need actually said something about the knowledge the subject already had. Nonetheless, even programmers familiar with the domain may lack much of the domain information and to the maintainers of legacy software the domain and its notions at the time the software was written may be very unfamiliar.

Naming conventions (#21) are typically information items which the programmer does not know undertaking the maintenance of an unfamiliar software. In some cases the naming conventions may be inferred from the code.

Specification of the naming conventions, acronyms, and other agreements about the way the code is written are of course helpful in the maintenance situation. Hence, information stored in documentation and other literal material is often needed together with information in the code.

4.3.4. Session history

Some of the needed information is created during the maintenance session. Information about browsed locations (#4) has to be collected during the code reading. Also, support for checking the history of past modifications (#8), determining the location of where to put changes (#16), and finding the location to set a breakpoint (#18) may require analysis of the user operations.

4.3.5. Combining multiple sources

An important result in the studies by von Mayrhauser and Vans is that programmers use a multilevel approach to program understanding, switching between program, situation, and domain models, and flexibly needing different kinds of information. A specific information need called "Connected domain-program-situation model knowledge" (#3) related to the different layers was identified. Fulfilling this need typically requires multiple sources. Also many other information needs (*e.g.*, #6,7,8,12,16,18,22,23) relate to multiple sources. In their papers von Mayrhauser and Vans suggest rich cross-referencing between different kinds of information and information sources. Recall that our hypertextual software maintenance environment applies source code and session history as information sources. However, the environment could well be extended by incorporating capabilities for code execution and partially for systematically structured documentation as well.

5. ACCESS STRUCTURES FOR INFORMATION NEEDS

As defined in Section 4, an information need consists of information and knowledge items that support the successful completion of a maintenance task. As further discussed in Section 4, these items can be obtained from a number of sources, either by human or automated means. In this section we shall go one step further by analyzing which kinds of concrete access structures based on the information and knowledge items can be provided in a software maintenance environment. We use the HyperSoft system as a case environment, and thus the information needs are mapped onto the special forms of transient hypertextual access structures.

The access structures are classified according to their structural topology: *references* are discussed in Section 5.1, *lists* in Section 5.2, *sets* in Section 5.3, *trees* in Section 5.4, and *general graphs* in Section 5.5. These topological access structure types and their relational foundations are analyzed in more detail

in [34]. Each topological type is illustrated with sample access structures. The access structures are mapped together with their target information needs by giving the identifiers (*e.g.*, R1) of the relevant access structures in the last column of Table 1.

5.1. References

References are the most simple form of access structures: a link is provided from an information or knowledge item to its definition. Although being very simple, the usefulness of references has been shown both by the empirical studies of von Mayrhauser *et al.* and in studies made by ourselves [23,24,35].

(R1) Definition references

Any occurrence of an item in the source code that needs explicit definition immediately raises the need for a reference access structure - a direct link from an occurrence to its definition. The named items of programming languages (constants, variables, types, routines, procedures, and functions) are the basic elements in any program; hence their meaning must be frequently studied in typical maintenance tasks.

The descriptive information needs over data structures, variables, and routines supplied in Table 2 (#7, 12,19) can be served by definition references. As an example, Figure 3 illustrates a hypertextual access structure for information need #12: "variable definitions, including why necessary and how used, default and expected values". In the figure, the maintainer wants descriptive information for the variable `s` in trying to understand a statement in a chess program. Notice that the information may be spread over multiple sources. In this case the information items are found in the source code (by static analysis) and in the program's documentation.

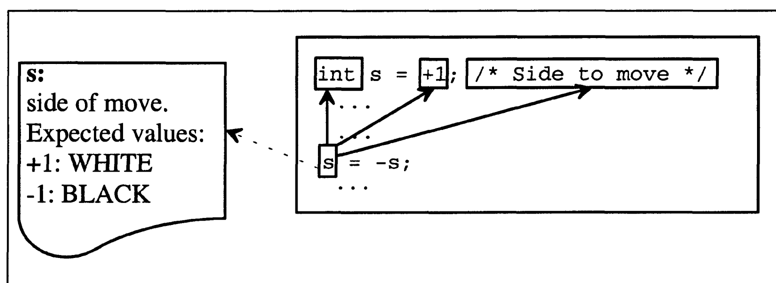


Figure 3. Definition references for a variable.

In the case of a structured variable (#7), such as a record, the access structure would contain the same information but the link leading to the type (`int` in Figure 3) would then lead to descriptions of its fields. In the case of library and

system routines (#19) the descriptive information would mostly be found in literal documentation or in software libraries.

HyperSoft provides definition references for variables, user-defined types, and function identifiers. Thanks to the use of a static program database, these access structures can be formed very rapidly. Note that since HyperSoft only applies static program analysis over a context-free grammar, documentation and comments are excluded from its access structures.

5.2. Lists

A list is composed of an ordered sequence of items. In a software maintenance session, the order of the user's operations is a most natural criterion for ordering the items over time (*cf.* Section 4.3.4). The source code also spans a number of natural ordering criteria, such as the textual ordering of the statements in the program (or, the preorder of the statement nodes in the parse tree over the program). A list is also often a natural representation also for (partial) hypertextual access structures that by their inherent nature are sets, trees, or graphs. This aspect will be explored in Sections 5.3, 5.4, and 5.5, respectively.

(L1) History information

Information about browsed program locations (#4) and past program modifications (within a session) (#8) has to be collected during the maintenance session. Session-specific history traversal is a standard facility in hypertext systems, being supported in various ways such as links to the previous node, next node, and home node. In HyperSoft the user generates a transient path when browsing the source code along an access structure, and this path can be traversed in a standard hypertextual manner. Notice that such a path always forms a special route (*i.e.*, a subpath) in the access structure.

Figure 4 depicts a situation where the maintainer is browsing the occurrences of variable `s` and is only interested in those statements where `s` is assigned a value. The browsed program locations are emphasized.

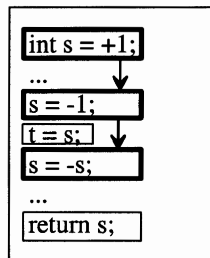


Figure 4. List of browsed locations.

Code execution also induces a temporal order among information items. For instance, information need #10 in Table 1 calls for a list of executed statements and procedure calls. The information need can be satisfied with a similar list over program statements as shown in Figure 4. Now the items in the list would stand for the statements that have been executed when running the program, in their execution order (thick frames in Figure 4), while the unexecuted statements would remain outside the list (thin frames in Figure 4), *i.e.* `t=s;` and `return s;`.

The history mechanism could even be extended beyond session boundaries. For instance, by comparing two versions of a program in terms of their parse trees [51] one could automatically find the program deltas for a source code control system or a configuration/version management system (#8). The program parts would then be arranged in a linear list according to their age.

5.3. Sets

In many cases the maintainer needs a collection of information or knowledge items that share a certain property. For instance, need #2 in Table 1 calls for a set of identifiers in the program, with no particular preference among the occurrences. In relational terminology such a set of similar items forms an equivalence relation; see [34]

Representing a set-based access structure as hypertext introduces a problem: how should the items in the set be linked? In most cases a complete *n*-to-*n* linkage over the access structure, from each item to each item would not be sensible, even though this solution would in fact match with the equivalence-relational property of the access structure. Inducing an order between the items in the access structure effectively transforms its representation into a list, whereby the maintainer will browse the items of the set via the ordered links between them.

(S1) Variable occurrences

As was shown by the tables in Section 4, identifiers (#2) in the program are a central information source for a professional maintainer. This information need is served in HyperSoft with an access structure that contains all the occurrences of a selected variable or user-defined type. The access structure can be entirely generated by static analysis of the source code. As additional information, HyperSoft also gives the size of the access structure as the number of occurrences in it. According to our reference studies, count information is freqently needed by maintainers when estimating the efforts of their comprehension and maintenance processes.

As discussed above, the occurrences of an identifier conceptually form a set with no specific ordering. To make the access structure usable, HyperSoft links

the occurrences into a directed list starting from the occurrence selected by the maintainer as the seed. The linkage follows the textual order of the occurrences within the program. Note, however, that this choice is not the only one possible: one could, for instance, link occurrences according to their nesting level (first global occurrences, then local occurrences), or according to the order of execution of the statements that access the occurrences. Figure 5 gives an example of an occurrence list, starting from an occurrence for variable `s`. Since the relevant information can be found directly from the static program database, the generation of occurrence lists is very fast. On the right, the figure shows the names of the files of the project currently being processed (project files window) and status information generated during the formation of the THAS (debug messages window).

Other frequent information needs that can be served as list-represented sets of information or knowledge items are routines that call a specific routine (#5), and location of library routines and system calls within the source code (#19).

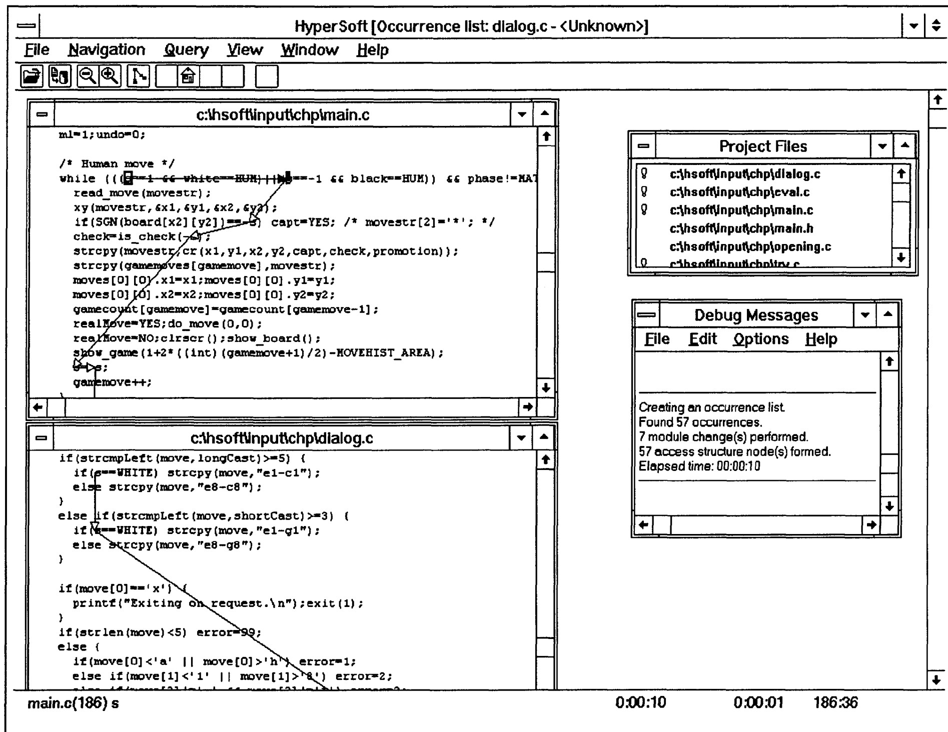


Figure 5. Occurrence list for a variable.

5.4. Trees

If certain information or knowledge items have dominance over other items, a convenient representation for the access structure is a tree, or hierarchy. In such

cases the items are in an antisymmetric relation; *cf.* [34]. Tree-formed access structures are quite useful in software maintenance environments, especially when the source code is a central information source whose standard internal representation takes the shape of a dedicated tree: a parse tree. This is also the case in HyperSoft where the whole source code could actually be represented to the maintainer as a hypertextually linked tree. Such a huge tree would, of course, not be usable; hence some filtering mechanisms must be applied to abstract the parse tree in order to create a more compact access structure.

(T1) Domain concept descriptions

Professional maintenance engineers, especially those familiar with the application domain, frequently make assumptions about the module and control structure of the program they are working on. Such assumptions are based on general concepts of the domain and standard solutions as given in text books and other documentation. For instance, a person responsible for the maintenance of a compiler can usually safely assume that it is organized as a sequence of "phases" such that a "parser" calls a "scanner", a "semantic analyzer" and a "code generator".

These kind of frequently arising opportunistic information needs (#1 in Table 1) can be served by an access structure that contains the main components of the software organized into a hierarchy according to their structural or calling relationships. For instance, the access structure in Figure 6 exemplifies the domain concepts of a typical compiler. In cases where the links stand for calling relationships, the access structure is commonly termed a “call graph” (*cf.* Section 5.5), while the access structure would be termed a “module dependency graph” or “class diagram” in the case of a structural containment relationship.

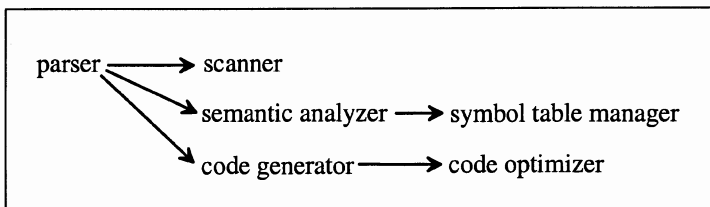


Figure 6. Domain concept description for a compiler.

If documentation is available in digital format, the code-level access structure can also contain links (references) from its nodes to the documentation that explains how the concepts are typically or in this particular case implemented. Note that links to documented standard practices are quite useful, *e.g.* when analyzing the quality of the particular solutions in the system under maintenance by comparing the actual software architecture with the standard conceptual architecture; *e.g.* [6]. Tree-formed access structures are natural

representations also for other kinds of organized information, such as directory and file structures (information need #14).

The HyperSoft system supports the information needs of domain concept descriptions by the view of "global module dependencies". Views are provided in HyperSoft to supplement the hypertextual browsing. An example is given in Figure 7, with the module dependency view on the right. The view has been formed automatically for the active access structure – in this case the call graph used as an example earlier in Figure 2. The upper part of the view shows the interrelations on the module level and is thus rather abstract, whereas the lower part shows them on a more detailed level (as individual functions which generate the existence of module-level relationships). The items may be ordered in many ways - in this case they are ordered on the basis of calling dependencies. The view shows, among other things, that `find_moves` (implemented in the module `eval.c`) calls the function `try_piece` (implemented in the module `try.c`) (which were the two first levels in Figure 2). Note that the user can collapse the details within the boxes. For example, if the user wants an abstracted view of the content of the module `eval.c`, only its external interface would be shown. As the figure suggests, at the detailed level the conceptual information tends to form a general graph, which is discussed in Section 5.5.

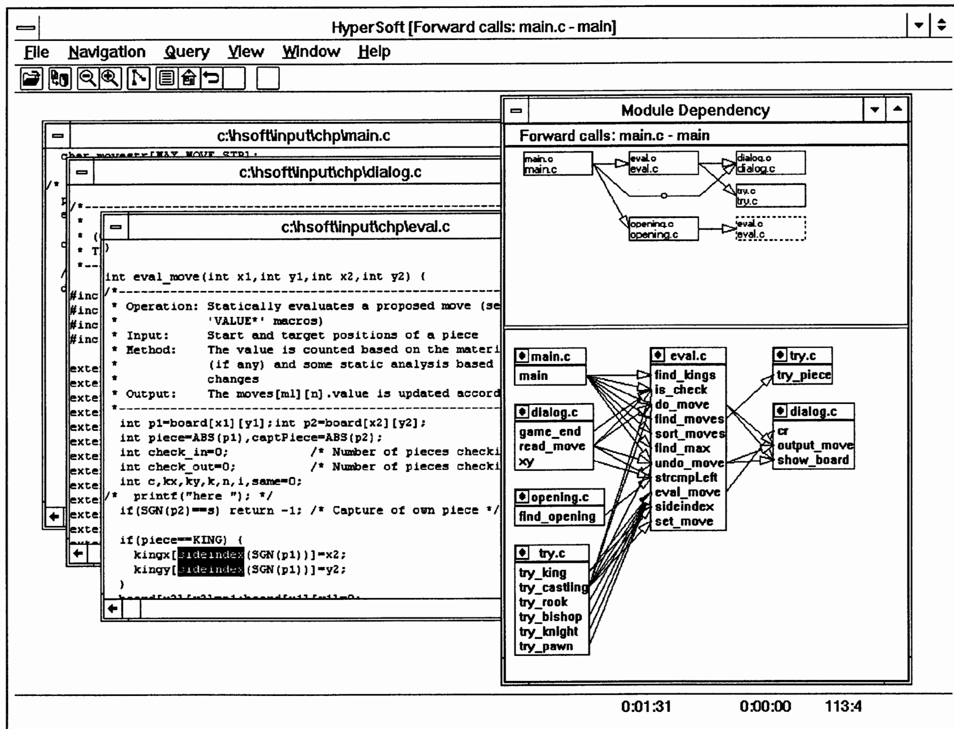


Figure 7. Global module dependencies.

(T2) Control blocks

The boundaries of control blocks help a maintainer in grasping a general understanding of the program's internal structure (information need #15 in Table 1). Since the control blocks can be found by pure static analysis of the source code and are nested within the underlying parse tree, a natural access structure for them is a tree where the levels stand for the textual containment of the blocks. For a control block standing for a procedure, its nesting level can also be indicated in the access structure, so as to serve information need #24. Moreover, the blocks for conditional and iterative statements can be automatically associated with a link to the predicate that controls the branching within the statement, so as to serve information need #17. Although HyperSoft currently supports the representation of control blocks explicitly in the form of a simple list only, it also provides a more general solution, program slicing, which is discussed in next section.

5.5. General graphs

The flow of control and data, two of the most central aspects in a software system, are usually presented as control- and data-flow graphs (or their variants). Accordingly, graphs are quite often needed by a software maintainer who has to understand the internal flow of information within the target software.

(G1) Call graphs

Call graphs are one of the most well-known abstract representations of a program. A call graph visualizes the calling dependencies between the subroutines, procedures, and functions of the program, and by this expresses its overall control flow. HyperSoft supports control-flow understanding to two directions: backwards with respect to the selected routine, and forwards with respect to that routine. A backward call graph shows how program execution has reached the current point of interest, while a forward flow graph shows how the execution will proceed from the current point. A forward call graph was depicted in Figure 2.

Software maintainers quite often need a total call graph over the program, such as the one in Figure 2, in order to obtain a general understanding of its functional behavior (information need #11 in Table 1). In more specific maintenance situations, however, more detailed views may be more helpful. Most notably, information need #5 calls for a partial graph: the routines that call the routine of interest. Such partial call graphs are also provided in HyperSoft since the user may specify the extent of the graph to be produced.

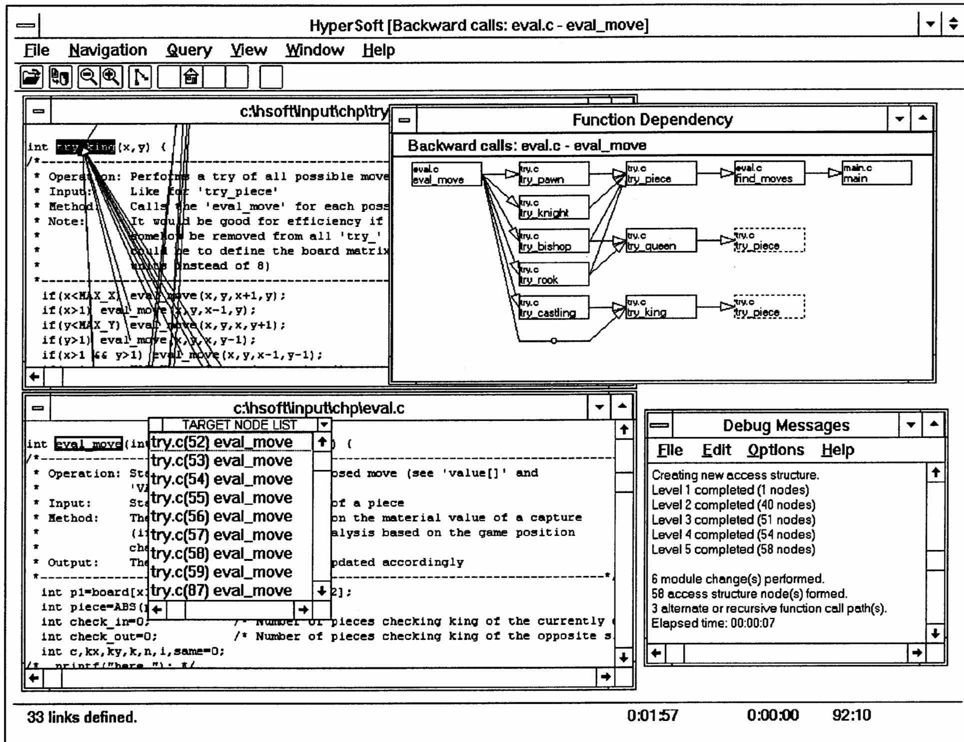


Figure 8. Partial backward call graph.

For example, the partial backward call graph in Figure 8 serves information need #5 by showing (as a target node list) links to the places from where the selected function `eval_move` is called from (in the pop-up window). The figure is supplemented by a graphical function dependency view showing the linkage graphically in abstracted form.

When going beyond one calling level, larger partial call graphs can be obtained. This HyperSoft mechanism can be used, for instance, in the classification of routines into calling-coupled groups (information need #6). As in the usual case, a call graph can be shown to the user both as hypertext and as an abstract graphical view, whichever variant it may take (forward, backward, complete, partial). The rich selection of different call graphs in HyperSoft is quite competitive with dedicated call graph extractors that usually just generate a single variant of them [32]. A further improvement would be to involve dynamic code analysis as well, for instance to generate a list of executed procedure calls (information need #10).

(G2) Backward slices

When more detailed information about program execution is needed, it can be provided by access structures commonly known as program slices. The concept

of slices were originally introduced by Weiser [47] to support program comprehension and, especially maintenance tasks related to debugging. Since, slices have found useful applications in a number of other software engineering areas as well, examples being impact analysis, testing, and compilation. An extensive survey of slicing techniques is given in [45].

A slice combines the representations of control flow and data flow, usually with respect to occurrence of a variable in the program under investigation. The original notion of a slice, currently known by the special term backward slice, contains all those statements of the program that may affect the indicated variable occurrence (the slicing criterion). Slices may be generated by pure static analysis, or by a combination of static and dynamic analysis. Static slices are more general (since they apply to all the possible executions of the program), whereas dynamic slices are more precise (since they grasp precisely the relevant statements for a specific execution).

Since HyperSoft is a general software comprehension and maintenance tool, it produces the slices by static analysis. The backward slices are provided in intraprocedural form, that is, they express the data flow within the current C function. The slicing criterion is a certain variable occurrence in the function, and the access structure includes all the occurrences of the identifier, within the function, whose value may have an effect on the value of the slicing criterion. The analysis can be extended to macros in HyperSoft: in addition to variables, the identifiers in the slice may stand for macros as well.

Figure 9 shows an example of a backward slice in HyperSoft, with the last occurrence of variable `x` as the slicing criterion. The slice reflects the order of computation within the program and shows how the computed values proceed towards the criterion. As noted above, backward slices are useful especially in debugging and related activities – in this case the slice includes those parts of the program that might be the origins of the incorrect (output) value for `x`. Hence, the person maintaining the program can focus on analyzing and debugging the slice only (information needs #9,13,16,18,23), and omit the rest of the program. Note that HyperSoft uses highlighting and colors to convey session-specific information, in this case the currently visited node is `z3=z5+x-f+a`, from which four links originate to points were `a`, `x`, `z5`, and `f` obtain their values.

The analysis could be supported further by a sophisticated "algorithmic" debugger that would automatically execute the program and stop for user interaction at those statements that are included as breakpoints in the slice [17]. Such an access structure extension to HyperSoft would support information needs where knowledge about code execution is most essential (#17,18,22).

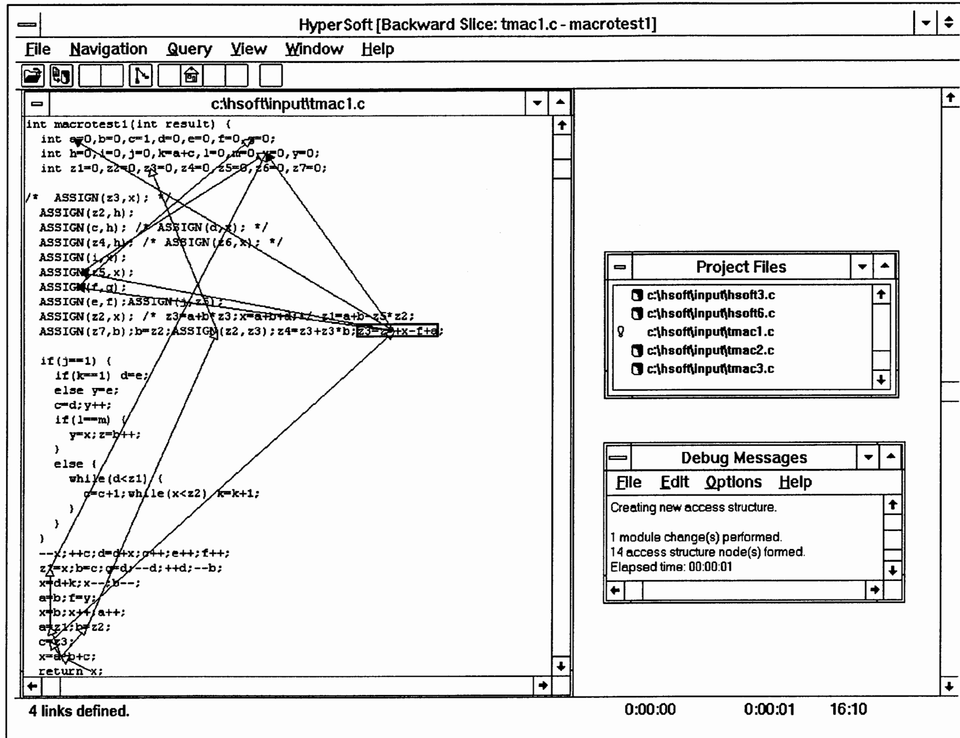


Figure 9. Backward slice.

(G3) Forward slices

As characterized above, a backward slice contains those parts of the program that may affect the value of the slicing criterion. This analysis can be reversed, so as to generate those parts that may be affected by the slicing criterion (occurrence of a variable). In such a case the resulting access structure is called a forward slice, the term emphasizing the direction of the data flow within the slice with respect to a value of the indicated variable.

Forward slices are useful especially in ripple effect analysis (information need #20), since they contain those parts of the program whose behavior might be crippled when modifying the slicing criterion; see *e.g.* [38]. Forward slices also support maintenance tasks that utilize a sequence of relevant program parts on the basis of the program's control and data flow (information needs #13,16,23). Akin to backward slices, forward slices can be generated by static or dynamic means. Dynamic slices are useful especially in cases when it is important to know the program's precise flow of execution forwards from the slicing criterion (information needs #17,22).

HyperSoft provides (static) forward slices in interprocedural form, that is, over the whole program across procedure and function borders. An example is

given in Figure 10, with the first occurrence of variable `g1` in function `f21` as the slicing criterion. The slice can be used, for instance, in analyzing what parts of the program would be affected if the value of the criterion `g1` would be modified during maintenance. As noted earlier, colors are used to represent information. In this case, the reverse color denotes nodes within a slice from which originates a link to at least one relevant program part. Note that the complete slice can be very complex. Large blocks of code can be parts of the slice. In such cases only the most important links are shown. There are, for example, links to places where `g1` or some other relevant variable obtains its value, links to the called relevant functions and links from a function to the function where it is called from. In the case of multiple (different kinds of) links originating from a node, a pop-up window will list the link type and destination alternatives. The user may choose the level of detail shown. Since the generation of this data may take much time, the user is provided with time estimates and intermediate information about the rate and status of the generation (in the window on the right).

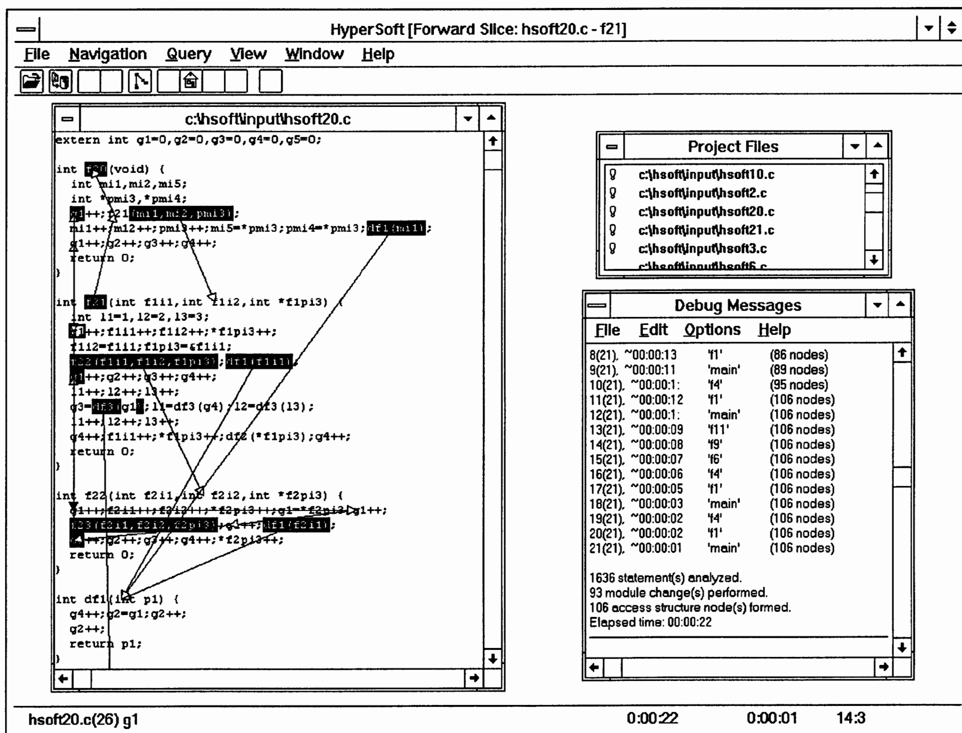


Figure 10. Forward slice.

Since complete slices for large programs may be very large, slow to generate, and therefore factually useless in practise, HyperSoft can provide them in a

partial form as well. The mechanism is the same as for call graphs: the user can indicate how many calling levels between functions the slice shall cover. The drawback of such an incomplete analysis is a possible loss of precision, because some of the relevant parts of the program might be left out of the partial slice. Another way to make large slices more comprehensible is to use the general visualization facilities of HyperSoft, and study the (forward) slices on the module or function level only.

6. CONCLUSION

In this paper we have analyzed the information needs of software maintainers as represented in the series of empirical studies by von Mayrhauser, Vans and Howe, which provide data on the information needs of professional C programmers on the most detailed and comprehensive level that is currently available. We have described how the most typical information needs relate to the forming of transient hypertextual access structures. We have represented five large access structure categories - references, lists, sets, trees, and general graphs - as a way of gathering the information satisfying the most prominent information needs.

The last column in Table 1 shows that all the access structure types (R1, L1, S1, T1, T2, G1, G2, G3) described in Section 5 are useful in finding the listed information. However, the table contains two information needs for which our approach does not seem to offer any help: "Connected domain-program-situation model knowledge" (#3) and "Naming conventions separated by systems or library objects that use them; rules for naming new procedures" (#21). Both of these needs are related to information stored at least partly in documentation. As we discussed earlier, our research so far has been focused on improving capabilities to obtain information in the source code using static program analysis. The HyperSoft model underlying our approach, however, allows for the extension of access structures to cover documentation as well, and the use of dynamic analysis for identifying needed program parts. The automatic formation is here essential in order to enable the formed documentation-based structures to be useful after program modifications.

On the basis of their information needs data von Mayrhauser and Vans [28,29] suggest certain maintenance-tool capabilities. The three most important categories are: pop-up object definitions with hypertext capabilities, cross-references with keyword search, and user-guided call graph representation. Supplementing hypertextual navigation with search would obviously be the first step towards querying, which has been investigated in depth by, for example, Paul and Prakash [36]. The formation of call graphs is user-guided within HyperSoft. Von Mayrhauser and Vans hope especially for features related to the process of comprehension (such as defining hypotheses) which current tools

frequently lack, as well as support for the different layers of their comprehension model. These capabilities could be based on recording user operations and session history and on providing hypertextual links between nodes within different fragmentations, as was discussed in Sections 4.3.4 and 4.3.5, respectively.

Our investigation into the potential usefulness of transient hypertextual access structures in software maintenance has been empirically validated in a number of studies with HyperSoft as the support tool [23,24,35]. These studies have clearly shown the practical value of the access structures currently provided in HyperSoft. However, as indicated in this paper, the tool can significantly be further improved by extending the sources of the hypertextual access structures to documentation and run-time information, and by making it possible to flexibly combine multiple sources and to navigate the resulting combined access structures.

REFERENCES

[1] M. Agosti and J. Allan, "Introduction to the special issue on methods and tools for the automatic construction of hypertext", *Information Processing & Management*, vol. 33, no. 2, pp. 129-131, 1997.

[2] A.V. Aho, R. Sethi, and J. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] V. Arunachalam and W. Sasso, "Cognitive processes in program comprehension: an empirical analysis in the context of software reengineering", *The Journal of Systems and Software* vol. 34, pp. 177-189, 1996.

[4] J. Bigelow, "Hypertext and CASE", *IEEE Software,* vol. 5, no. 2, pp. 23-27, 1988.

[5] D. Binkley and K.B. Gallagher, "Program slicing", *Adv. Comput.,* vol. 43, pp. 1-50, 1996.

[6] I.T. Bowman, R.C. Holt, and N. Brewster, "Linux as a case study: its extracted software architecture", *Proc. 21st Int. Conf. Software Engineering (ICSE'99),* pp. 555-563. Los Angeles, CA, IEEE Computer Society Press, 1999.

[7] K. Brade, M. Guzdial, M. Steckel and E. Soloway, "Whorf: a hypertext tool for software maintenance", *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, no. 1, pp. 1-16, 1994.

[8] R. Brooks, "Towards a theory of the comprehension of computer programs", *Int. J. Man-Machine Studies*, vol. 18, no. 6, pp. 543-554, 1983.

[9] G. Canfora, A. Cimitile, U. De Carlini, and A. De Lucia, "An extensible system for source code analysis", *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 721-740, 1998.

[10] Y.-F. Chen, M, Nishimoto, and C. Ramamoorthy, "The C information abstraction system", *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 325-334, 1990.

[11] E. Chikofsky, and J. H. Cross II, "Reverse engineering and design recovery: a taxonomy", *IEEE Software*, vol. 7, no. 1, pp. 13-17, 1990.

[12] J. Conklin, "Hypertext: an introduction and survey", *Computer*, vol. 20, no. 9, pp. 17-41, 1987.

[13] J. Cybulski, and K. Reed, "A hypertext-based software-engineering environment", *IEEE Software*, vol. 9, no. 2, pp. 62-68, 1992.

[14] I. Duncan and D. Robson, "An exploratory study of common coding faults in C programs", *J. of Software Maintenance: Research and Practice*, vol. 8, no. 4, pp. 241-256, 1996.

[15] M. Eisenstadt, "My hairiest bug war stories", *Comm. ACM*, vol. 40, no. 4, pp. 30-37, 1997.

[16] J. French, J. Knight, and A. Powell, "Applying hypertext structures and software documentation", *Information Processing & Management*, vol. 33, no. 2, pp. 219-231, 1997.

[17] P. Fritzson, T. Gyimothy, M. Kamkar and N. Shahmehri, "Generalized algorithmic debugging and testing", *Proc. ACM SIGPLAN'91 Conf. Compiler Construction*, pp. 317-326, ACM Press, 1991

[18] G. Gannod and B. Cheng, "Using informal and formal techniques for the reverse engineering of C programs", *Proc. Int. Conf. Software Maintenance (ICSM'96)*, pp. 265-274, 1996.

[19] C. Goldfarb, *The SGML Handbook*, ed. Y. Rubinsky, Oxford Univ. Press, 1990.

[20] M. Harrold and B. Malloy, "A unified interprocedural program representation for a maintenance environment", *IEEE Transactions on Software Engineering*, vol. 19, no. 6, pp. 584-593, 1993.

[21] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering", *Proc. 14th Int. Conf. on Software Engineering (ICSE'92)*, pp. 392-411, ACM Press, 1992.

[22] B. Kernighan and D. Ritchie, *The C Programming Language*, (2nd ed.), Prentice Hall (Software Series), 1988.

[23] J. Koskinen, "Empirical evaluation of hypertextual information access from program text", *Proc. 7th Int. Workshop on Program Comprehension (IWPC'99)*, pp. 162-169, IEEE Computer Society, 1999.

[24] J. Koskinen, "Empirical evaluations of hypertextual information access from program text", Computer Science and Information Systems Reports, Working paper WP-36, University of Jyväskylä, Jyväskylä, Finland, 1999 (Submitted for publication).

[25] S. Letovsky and E. Soloway, "Delocalized plans and program comprehension", *IEEE Software*, vol. 3, no. 3, pp. 41-49, 1986.

[26] P. Linos, P. Aubet, L. Dumas, Y. Helleboid, P. Lejeune, and P. Tulula, "CARE: an environment for understanding and re-engineering C programs", *Proc. Conf. on Software Maintenance (ICSM'93),* pp. 130-139, IEEE Computer Society Press, 1993.

[27] A. von Mayrhauser and A.M. Vans, "Program comprehension during software maintenance and evolution", *Computer*, vol. 28, no. 2, pp. 44-55, 1995.

[28] A. von Mayrhauser and A.M. Vans, "Industrial experience with an integrated code comprehension model", *Software Engineering Journal*, vol. 10, no. 5, pp. 171-182, 1995.

[29] A. von Mayrhauser and A.M. Vans, "Program understanding needs during corrective maintenance of large scale software", *Proc. 21st Annual Computer Software & Applications Conference (COMPSAC'97)*, pp. 630-637, Washington, D.C., IEEE Computer Society Press, 1997.

[30] A. von Mayrhauser and A.M. Vans, "Program understanding during software adaptation tasks", *Proc. Int. Conference on Software Maintenance (ICSM'98)*, pp. 316-325, Bethesda, Maryland, IEEE Computer Society Press, 1998.

[31] A. von Mayrhauser, A.M. Vans, and A.E. Howe, "Program understanding behaviour during enhancement of large-scale software", *J. of Software Maintenance: Research and Practice*, vol. 9, pp. 299-327, 1997.

[32] G. Murphy, D. Notkin, and E. Lan, "An empirical study of static call graph extractors", *Proc. 18th Int. Conf. Software Engineering (ICSE'96)*, pp. 90-100, IEEE Computer Soc. Press, 1996.

[33] K. Nørmark and K. Østerbye, "Representing programs as hypertext", *Nordic Workshop on Programming Environment Research (NWPER'94*), LU-CS-TR: 94-127, pp. 11-24, eds: B. Magnusson, G. Hedin, and S. Minör, Lund University, Lund, Sweden, 1994.

[34] J. Paakki, J. Koskinen, and A. Salminen, "From relational program dependencies to hypertextual access structures", *Nordic J. Computing*, vol. 4, no. 1, pp. 3-36 (Special issue on Programming Environments), 1997.

[35] J. Paakki, A. Salminen, and J. Koskinen, "Automated hypertext support for software maintenance", *The Computer J.*, vol. 39, no. 7, pp. 577-597, 1996.

[36] S. Paul and A. Prakash, "A query algebra for program databases", *IEEE Transactions on Software Engineering*, vol. 22, no. 3, pp. 202-217, 1996.

[37] R.S. Pressman, *Software Engineering - A Practitioner's Approach*, (3rd ed.), McGraw-Hill, 1992.

[38] J.-P. Queille, J.-F. Voidrot, N. Wilde, and M. Munro, M. (1994), "The impact analysis task in software maintenance: a model and a case study", *Proc. Int. Conf. Software Maintenance (ICSM'94)*, 1994.

[39] V. Rajlich and S. Varadarajan, "Using the web for software annotations", *Int. J. Software Engineering and Knowledge Engineering*, vol. 9, no. 1, pp. 55-72, 1999.

[40] R. Riecken, J. Koenemann-Belliveau, and S. Robertson, "What do expert programmers communicate by means of descriptive commenting", *Empirical Studies of Programmers - 4th Workshop (ESP'91)*, eds: J. Koenemann-Belliveau, T. Moher, and S. Robertson, Ablex, 1991.

[41] A. Salminen, J. Tague-Sutcliffe, and C. McClellan, "From text to hypertext by indexing", *ACM Transactions on Information Systems*, vol. 13, no. 1, pp. 69-99, 1995.

[42] A. Salminen, and C. Watters, "A two-level structure for textual databases to support hypertext access", *J. American Society for Information Science*, vol. 43, no. 6, pp. 432-447, 1992.

[43] M. Shepherd, C. Watters, and Y. Cai, "Transient hypergraphs for citation networks", *Information Processing & Management*, vol. 26, no. 3, pp. 395-412, 1990.

[44] S. Tilley, S. Paul, and D. Smith, "Towards a framework for program understanding", *Proc. 4th Workshop on Program Comprehension (IWPC'96)*, pp. 19-28, eds: A. Cimitile, and H. Muller, IEEE Computer Soc. Press, 1996.

[45] F. Tip, "A survey of programming slicing techniques", *J. Programming Languages*, vol. 13, no. 3, pp. 121-189, 1995.

[46] C. Watters and M. Shepherd "A transient hypergraph-based model for data access". *ACM Transactions on Information Systems*, vol. 8, no. 2, pp. 77-102, 1990.

[47] M. Weiser, "Programmers use slices when debugging", *Comm. ACM*, vol. 25, no. 7, pp. 446-452, 1982.

[48] J. Welsh and M. McKeag, *Structured System Programming*, Prentice Hall (International Series in Computer Science), N.J., USA, 1980.

[49] N. Wilde, A. Chapman, and R. Richardson, "The extensible dependency analysis tool set: a knowledge base for understanding industrial software", *Int. J. Software Engineering and Knowledge Engineering*, vol. 4, no. 4, pp. 521-534, 1994.

[50] P. Wright, "Cognitive overheads and prostheses: some issues in evaluating hypertexts", *Proc. 3rd ACM Conf. Hypertext*, pp. 1-12, ACM Press, 1991.

[51] W. Yang, "Identifying syntactic differences between two programs", *Software - Practice and Experience*, vol. 21, no. 7, pp. 739-755, 1991.

[52] K. Østerbye, "Literate Smalltalk programming using hypertext", *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 138-145, 1995.

VII

EVALUATIONS OF HYPERTEXT ACCESS FROM C PROGRAMS

Koskinen, J. 1999. Submitted (July 1999) and conditionally accepted (Jan. 2000) to be published in *Journal of Software Maintenance: Research and Practice.*

(C) 1999 John-Wiley & Sons Ltd. Reproduced with permission.

An earlier version of the paper has been published as: Koskinen, J. 1999. *Empirical Evaluations of Hypertextual Information Access from Program Text.* University of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Working paper WP-36.

The results of the 1st experiment has been published as: Koskinen, J. 1999. Empirical evaluation of hypertextual information access from program text. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99).* IEEE Computer Soc., 162-169.

EVALUATIONS OF HYPERTEXT ACCESS FROM C PROGRAMS

Jussi Koskinen

Department of Computer Science and Information Systems
University of Jyväskylä
P.O. Box 35, SF-40351 Jyväskylä, Finland
Email: koskinen@cs.jyu.fi

Abstract

Transient hypertextual access structures (THASs) are temporary graphs formed automatically on the basis of the situation-dependent information needs of software engineers. The approach is implemented in the HyperSoft system, which is a hypertext-based software maintenance support tool. THASs highlight the relevant parts of the program and enable nonlinear browsing between them. The system also supports various graphical views whose elements are linked to the program text. The paper describes the effects of using these hypertextual structures in two separate experiments. The subjects of both experiments were computer science students (total N=70). In both experiments, the subjects performed a series of sample information accesses from a C program. HyperSoft and conventional text browsing and searching were compared. The results from the two experiments are well in line with our hypothesis of the usefulness of the approach and with each other. The results indicate significantly better task performance while using THASs as compared to using the information seeking capabilities of a conventional compiler environment.

Keywords: CASE (Computer Assisted/Aided Software Engineering), hypertext, software maintenance, reverse engineering, program comprehension

1 Introduction

The readiness to recognize the importance of *software maintenance* and program comprehension has improved during the few recent years partly because of the potentially great effects of the Y2K problems; see *e.g.* (Marcoccia, 1998). The maintenance problems are often most severe in case of large, undocumented legacy systems. *Program comprehension* (Brooks, 1983; Letovsky & Soloway, 1986; von Mayrhauser & Vans, 1995) is a process which aims to enhance the level of knowledge about issues which are important to the fulfillment of

programming and maintenance tasks. The relation between software maintenance and program comprehension is such that typically program comprehension is needed in order to perform software maintenance tasks successfully. Although many kinds of complex software maintenance tasks and ways to categorize them exist, at the basic level most of them require information access from the program text in order to be performed.

HyperSoft system has been constructed as a proof of the concept of transient hypertext support for software maintenance and to allow us to test the effects of different transient hypertextual access structures (THASs). HyperSoft supports the user by providing various THAS types and view types. THASs are formed automatically on the basis of situation-dependent information needs and are transient, which means that they are not stored permanently but are available only during one session. HyperSoft has been constructed in co-operation with three large Finnish software companies: Nokia (Research Center), Novo Group and Tieto corporation. This paper summarizes the results of the empirical testing of the constructed system within the University of Jyväskylä using computer science students as subjects. The study consisted of two separate experiments (1st experiment: N=23, median=4th year students, 2nd experiment: N=47, median=2nd year students). The 2nd experiment was basically an iteration of the 1st experiment. The results of the 1st experiment were published in (Koskinen, 1999). Within the 2nd experiment, the subjects were not the same and the task-set was varied. This paper summarizes the results of both experiments.

The paper is organized as follows: first the nature of THAS-based software maintenance support and the HyperSoft system are outlined in Sections 2 and 3, then the general layout of the experiments is described in Section 4. Section 5 represents the results of the empirical study, and Section 6 summarizes the paper.

2 Background and THAS-based information access

Hypertext (Conklin, 1987) is text with nonlinear browsing capabilities. Hypertext consists of text fragments called nodes, and links connecting these nodes. The usefulness of hypertext is often motivated by asserting that it complements the more traditional global search techniques with local navigation (Nielsen, 1990) based on the linkage that it provides and by that it provides an open, exploratory information space (Nielsen, 1989).

A qualitative synthetic review of quantitative experimental studies on the use of hypertext is provided by Chen and Rada (1996). Based on the analyzed 23 studies they have made the following main observations: generally, hypertext appears to enhance performance (although there is a wide variance among the studies, partly because of the different "benchmarks", different level of sophistication of the provided features and different kind of experimental designs), hypertext appears to benefit the users more in case of open tasks (which demand *e.g.* browsing and are typically more complex than closed ones,

e.g. simple searches), the effect of cognitive styles appears to be small and the effect of spatial abilities great. Users clearly benefit from the graphical overview maps. Another meta-analysis of experimental studies on hypertext is (Nielsen, 1989). There exists comparisons of: hypertext vs (linear) text (McKnight *et al.*, 1990; Lehto *et al.*, 1995), hypertext navigation vs scrolling (Monk *et al.*, 1988), hypertext navigation vs boolean search (Dimitroff & Wolfram, 1995; Wildemuth *et al.*, 1998), and searching vs browsing (Rada & Murphy, 1992; Qiu, 1993) in information retrieval. In most systems, hypertext is created manually, although some efforts have been made to form hypertext automatically (Furuta *et al.*, 1989; Agosti & Allan, 1997). The findings of Dimitroff & Wolfram (1995) and Tebbutt (1999) suggest that the inclusion of automatically generated semantic links between documents speeds the location of information, when compared with the use of search engine alone. The application areas of the above mentioned systems are varied.

Software hypertext systems support program maintenance activities by allowing the user to create links between the source code and the related documentation (Bigelow, 1988; Cleveland, 1989; Garg & Scacchi, 1990; Brown, 1991; Cybulski & Reed, 1992; Brade *et al.*, 1994; Østerbye, 1995; Oinas-Kukkonen, 1997). The tool most related to the HyperSoft system is WHORF (Brade *et al.*, 1994) which is a hypertext tool for maintenance of C programs targeted at supporting the recognition of delocalized program plans based on as-needed strategy via multiple, concurrent views of software. The aims of HyperSoft and WHORF are similar in the following regards: both support C language, apply hypertext explicitly, aim to support the users related to as-needed program comprehension strategies and delocalized program plans, and apply multiple representations which are linked to each other. WHORF does not include program slicing. The evaluation of WHORF usage as compared to using paper documentation suggests that the applied approach is useful for accessing information from software. Related research also includes program slicing (Weiser, 1982; Binkley & Gallagher, 1996) and reverse engineering tools for C language (Chen *et al.*, 1990; Linos *et al.*, 1993; Wilde *et al.*, 1994; Gallagher, 1996). For example, CARE (Linos *et al.*, 1993) is a re-engineering tool for C programs, maintaining a repository of entities and relations (control-flow and data-flow dependencies). CARE focuses on visualization and supporting incremental modifications of programs. Nowadays, many commercial tools also provide some sort of hypertextual browsing capabilities, e.g. Cygnus (Red Hat, 2000), Discover (SET, 2000), Logiscope (Verilog, 2000), and Sniff+ (TakeFive, 2000).

Transient hypertextual access structures (THASs) (Koskinen *et al.*, 1994; Koskinen, 1996; Koskinen 1997) are automatically formed, temporary data structures enabling hypertextual browsing capabilities for program text. They are based on *transient hypergraph datamodels* (Watters & Shepherd, 1990; Shepherd *et al.*, 1990; Salminen & Watters, 1992). THASs are graphs consisting of pieces of program text called *nodes*. The nodes are connected by hypertextual *links* enabling nonlinear text browsing. The links are formed on the basis of the program dependencies which have been described and classified (Paakki *et al.*, 1997).

The general idea of THASs is to help software engineers to focus their attention on the relevant program parts. Hypertext makes explicit the structure of the text and the dependencies between its components, thus supporting its investigation. Because many different THAS types can be formed, the text can be viewed from many different points of view within a single paradigm of information representation. Since THAS types can be tailored to meet the requirements of specific maintenance tasks, they can effectively aid in focusing the attention of the user thus sparing mental resources and ensuring that all the relevant program components are noted. In particular, THASs can help to avoid the tedious work related to the efforts to comprehend undocumented legacy systems. Many of the most important concepts in program comprehension theories are congruent with the nature of THAS-based maintenance support. THASs are especially suited to helping the user to obtain information related to situations where delocalized program plans (Letovsky & Soloway, 1986) are involved, since via hypertext it is possible to quickly examine the dependencies between various components. Systematic strategies of program comprehension are supported by providing a way to browse through the relevant program components. Hypertextual nodes may serve as beacons for further browsing efforts.

3 The HyperSoft system

The HyperSoft system (Salminen *et al.*, 1994; Paakki *et al.*, 1996; Koskinen, 1997) can be characterized as a reverse-engineering tool. It supports ANSI-C (Kernighan & Ritchie, 1988) and embedded SQL and runs under MS-Windows 3.1/95/NT. HyperSoft provides various THAS and view types which can be used as aids in comprehending programs. THASs are formed automatically by the tool. For large programs manual formation of many of the hypertextual structures would be impossible or impractical. The HyperSoft system has already been evaluated within our partner enterprises, although with a relatively low number of subjects, and received positive feedback on its usefulness (Paakki *et al.*, 1996). This study aimed to systematically evaluate different aspects of the system with a larger number of subjects.

Figure 1 represents part of an example THAS as shown within the HyperSoft system. The shown representation is the basic hypertextual view used within HyperSoft. There is one THAS whose nodes belong to three modules. The three windows contain some of the C source code related to three modules of the program used within the evaluation experiments. The example THAS is a partial forward call graph initiated from the function identifier *find_moves* (top-left window). The THAS contains a node for each call and implementation of the functions reachable from the *find_moves* function by following function calling dependency. The highlighted text blocks represent the hypertextual nodes and arrows the links between the nodes. Since (in case of the basic hypertextual view) the information belonging to a THAS is

represented within the original text, the understanding of the context and surroundings of the nodes is enhanced, compared to the situation in which there would be only a disconnected additional view.

The example THAS can be traversed in various nonlinear ways by following the links provided by the system. Selection of a node causes the cursor to move to the appropriate target/destination node (and to change the active module when due). In the case of multiple links originating from a node, a pop-up window for making a selection is shown (as in the bottom-left pane of Figure 1). The graphical representations of linkages are optional. Navigation is supported via various mechanisms, including the home node link, back-tracking, history lists and abstract, graphical views.

The HyperSoft system supports the following THAS types: (a) occurrence lists for variables and functions, (b) call graphs (showing both the forward and backward calling dependencies), and (c) static program slices. The formation of a program slice is initiated by specifying a *slicing criterion,* which is - in HyperSoft - a variable occurrence at a specified program point. There are two variants of program slices: a backward slice contains the statements which may have effect on the value of the slicing criterion, and a forward slice contains the statements which may be affected by the value of the slicing criterion.

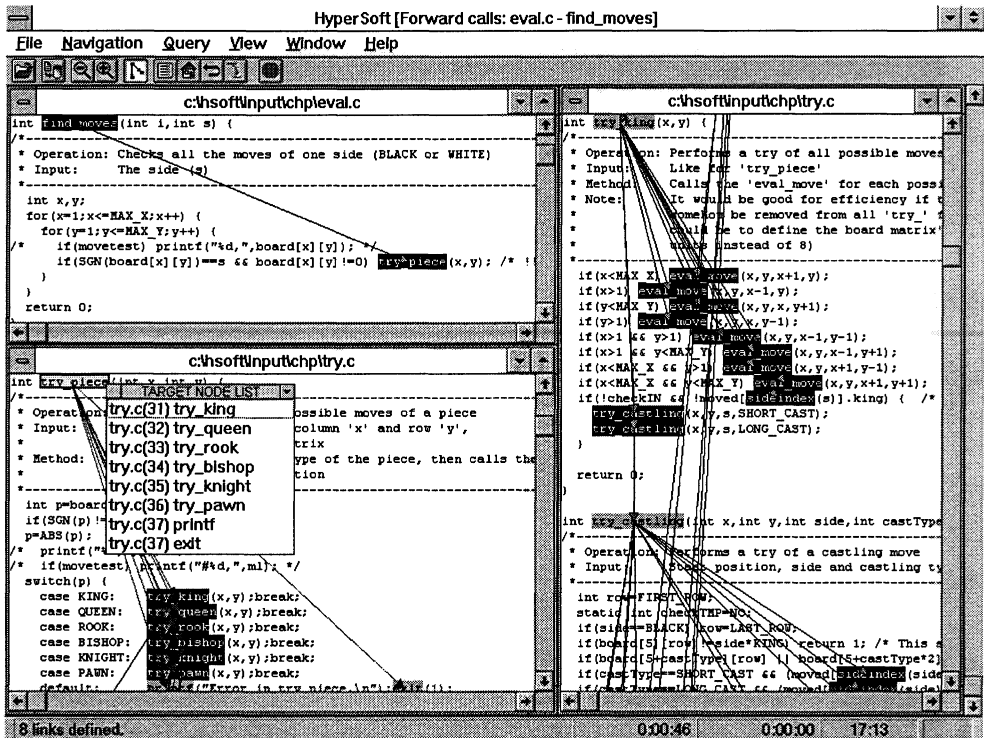


FIGURE 1 Visualization of forward call graph THAS in the HyperSoft system

The importance of graphical views is noted *e.g.* by Linos *et al.* (1993), Chen and Rada (1996), and Gallagher (1997). HyperSoft provides various views of the program text and of the THASs formed. The components appearing within the graphical views are linked such that from within them the user can directly move to the corresponding place within the basic hypertextual view (and thus program text). The applied views include: (a) the basic hypertextual view, (b) a structured map view for hierarchic examination of a THAS showing the modules, functions and places of the nodes within a THAS, and (c) a function dependency view, showing the dependencies between the functions of a THAS graphically. Figure 2 describes the views. The figure shows a project file window at right, which can be used in moving to the beginning of each module within the active project, a structured map view on top and a function dependency view at bottom. The views show part of the example THAS.

The THAS generation requires one preliminary phase - generation of the so-called static program database for the specified project. The database consists of a parse tree and symbol table with information about the positions of the important program components. This phase is needed only once, or after changes have been made to a certain module, for that module. Since extreme sizes of well-designed modules are atypical, this does not represent a serious threat to the upscaling of the approach.

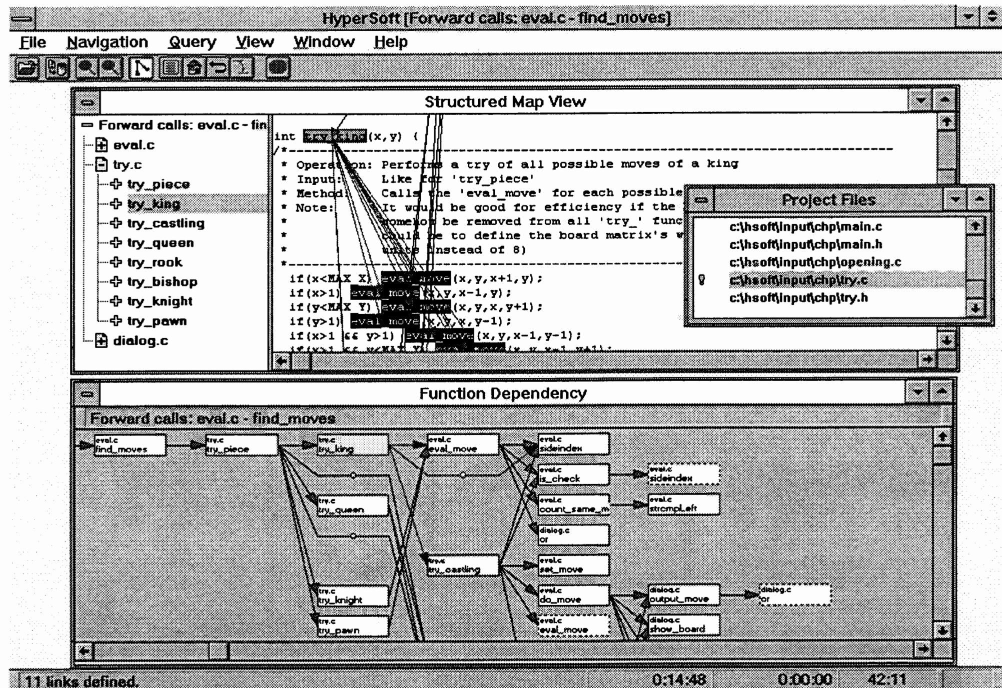


FIGURE 2 Views of the example THAS within the HyperSoft system

4 The experiments

In this section we describe the general layout of the experiments. We chose a laboratory experiment (instead of field testing) since this strategy makes it easier to obtain sufficient number of participating subjects, a reasonable level of control over the various aspects of the experiment, and repeatability; see (Shneiderman, 1986; Benbasat, 1987). Control over the experiment is important in order to enable a systematic evaluation of the different kind of software maintenance situations and related information requests.

There were two separate experiments, which will be called as *1st experiment* and *2nd experiment*. The experiments aimed to cast light on the relative usefulness of transient hypertext support (the HyperSoft system) when compared to the conventional, compiler environment generally used (Borland C/C++) for seeking information from the source code. We organized a classical experimentation using a control group and a test task. Related to both experiments, there were two groups of subjects: HyperSoft group (which used the Borland environment for performing the test task and the HyperSoft system for the actual tasks) and control group (which used the Borland environment for all tasks).

4.1 Context and procedure of evaluation

The environments compared were Borland C/C++ 5.02 for MS-Windows and HyperSoft 1.0. The operating system was MS-Windows NT Workstation 4.0. The technical environment consisted of identical P-II, 350 MHz PCs with 128 MB RAM.

The 1st experiment was conducted in November 1998 and the 2nd experiment in February 1999. The subjects of the 1st experiment were 23 computer science students at the University of Jyväskylä (median: 4th year students). The subjects of the 2nd experiment were similarly 47 computer science students (median: 2nd year students). The 1st experiment was conducted as part of an advanced computer science course ('Software Production') and the 2nd experiment as part of a (2nd year) course ('Software Engineering'). Both experiments were conducted as part of the demonstration sessions of the courses. The students participated voluntarily in the experiments. As an incentive, the subjects obtained a point to the final examination. Assurances of confidence were targeted to the subjects. The background of the subjects is detailed in Section 4.2.

The subjects were assigned to the HyperSoft groups (12 and 24 subjects) and control groups (11 and 23 subjects), which were intended to be similar in regard to the subjects' relevant maintenance skills and experience. The assignment was done primarily on the basis of the results of a test task. The purpose of the test task was to provide information about the capabilities of the subjects in the situation and consequently to reduce the uncontrolled effects of variability in performance among them; see *e.g.* (DeMarco & Lister, 1985). The test task was performed by using the Borland environment and without

HyperSoft. The HyperSoft group performed the actual tasks solely with the aid of HyperSoft and the control group solely with the aid of the Borland environment's basic editor capabilities (text browsing and search). HyperSoft does not currently apply color-shades to high-light different kind of syntactical structures, as the Borland-environment does, nor does HyperSoft contain a search function, which reasons may cause a slight bias in favor of Borland.

The subjects were provided with a questionnaire and a one-page instruction sheet. Instructions for the use of HyperSoft were minimal, consisting of a 1-hour demonstration session, a one-page list of the basic functions of the system and general information about the relevant THAS types and view types related to the tasks. Before the experiments, the students were adviced to read through the distributed instructions.

The information about the relevant types was given since we aimed to evaluate the usefulness of transient hypertext support in case where the users are properly acquainted with it, which is the normal case in the actual use of any support environment. The exact starting point of a task was provided so that everyone would be able to start the experiment from the same line. Both groups participated in the HyperSoft tutorial and the same information was made available. To both the objective was to maximize the correctness of the answers to the questions posed within the available time. The tasks were completed one at a time. The way that HyperSoft was (instructed to be) used was simple and consisted of the following phases:

(a) searching for the task-related point in the program,
(b) selecting the specified THAS type,
(c) waiting for the generation of the THAS to be completed,
(d) possibly selecting a view type,
(e) using the formed THASs and views to solve the task.

The answer was typically a list of names of variables, functions or modules. After the task was solved both groups evaluated its difficulty. There were five missing values (out of 207) related to the difficulty of the 1st experiment, which were supplemented by the median of the variable for the task and the group that the subject belonged to.

The HyperSoft group also evaluated the usefulness of the HyperSoft features used in accomplishing a task. The subjects were individually informed of the time they had taken to complete each task, which was then written on the questionnaire. The time needed to form the THASs was included in the time-values of the HyperSoft group. Since the time required to generate the static program database was only 6.3 seconds, its generation had no visible effect on the time estimates discussed later in this paper.

4.2 Independent variables

The competence of the subjects was measured by their experience on the basis of completed studies, programming work experience, and other relevant factors. The variables related to the studies included: major-subject, year course, credit units (cu:s), computer science cu:s, programming language course cu:s, and programming language course grades. In addition to programming work

experience, other relevant factors included experience with tools (Borland C/C++, HyperSoft, other reverse-engineering tools), the language (C), the application domain (chess, the chess-program to be comprehended), and the general operating system environment (MS-Windows).

Tables 1-2 gather the background of the subjects. The values given are averages within the groups. Table 1 shows the information related to relevant studies and work experience for the two series and the groups. Table 2 shows the information about the factors of direct importance to the fulfillment of the tasks. In Table 2, the values shown are averages of the estimates given by the subjects on the scale 0-5, with the exception that the lower values in the Borland and HyperSoft columns indicate experience in hours. The HyperSoft and control groups were relatively even in regard to the distribution of the values of the background variables in both experiments. 51 of the subjects were majoring information systems science, 15 information technology, and 4 other sciences.

TABLE 1 The studying and working background of the subjects

Experim.[1] /Group[2]	N	Total cu:s	Computer science cu:s	Program-ming course cu:s	Program-ming course grade (max=3)	Programming work experience (weeks)
E1/HS	12	100	51	14,3	2,48	19,3
E1/BC	11	103	59	15,6	2,60	18,9
E2/HS	24	62	25	5,8	2,09	2,3
E2/BC	23	70	29	6,8	2,01	1,9

TABLE 2 Other experience factors relevant to the experiment[3]

Experim.[1] /Group[2]	Tools			Lang uage	Application domain		Oper. system	Total weighted experience
	Bor-land	Hyper Soft	RE-tools	C	Chess	Chess '93	MS-WIN	
E1/HS	2,58 120 h.	1,00 1 h.	0,75	3,25	2,58	0,25	4,42	0,53
E1/BC	3,27 139 h.	1,00 1 h.	0,82	3,55	3,00	0,64	4,45	0,59
E2/HS	2,29 59 h.	1,00 1 h.	0,37	2,25	1,92	0,13	4,25	0,27
E2/BC	2,00 93 h.	1,00 1 h.	0,43	2,22	1,61	0,00	4,30	0,26

[1] E1 refers to the 1st experiment and E2 to the 2nd experiment.

[2] HS refers to the groups using HyperSoft, and BC to the groups using Borland C/C++.

[3] The values represented are the averages of the estimates given by the subjects on the scale 0-5, with the exceptions that 1) the lower values in Borland and HyperSoft columns for each group show the experience in hours, and 2) the last column shows the calculated, weighted average experience values for the groups.

In case of the 1st experiment the subjects' average grade for programming courses was 2.5 (maximum 3), which means that the subjects were talented. In case of the 2nd experiment the grade was 2.0, which means that those subjects were average. All groups had 1 hour experience on the use of the HyperSoft system (the tutorial). Average experience on the use of the Borland environment was about 130 hours (1st experiment) and about 75 hours (2nd experiment).

An additional variable was derived from the background information on skills relevant to the performance of the experimental tasks. The variable labelled *experience* was calculated via weighting programming work experience (1/3 of the weight), programming courses with their grades (1/3), and situationally important characteristics of experience (1/3). Within the situational component, experience on the Borland environment, C, and the program to be comprehended in particular were weighted. The experience values were 0.53 (1st experiment) and 0.27 (2nd experiment) for the HyperSoft group and 0.59 (1st experiment) and 0.26 (2nd experiment) for the Borland group.

4.3 Dependent variables

A model of the variables to be explained is given in Figure 3. Usefulness of the tool was modelled by the effect on the efficiency of maintenance task performance, subjectively felt difficulty of a task (which is parallel to the needed effort), and subjectively felt usability of the tool. Since efficiency, usability and difficulty are not commensurable, their combined effect on usefulness can only be estimated at non-quantitative level. Difficulty and usability were measured according to the subjects' subjective statements, on a six-point scale (0-5). The usability information was gathered from the HyperSoft group. In addition, we gathered information about the HyperSoft system and its features (applied THAS and view types).

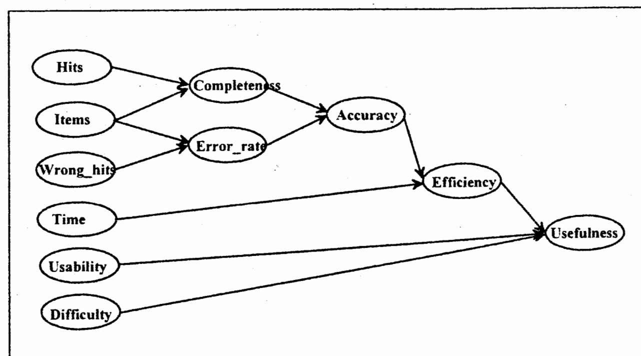


FIGURE 3 Model of the variables to be explained

Of the above-mentioned three estimates, the first is the most important. The task performance efficiency is affected by the time needed to complete a task

and correctness of the conclusions drawn. The correctness of the performed tasks was judged according to how they conformed to the predefined answers (Hits in Figure 3 representing the number of correct answers). The derived variables were calculated as follows:

Completeness= Hits / Items (to be found),
Error rate= Wrong hits / Items (to be found),
Accuracy= Completeness - Error rate,
Efficiency= Accuracy / Time.

4.4 The target program

The sample program was a non-commercial chess program (Chess'93) consisting of 5 C-modules and 2 user-defined header files. The size of the program was about 2,700 LOC. For each of the program's functions, there were comments at its beginning describing its operation, purpose, input, and output. There was no other documentation and the system was not familiar to the subjects before the experiment, so in that sense the program resembled a legacy system.

4.5 The information requests and tasks

In order to be able to evaluate the correctness of the answers that subjects provide, the tasks need to be strictly defined and have objectively correct solutions. Therefore, we decided to use elementary information requests as tasks. More complex software maintenance tasks are inevitably composed of elementary information requests. The information request types to be performed were selected according to their relative importance (von Mayrhauser & Vans, 1995) and on the availability of their support via the HyperSoft system. From the information needs listed by von Mayrhauser and Vans (1995) the following were covered by the requests: "call graph display" (forward call graph THAS), "list of routines that call a specific routine" (backward call graph THAS), "location and uses of identifiers" (occurrence list, slices), "count of variable use" (occurrence list), and "list of browsed locations" (all THASs). The information requests and tasks are detailed in the Appendix. The task set consisted of a test task (which was the same for the both experiments), eight actual tasks related to the 1st experiment and five actual tasks related to the 2nd experiment.

There were 4 demonstration groups related to the 1st experiment and 6 demonstration groups related to the 2nd experiment. Within each demonstration group the subjects were alloted as evenly as possible to the HyperSoft group and to the control group. In case of the 2nd experiment, group 1 (10 subjects) was unable to accomplish the tasks T2.4 and T2.5 and group 3 (7 subjects) the task T2.5 within the available time. Moreover, in case of the 2nd experiment, due to the fact that the subjects were novices, three more complex tasks could not be accomplished within the available time.

The objective of all of the tasks was to find correct component (name)s from within the C source code. The test task (*cf.* Section 4.1) was to find the functions which are called from the main function and implemented within specified modules. Two of the actual tasks dealt with searching for variable occurrences (T1.1, T2.1), two with searching function occurrences (T1.5, T2.2), three with dataflow analysis (T1.4, T1.8, T2.5), and six with following calling dependencies (T1.2, T1.3, T1.6, T1.7, T2.3, and T2.4).

4.6 Expected results

Since the general idea behind the THAS-based maintenance support is congruent with many of the issues of program comprehension theories and the current THAS set of the HyperSoft system with the empirical findings of the information needs of professional software maintainers (von Mayrhauser & Vans, 1995), our hypothesis was that transient hypertext support would have a positive effect on the task performance. Suggestions of the usefulness of the approach had already been received in form of subjective estimates of professional software maintainers (Paakki *et al.*, 1996).

Our 1st experiment mainly confirmed our initial hypothesis. Since our 1st experiment did show only an almost significant difference between the groups related to used time, we assumed that the 2nd experiment probably would not reveal a more significant difference, since the subjects of the 2nd experiment were less experienced. The difference in the rate of correct solutions should be high if the subjects were able to use the system on the basis of the provided instruction.

5 Results and discussion

In this section we will represent the general results of the two experiments, combined results, and task-wise results. We will also discuss on the experiments and the interpretation of the results and propose further research areas.

5.1 General results

The main results of both experiments are presented in Tables 3a and 3b. The tables show the values of the actual dependent variables as averages within the groups (based on the actual tasks, E1 denoting the 1st experiment, E2 the 2nd experiment, HS the HyperSoft group, and BC the Borland group), the values of the scaled test variables *test1* and *test2*, and the significance values of a (1-tailed, independent samples) Student's *t*-test and Mann-Whitney U test for the differences in the test variables between the groups. All the variances between the groups are in favor of HyperSoft's usefulness.

TABLE 3a Summarized results of the differences of performance between the groups

		Efficiency		Accuracy		Completeness	
Exper.[4]		E1	E2	E1	E2	E1	E2
Actual[5]	HS[6]	0,26	0,20	0,73	0,76	0,81	0,76
	BC	0,10	0,07	0,32	0,41	0,63	0,44
test1[7]	HS	0,21	0,16	0,17	0,43	0,23	0,41
	BC	0,04	0,02	-0,26	-0,03	-0,01	-0,00
test2[8]	HS	0,52	0,69	1,45	2,72	1,62	2,73
	BC	0,17	0,26	0,58	1,47	1,17	1,59
t-test[9]	*(test1) t*	4,39	7,22	2,66	4,50	1,87	4,61
	df	21	29	21	29	21	29
	p	0,000	0,000	0,008	0,000	0,038	0,000
		(***)	(***)	(**)	(***)	(*)	(***)
M-W.[10]	*(test1) N*	23	31	23	31	23	31
	p	0,000	0,000	0,009	0,000	0,040	0,000
		(***)	(***)	(**)	(***)	(*)	(***)
t-test[11]	*(test2) p*	0,000	0,000	0,000	0,000	0,013	0,000
		(***)	(***)	(***)	(***)	(*)	(***)

TABLE 3b Summarized results of the differences of performance between the groups

		Error rate		Used time		Difficulty	
Exper.[4]		E1	E2	E1	E2	E1	E2
Actual[5]	HS[6]	0,08	0,01	2,90	4,07	0,92	2,35
	BC	0,31	0,03	3,63	5,70	1,64	3,80
test1[7]	HS	0,06	-0,02	-6,59	-5,31	-1,21	-0,65
	BC	0,24	0,02	-5,75	-3,47	-0,77	0,44
test2[8]	HS	0,04	0,00	1,51	1,14	0,79	0,64
	BC	0,18	0,01	2,10	1,65	1,30	1,06
t-test[9]	*(test1) t*	-3,67	-1,37	-1,98	-2,82	-1.33	-3,04
	df	21	29	21	29	21	29
	p	0,001	0,091	0,031	0,005	0,238	0,003
		(***)	-	(*)	(**)	-	(**)
M-W.[10]	*(test1) N*	23	31	23	31	23	30
	p	0,000	0,001	0,067	0,004	0,330	0,003
		(***)	(***)	-	(**)	-	(**)
t-test[11]	*(test2) p*	0,001	0,002	0,022	0,025	0,027	0,002
		(**)	(**)	(*)	(*)	(*)	(**)

4 E1 refers to the 1st experiment and E2 to the 2nd experiment.

5 Values of the actual dependent variables for actual tasks as averages for the groups.

6 HS refers to the groups using HyperSoft, and BC to the groups using Borland C/C++.

7 *test1* is the value of the actual variable - the result of the test task.

8 *test2* is calculated by scaling the actual performance with the background *experience.*

9 This *t*-test compares the values of *test1* for HS and BC groups, shown are the *t* value, degrees of freedom (*df*), and risk level (*p*). $0.01 \le p < 0.05$ are almost significant results (*), $0.001 \le p < 0.01$ are significant (**), and $p < 0.001$ are highly significant (***).

10 This Mann-Whitney U test compares the values of *test1* for HS and BC groups, shown are the number of valid cases (*N*) and risk level (*p*).

11 This *t*-test compares the values of *test2* for HS and BC groups, showing the risk level (*p*).

A *t*-test is a solution to the problem of the comparison of the means of small samples. The row of *t*-test represents the significance values of the differences between the groups. The hypothesis of the equal performance among the groups can be rejected with the represented risk level (*p*). The *t*-test (in its basic form) assumes the equality of variances between the groups and normality of the distribution of the test variables. The test and remedy of the (lack of) equality of variances (Levene's test) is embedded into the applied *t*-test. For the *t*-test of the *test1* variables, Tables 3a and 3b show the value of *t*, degrees of freedom (*df*), risk level (*p*) and the significance of the differences. The interpretation for the statistical significance of the *p* values, in general, is as follows: $0.01 <= p < 0.05$ are almost significant and marked with (*), $0.001 <= p < 0.01$ are significant, marked with (**), and $p<0.001$ are highly significant, marked with (***).

Kolmogorov-Smirnov (exact, 2-tailed, one-sample) test of the normality of the distribution of the test variables was applied group-wise. The assumption of the normality can be accepted (with the normal 5% risk-level) with the only exception that in case of the 2nd experiment the distribution of the error rate is not normal for the HyperSoft group. There is neither significant differences in the values of the test variables among the groups in this sense. These results are probably due to the very low error rate of the HyperSoft group related to the 2nd experiment. Consequently, this does have only a very slight effect on the derived variables (related to which the differences are highly significant regardless of the effect of error rate).

The most important variable is the efficiency of performance. The values of this variable were 0.26 (1st experiment) and 0.20 (2nd experiment) for the HyperSoft group and 0.10 (1st experiment) and 0.07 (2nd experiment) for the Borland group. This means that the efficiency of performance of the control group was less than 50% of the efficiency of the HyperSoft group in both experiments.

There were no statistically significant differences in the values of the test task results between the HyperSoft and control groups, which suggests that the groups were similar in this sense. Due to the nature of the experiments, the most important quantity is the efficiency of performance "cleaned" with the possible initial variation among the groups, this is the *test1* value. In each column, the value of the *test1* variable has been calculated by reducing the result of the test task from the value of the actual variable. Thus, these values represent the change in performance after HyperSoft had been introduced to the groups and starting performing the actual tasks. The actual *test1* values are dependent on the differences between the test task and the actual tasks and the availability of the HyperSoft. The differences of *test1* values between the groups is dependent only on the availability of the HyperSoft (since the tasks were the same to the groups). The general performance efficiency values that were scaled this way were 0.21 (1st experiment) and 0.16 (2nd experiment) for the HyperSoft group and 0.04 (1st experiment) and 0.02 (2nd experiment) for the Borland group.

The differences found between the groups were as follows: efficiency of performance (1st experiment: $p=0.000$ (***); 2nd experiment: $p=0.000$ (***)),

accuracy (p=0.008 (**); 0.000 (***)), completeness (p=0.038 (*); 0.000 (***)), error rate (1st experiment: p=0.001 (***)), time used (p=0.031 (*); 0.005 (**)), and difficulty of performing a task (2nd experiment: p=0.003 (**)). Generally, the differences are clearer in case of the 2nd experiment. The unexperience of the HyperSoft group with the tool has affected the used time. The clear difference in time used in case of the 2nd experiment was not quite expected. The unexpectedly clear difference in efficiency in case of the 2nd experiment suggests that especially novice programmers benefit from the HyperSoft system.

As an additional verification of the differences, we have applied a (exact, non-parametric) Mann-Whitney U test for *test1* variables for testing the significance of differences of the means between the groups. The rows show the number of valid cases (N) and the risk-level (p) as in case of a t-test. The results of this test are well in line with the results of the t-test, except that the (exact, 1-tailed) Mann-Whitney test additionally revealed a highly significant difference in case of error rate of the 2nd experiment; p=0.001 (***), which the t-test could not recognize.

In Tables 3a and 3b *test2* refers to a value calculated by scaling the actual performance with the value of the *experience* variable (refer to Section 4.2), thus also aiming to eliminate the effects of varying experience among the groups. This test was applied merely to further increase the reliability of the results. The normality of the distribution of *experience* was tested by using (exact, 2-tailed) Kolmogorov-Smirnov test groupwise, independently for the two series, which confirmed the normality. It should be noted that *test2* approximates the effect of experience by assuming that task performance, in general, is a linear function of relevant experience (the results of the test task support well this hypothesis; see Figure 4). The *test2* results are in line with the above described results, except that a significant difference was found in *test2* value related to the error rate of the 2nd experiment, which could not be found based on *test1*. Within the 1st experiment, a statistically highly significant difference between the groups was found related to error rate. The partial failure of duplicating that result in the 2nd experiment is probably due to the very low error rate within the 2nd experiment, which in turn is most likely due to the relative easiness of the actual tasks. All the t-tests were performed as 1-tailed, since we assumed - based on the feedback received during the initial evaluations (Paakki *et al.*, 1996) (and related to the 2nd experiment, based on the results of the 1st experiment) - that HyperSoft has positive effect on the measured variables, and the means of those variables between the groups show the direction of the variance to be such.

5.2 Combined results

Figure 4 shows the results of the test-task (HyperSoft not in use). The figure also shows the relation between the relevant background experience and task performance in case of the test task. In Figures 4-6, triangles represent the subjects within the HyperSoft groups and the spheres the subjects within the

Borland groups. The figures summarize the results of the two experiments. The larger symbols represent the subjects of the 1st experiment (who were more experienced). Figure 4 shows that there is no significant differences in average efficiency values among the groups in case of the test task.

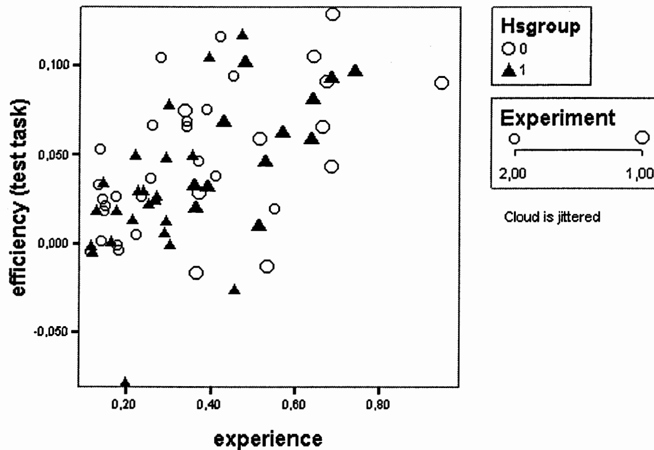


FIGURE 4 Efficiency of performance, results of the test task

Figure 5 shows the actual values of efficiency for the both experiments and groups (as averages of the actual tasks). Finally, Figure 6 shows the relation between the use of the HyperSoft and the "cleaned" performance efficiency estimate (*test1* for efficiency). The figure also shows the relation between the relevant background experience and the efficiency of performance.

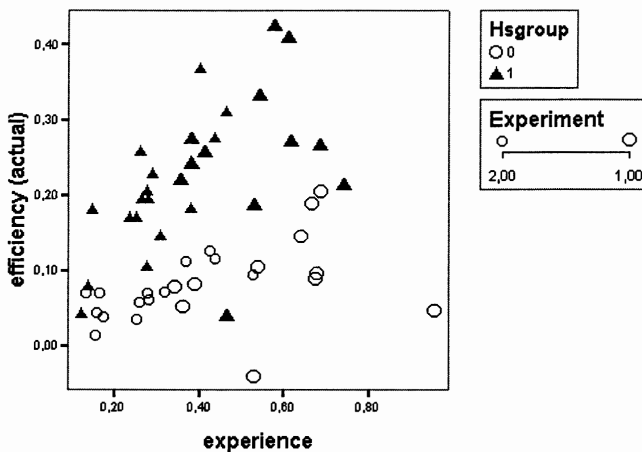


FIGURE 5 Efficiency of performance, results of the actual tasks

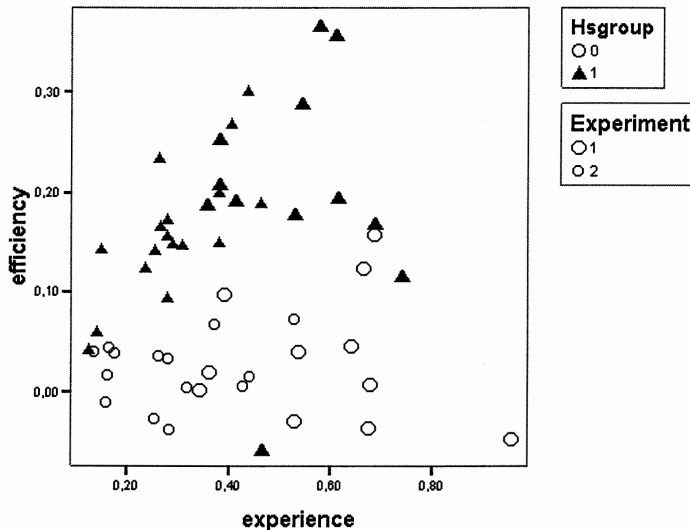


FIGURE 6 "Cleaned" efficiency of performance, results of the actual tasks

The normality of the distribution of *experience* and the test task efficiency were tested by using (exact, 2-tailed, one-sample) Kolmogorov-Smirnov test groupwise for the combined data, which confirmed the normality. The received negative efficiency values represent the cases where subjects performed the test task better than the actual tasks. It can be noted graphically from Figure 6 that experience has bigger effect on the results of the HyperSoft group than that of the control group. It can also be noted that, at the same experience levels, the results of the weakest subjects within the HyperSoft groups are - in most cases - better than those of the best subjects within the control groups.

Figure 7 shows the variance of *test1* efficiency as a box-plot. Efficiency is shown for the four categories formed based on the two experiment series (1st experiment=1, 2nd experiment=2) and the group (control=0; at left, HyperSoft=1; at right). It can be noted graphically from the figure that both the experience (the subjects of the 1st experiment were more experienced) and HyperSoft use have positive correlation with the amount of variance of efficiency. The spheres represent the extreme cases.

A two-way variance analysis was performed for the combined data in order to find out the differences that the experiment (1st/2nd) may have caused to the performance efficiency. The model consisted of the group (whether a subject belonged to the HyperSoft group or to the control group), and series (the 1st/2nd experiment series) as factors and *experience* as a covariate. Neither the interaction effect of group and series (p=0.367), nor the series (p=0.842) or experience (p=0.227) were statistically significant. The best explanatory factor was the group: F=60.8; N=53; p=0.000 (***). The explanatory power (determination coefficient) was 52.8%, which is a good rate for a variance analysis with a single explaining variable. The variance analysis assumes

normality of the distribution of the dependent variables and equality of variances within the groups. These requirements were tested and met group-wise.

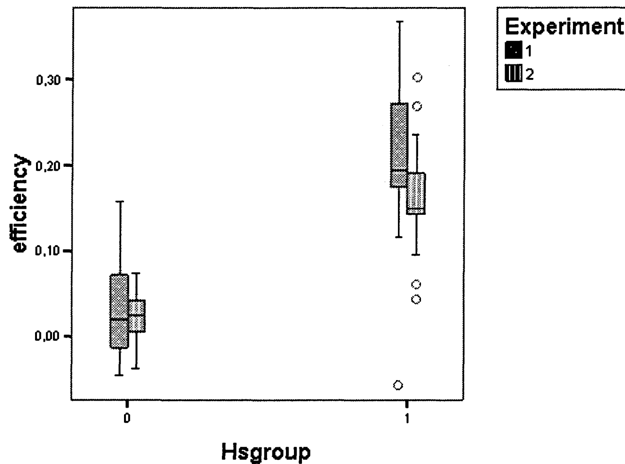


FIGURE 7 Variance of efficiency related to the experiment series and HyperSoft use

The reliability of the test of the differences within the combined data requires that the difficulty of the tasks within the two experiments would not differ significantly. This was tested by (exact, 2-tailed) Mann-Whitney test. First the combined data was divided into two groups of the same size based on the result of the test task (the test task was the same for the both experiments). Then, separately for the formed groups (for the 'good ones' and the 'poor ones') was applied the Mann-Whitney test to find out the possible differences between the experiments. There were no significant differences between the experiments (p=0.395 within the 'good ones' and p=0.802 within the 'poor ones'). The general performance efficiency values (*test1*) were 0.18 (for the combined HyperSoft group) and 0.03 (for the combined Borland group). The results of a t-test for the differences between the groups were: t=7.65; df=52; p=0.000 (***) for the *test1* and t=8.03; df=42.8; p=0.000 (***) for the *test2*. Basically, these t-tests represent another way to test the same thing as in the case of the above-described variance analysis.

5.3 Task-wise results

The results were as follows. Table 4 summarizes the task-wise, significant differences in the efficiency of performance between the groups. Similarly, the following tables summarize the differences related to accuracy (Table 5), completeness (Table 6), error rate (Table 7), time used (Table 8), and difficulty (Table 9). The tasks are detailed in the Appendix.

TABLE 4 Task-wise differences of efficiency between the groups

Efficiency		T1.1[12]	T1.2	T1.4	T1.7	T2.1	T2.2	T2.3	T2.4	T2.5
Actual	HS[15]	0.12	0.63	0.73	0.33	0.24	0.43	0.24	0.32	0.07
[13]	BC	-0.28	0.19	0.33	0.00	0.01	0.16	0.10	0.08	0.03
t-test[14]	*test1 t*	3.16	2.76	2.87	3.20	6.37	5.80	7.61	8.42	3.31
	df	21	12.2	15.7	16.2	44	45	44	35	30
	p	0.005	0.017	0.011	0.005	0.000	0.000	0.000	0.000	0.001
		(**)	(*)	(*)	(**)	(***)	(***)	(***)	(***)	(***)

TABLE 5 Task-wise differences of accuracy between the groups[16]

Accuracy		T1.1	T1.7	T2.1	T2.2	T2.3	T2.4	T2.5
Actual	HS	0.33	0.67	0.75	0.89	0.78	0.86	0.45
	BC	-1.21	0.05	0.05	0.50	0.51	0.44	0.29
t-test	*test1 t*	3.18	2.26	6.50	4.41	4.45	5.58	2.23
	df	21	21	44	45	44	35	30
	p	0.005	0.034	0.000	0.000	0.000	0.000	0.017
		(**)	(*)	(***)	(***)	(***)	(***)	(*)

TABLE 6 Task-wise differences of completeness between the groups[16]

Completeness		T1.1	T1.2	T1.7	T2.1	T2.2	T2.3	T2.4	T2.5
Actual	HS	0.75	0.86	0.67	0.79	0.89	0.78	0.86	0.46
	BC	0.39	0.58	0.18	0.17	0.52	0.51	0.44	0.31
t-test	*test1 t*	2.12	2.09	2.84	7.30	4.48	4.22	5.81	2.19
	df	21	21	21	44	45	44	35	30
	p	0.046	0.049	0.010	0.000	0.000	0.000	0.000	0.019
		(*)	(*)	(**)	(***)	(***)	(***)	(***)	(*)

TABLE 7 Task-wise valid differences of error rate between the groups[16]

Error rate		T1.1	T2.1
Actual	HS	0.42	0.04
	BC	1.61	0.13
t-test	*test1 t*	-3.13	-2.40
	df	21	44
	p	0.005	0.011
		(**)	(*)

[12] T1.x refers to the tasks of the 1st experiment and T2.x to the tasks of the 2nd experiment.

[13] Values of the actual dependent variable for actual tasks as averages for the groups.

[14] This *t*-test compares the values of *test1* for HS and BC groups, shown are the *t* value, degrees of freedom (*df*), and risk level (*p*). $0.01 \le p < 0.05$ are almost significant results (*), $0.001 \le p < 0.01$ are significant (**), and $p < 0.001$ are highly significant (***).

[15] HS refers to the groups using HyperSoft, and BC to the groups using Borland C/C++.

[16] The meanings of the used acronyms are the same as in Table 4.

TABLE 8 Task-wise differences of time used between the groups[16]

Time used		T1.2	T1.7	T2.3	T2.4
Actual	HS	2.19	2.96	3.51	2.85
	BC	3.77	4.88	5.45	5.69
t-test	*test1 t*	-2.25	-3.91	-2.08	-3.68
	df	21	21	44	35
	p	0.035	0.001	0.022	0.001
		(*)	(***)	(*)	(**)

TABLE 9 Task-wise differences of task difficulty between the groups[16]

Difficulty		T2.1	T2.2	T2.3	T2.4	T2.5
Actual	HS	2.38	1.83	2.17	1.78	3.75
	BC	3.55	2.96	4.05	4.00	4.93
t-test	*test1 t*	-2.03	-2.01	-3.24	-3.45	-1.94
	df	44	45	42	34	29
	p	0.025	0.026	0.001	0.001	0.032
		(*)	(*)	(**)	(**)	(*)

In the 2nd experiment, the differences in efficiency were highly significant related to all actual tasks. The differences in accuracy in the 2nd experiment also were very clear and statistically highly significant, except in the case of task T2.5. The task-wise results of completeness are rather similar as the results of accuracy. Related to error rate *t*-test could reveal the differences only in case of tasks T1.1 and T2.1. This is partly due to the abnormality of the test distribution. Since both of the significant differences appear as first one of a series, the result should be interpreted with some caution. The HyperSoft group did more errors than the control group in task T1.2, where no additional view was specified. This underlines the importance of views.

Differences in time used and difficulty of tasks are less clear than in case of other dependent variables. The differences in time used are, however, clear in case of the most difficult tasks; T1.7 and T2.4, which suggests that tasks containing complex searches benefited most from HyperSoft in this sense. Some of the tasks (T1.5, T1.6, T1.7) had only few predefined correct answers. These kinds of tasks were used in testing the capabilities of the subjects to end the search, without wasting time on further speculations. The differences between the groups in tasks T1.5 and T1.6, which were relatively easy, were small. The variances in the data-flow/slicing tasks (T1.4, T1.8, T2.5) were not above the average. This is probably because the slicing tasks were easy (intraprocedural slicing) and partly due to technical problems (task T1.8).

The above mentioned reasons reduced the overall variance in performance between the groups. The difference was weakest in the case of task T1.3. This is probably because structured map view is not very well suited for representing calling dependence information (*cf.* task T1.7). This underlines the importance of using proper views.

5.4 Other results

The average usability values of the HyperSoft system as judged by the subjects (the HyperSoft groups) were 3.91 (1st experiment, based on 252 received elementary answers) and 3.82 (2nd experiment, based on 180 received elementary answers) on a six-point scale; 0-5 (where 0 represents useless and 5 extremely useful). It is reasonable to assume that the point of comparison was at least partly the Borland-environment, since it was used to complete the test task.

The combined, weighted usability values for THAS types and view types were as follows: occurrence lists (3.85), call graphs (3.89), slices (3.56), structured map views (4.08) and function dependency views (4.39). These usability results suggest that the usability of the HyperSoft system was sufficient, so that the weaknesses of some of the tool characteristics have not seriously, negatively affected the results.

5.5 General remarks, limitations and further research options

The received results suggest that the HyperSoft approach can leverage task performance as compared to ordinary text browsing and search. The benefits are probably proportional to the complexity of the tasks. The more complex a task is, the more prominent will be the related mental overhead (and cognitive complexity), which can be reduced through process automation. In particular tasks in which great certainty about the correctness of the outcomes is necessary (*e.g.* certainty that all the instances are found) can benefit from the HyperSoft approach. It is probable that in case of more complex information requests, more complex tasks and longer use (longer and more "wearing" sessions), the usefulness of the HyperSoft would be even more evident. Functions which HyperSoft contains and which are specially tailored for the more complex situations (but which were not used within these experiments) include: use of multiple THASs, formation of subprojects, and editor integration. Further studies would need to be conducted in order to find out more about the variances in cases of very complex or slightly differing kinds of information requests.

WHORF (Brade *et al.*, 1994) is in many regards similar to HyperSoft. Thus, the results received from the evaluation of HyperSoft are relevant to the development of tools like WHORF as well. Since, WHORF has been evaluated only with a small program (250 LOC) and only as compared to using paper documentation, the results with HyperSoft complement the results received from the use of WHORF. The best evaluated related tool is CARE (Linos *et al.*, 1993) (with 2,000 LOC program, N=40, comparing the features of the tool). Their observations of the important features are mostly taken into account within the implementation of the HyperSoft system: there are links from the graphical views to the basic hypertextual views (and thus to the program text), hypertextual nodes are represented as highlighted elements within the program text and program slices (which are considered as useful) are supported. It

should be noted that views are an essential part of the HyperSoft approach. The importance of a search function complementing browsing has been noted by (Halasz, 1988; Linos *et al.*, 1993; Storey *et al.*, 1997). HyperSoft currently does not include a search function, which obviously would be a simple but important additional feature.

One important aspect affecting the results of the evaluations is the experience on the use of the tool. The way of using the HyperSoft system is simple. However, related to more complex tasks, there is also elevated need for becoming familiar with the tool features. The subjects had only 1 hour experience on the use of HyperSoft. The effect of this kind of lack of tool experience could be reduced by organizing a course on the subject, before the use of the tool or by performing longitudinal studies on the tool usage as suggested by Chen and Rada (1996).

The experiments aimed to gather data on the usefulness of the HyperSoft approach as one kind of reverse engineering technique. Comparisons between different ways of reverse engineering were not performed, except that different THAS and view types were used and compared. It should also be noted that the results related to the used time represent the way that it can be reduced related to information seeking/comprehension tasks solely. Programming and maintenance naturally includes also other kind of activities, most notably code modifications, which take time.

6 Summary

Transient hypertextual access structures are automatically formed temporary graphs satisfying the situation-dependent information needs of software maintainers. The HyperSoft system is an implementation of the approach. We evaluated HyperSoft empirically in two separate experiments with computer science students as subjects. We compared performance between groups using HyperSoft and groups using the information seeking capabilities of the Borland C/C++ compiler environment (text browsing and searching). We measured task performance in sample information requests related to a sample C-program. The variables by which the performance was modelled were efficiency, accuracy, completeness, error rate, time used, and the subjectively felt difficulty of the tasks. In addition, we gathered information about the usability of the HyperSoft system and its various features. The results support our hypothesis about the usefulness of the HyperSoft system and of the transient hypertext support for software maintenance. In general, the subjects using the HyperSoft system were able to find more complete answers to the posed questions and to perform the tasks more efficiently and in less time than the control groups.

Acknowledgments

We wish to thank Lecturer, Doctor Annaliisa Kankainen, and Lecturer Anna-Liisa Lyyra from the Statistics Department, for their methodological guidance, Professor Airi Salminen for commenting on the manuscript, and Professor Markku Sakkinen and Professor Jari Veijalainen for their positive attitude towards the evaluation related to their courses. The study was made possible by funding from COMAS (Jyväskylä Graduate School in Computing and Mathematical Sciences) and by the participation of the computer science students.

References

Agosti, M. & Allan, J. 1997. Introduction to the special issue on methods and tools for the automatic construction of hypertext. *Information Processing & Management* 33 (2), 129-131.

Benbasat, I. 1987. Laboratory experiments in information systems studies with a focus on individuals: a critical appraisal. In R.J.Jr. Boland & R.A. Hirschheim (Ed.) *Critical Issues in Information Systems Research.* John-Wiley.

Bigelow, J. 1988. Hypertext and CASE. *IEEE Software* 5 (2), 23-27.

Binkley, D. & Gallagher, K. 1996. Program slicing. *Advances in Computers* 43, 1-50.

Brade, K., Gudzial, M., Steckel, M. & Soloway, E. 1994. Whorf: a hypertext tool for software maintenance. *International Journal of Software Engineering and Knowledge Engineering* 4 (1), 1-16.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18 (6), 543-554.

Brown, P. 1991. Integrated hypertext and program understanding tools. *IBM Systems Journal* 30 (3), 363-392.

Chen, C. & Rada, R. 1996. Interacting with hypertext: a meta-analysis of experimental studies. *Human-Computer Interaction* 11 (2), 125-156.

Chen, Y.-F., Nishimoto, M. & Ramamoorthy, C. 1990. The C information abstraction system. *IEEE Transactions on Software Engineering* 16 (3), 325-334.

Cleveland L. 1989. A program understanding support environment. *IBM Systems Journal* 28 (2), 324-344.

Conklin, J. 1987. Hypertext: an introduction and survey. *Computer* 20 (9), 17-41.

Cybulski, J. & Reed, K. 1992. A hypertext-based software-engineering environment. *IEEE Software* 9 (2), 62-68.

DeMarco, T. & Lister, T. 1985. Programmer performance and the effects of the workplace. In *Proc. 8th Int. Conf. on Software Engineering (ICSE'85).* IEEE Computer Soc. Press, 268-272.

Dimitroff, A. & Wolfram, D. 1995. Searcher response in a hypertext-based bibliographic information retrieval system. *Journal of the American Society for Information Science* 46 (1), 22-29.

Furuta, R., Plaisant, C. & Shneiderman, B. 1989. A spectrum of automatic hypertext constructions. *Hypermedia* 1 (2), 179-195.

Gallagher, K.B. 1997. Visual impact analysis. In *Proc. Int. Conf. on Software Maintenance (ICSM'96)*, 52-58.

Garg, P. & Scacchi, W. 1990. A hypertext system to manage software lifecycle documents. *IEEE Software* 7 (3), 90-98.

Halasz, F. 1988. Reflections on Notecards: seven issues for the next generation of hypermedia systems. *Communications of the ACM* 31 (7), 836-855.

Kernighan, B. & Ritchie, D. 1988. *The C Programming Language* (2nd ed.). Englewood-Cliffs: Prentice Hall.

Koskinen, J. 1996. Creating transient hypertextual access structures for C programs. In M. Kavanaugh (Ed. production) *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering*. Los Alamitos, CA: IEEE Computer Soc., 56-65.

Koskinen, J. 1997. *HyperSoft: Back-end Components*. Univ. of Jyväskylä, Jyväskylä, Finland. Computer Science and Information Systems Reports, Technical Report TR-17.

Koskinen, J. 1999. Empirical evaluation of hypertextual information access from program text. In B. Werner (Ed. production) *Proc. 7th Int. Workshop on Program Comprehension*. IEEE Computer Soc., 162-169.

Koskinen, J., Paakki, J. & Salminen, A. 1994. Program text as hypertext - using program dependences for transient linking. In *SEKE'94: Proc. 6th Int. Conf. on Software Engineering and Knowledge Engineering*. Skokie, IL: Knowledge Systems Institute (KSI), 209-216.

Lehto, M., Zhu, W. & Carpenter, B. 1995. The relative effectiveness of hypertext and text. *International Journal of Human-Computer Interaction* 7 (4), 293-313.

Letovsky, S. & Soloway, E. 1986. Delocalized plans and program comprehension. *IEEE Software* 3 (3), 41-49.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., & Tulula, P. 1993. CARE: an environment for understanding and re-engineering C programs. In *Proc. Int. Conf. on Software Maintenance (ICSM'93)*. IEEE Computer Soc., 130-139.

Marcoccia, L. 1998. Building infrastructure for fixing the year 2000 bug: a case study. *Journal of Software Maintenance: Research and Practice* 10 (5), 333-352.

von Mayrhauser, A. & Vans, A.M. 1995. Industrial experience with an integrated code comprehension model. *Software Engineering Journal* 10 (Sept.), 171-182.

McKnight, C., Dillon, A. & Richardson, J. 1990. A comparison of linear and hypertext formats in information retrieval. In R. McAleese & C. Green (Ed.) *Hypertext: State of the Art*. Oxford: Intellect, 10-19.

Monk, A., Walsh, P. & Dix, A. 1988. A comparison of hypertext, scrolling and folding as mechanisms for program browsing. In D. Jones & R. Winder (Ed.) *People and Computers IV*. Cambridge Univ., 421-435.

Nielsen, J. 1989. The matters that really matter for hypertext usability. In F. Halasz & N. Meyrowitz (Ed.) *Proc. ACM Conf. on Hypertext: Hypertext'89*. ACM Press., 239-248.

Nielsen, J. 1990. The art of navigating through hypertext. *Communications of the ACM* 33 (3), 296-310.

Oinas-Kukkonen, H. 1997. Towards greater flexibility in software design systems through hypermedia functionality. *Information and Software Technology* 39 (6), 391-397.

Paakki, J., Koskinen, J. & Salminen, A. 1997. From relational program dependencies to hypertextual access structures. *Nordic Journal of Computing* 4 (1), 3-36.

Paakki, J., Salminen, A. & Koskinen, J. 1996. Automated hypertext support for software maintenance. *The Computer Journal* 39 (7), 577-597.

Qiu, L. 1993. Analytical searching vs browsing in hypertext information retrieval systems. *Canadian Journal of Library & Information Science* 18 (4), 1-13.

Rada, R. & Murphy, C. 1992. Searching versus browsing in hypertext. *Hypermedia* 4 (1), 1-30.

Red Hat. 2000. Cygnus Source Navigator v. 4.5. Product information available in www-form at <URL: http://www.redhat.com/products/cygnus. html>. Company: Red Hat. Description: a reverse engineering tool. Date: 10-Mar-00.

Salminen, A., Koskinen, J. & Paakki, J. 1994. HyperSoft: an environment for hypertextual software maintenance. In B. Magnusson, G. Hedin & S. Minör (Ed.) *NWPER'94: Proc. Nordic Workshop on Programming Environment Research*, Lund Univ., Lund, Sweden. LU-CS-TR: 94-127, pp. 25-37.

Salminen, A. & Watters, C. 1992. A two-level structure for textual databases to support hypertext access. *Journal of the American Society for Information Science* 43 (6), 432-447.

SET. 2000. Discover. Product information available in www-form at <URL: http://www.setech.com/products>. Company: SET Inc. Description: a reverse engineering tool supporting many languages, including ANSI C/C++. Date: 10-Mar-00.

Shepherd, M., Watters, C. & Cai, Y. 1990. Transient hypergraphs for citation networks. *Information Processing & Management* 26 (3), 395-412.

Shneiderman, B. 1986. Empirical studies of programmers: the territory, paths, and destinations. E. Soloway & S. Iyengar (Ed.) *Empirical Studies of Programmers*. Ablex.

Storey, M.-A., Wong, K. & Muller, H. 1997. How do program understanding tools affect how programmers understand programs. In P. Storms (Ed. production) *Proc. 4th Working Conf. on Reverse Engineering (WCRE'97)*. IEEE Computer Soc., 12-21.

TakeFive. 2000. Sniff+. Product information available in www-form at <URL: http://www.takefive.com/products/sniff+.html>. Company: TakeFive software. Description: a reverse engineering tool supporting many languages, including C and C++. Date: 10-Mar-00.

Tebbutt, J. 1999. User evaluation of automatically generated semantic hypertext links in a heavily used procedural manual. *Information Processing & Management* 35 (1), 1-18.

Verilog. 2000. Logiscope. Product information available in www-form at <URL: http://www.csverilog.com/products/logiscop.htm>. Company: Verilog. Description: a reverse engineering tool supporting 80+ languages. Date: 10-Mar-00.

Watters, C. & Shepherd, M. 1990. A transient hypergraph-based model for data access. *ACM Transactions on Information Systems* 8 (2), 77-102.

Weiser, M. 1982. Programmers use slices when debugging. *Communications of the ACM* 25 (7), 446-452.

Wilde, N., Chapman, A. & Richardson, R. 1994. The extensible dependency analysis tool set: a knowledge base for understanding industrial software. *International Journal of Software Engineering and Knowledge Engineering* 4 (4), 521-534.

Wildemuth, B., Friedman, C. & Downs, S. 1998. Hypertext vs boolean access to biomedical information: a comparison of effectiveness, efficiency, and user preferences. *ACM Transactions on Computer-Human Interaction* 5 (2), 156-183.

Østerbye, K. 1995. Literate Smalltalk programming using hypertext. *IEEE Transactions on Software Engineering* 21 (2), 138-145.

APPENDIX Information requests

The information requests performed were as follows. There was one test task and eight actual tasks related to the 1st experiment and five actual tasks related to the 2nd experiment. Given below are the question, the HyperSoft features to be used (the THAS types and view type were applicable), the time needed by the HyperSoft system to generate the THAS and the number of hypertextual nodes within it (parentheses) and the correct solution to the tasks (curly brackets).

(T0) Test-task: Which of the functions are called directly from *main* (*main.c*/line 135) and are implemented either in *opening.c* or *eval.c*?
{read_opening, find_kings, is_check, do_move, find_opening, find_moves, sort_moves, find_max, undo_move}

(T1.1) In which functions the variable *ml* (*main.c*/line 116) is used?
Occurrence list/ Structured map (3 sec. / 94 n.)
{main, output_mlist, eval_move}

(T1.2) Which of the functions which are called from *do_move* (*eval.c*/line 300), are implemented in the module *eval.c*?
Forward Calls (2 sec./ 38 n.)
{side_index, is_check, move_cast_rook}

(T1.3) From which modules the function *is_check* (*eval.c*/line 187) is called from?
Backward Calls/ Structured map (6 sec./ 83 n.)
{eval.c, try.c, main.c, dialog.c}

(T1.4) Which variables inside the *main* function may be affected by the value of the variable *moves* (any of its fields) in line 330, column 10 (*main.c*)? (intraprocedural)

Forward Slice (1 Sec./ 27 n.)
{x1, y1, x2, y2, capt, check, promotion, gamevalues}

(T1.5) In which functions the function *find_moves* (*eval.c*/line 33) is used?
Occurrence list/ Structured map (1 sec./ 5 n.)
{main}

(T1.6) Which of the functions called from *try_piece* (*try.c*/line 17) calls some of the functions called from the *try_piece*?
Forward Calls/ Target node list (5 sec./ 132 n.)
{try_queen}

(T1.7) Which function(s), outside the module *try.c*, call function(s) which call the function *eval_move* (*eval.c*/line 92)?
Backward Calls/ Function dependency view (3 sec./ 58 n.)
{main, find_moves}

(T1.8) Which variables inside the function *eval_move* may have effect on the value of the variable *moves[ml][n].value* in line 177 (*eval.c*)? (intraprocedural)
Backward Slice (1 sec./ 23 n.)
{ml, n, k, value, piece, x1, kx, x2, y1, ky, y2, kingx, kingy, s, p1, board, c, gamemove, p2, check_out, capt_piece, side_index}

(T2.1) In which functions the variable *s* (*main.c*/line 112) is used?
Occurrence list/ Structured map (3 sec./ 57 n.)
{main, try_piece, try_king, read_move, xy, eval_move, do_move, undo_move, move_cast_rook}

(T2.2) In which functions appear a call of the function *sideindex* (*eval.c*/line 483)?
Occurrence list/ Structured map (1 sec./ 28 n.)
{eval_move, is_check, do_move, undo_move, move_cast_rook, find_kings, try_king, try_castling}

(T2.3) Which functions are called either directly or indirectly from *try_piece* (*try.c*/line 17)?
Forward Calls/ Function dependency view (5 sec./ 132 n.)
{try_king, try_queen, try_knight, try_pawn, try_rook, try_bishop, eval_move, try_castling, side_index, is_check, count_same_moves, cr, set_move, do_move, undo_move, output_move, show_board, strcmpLeft, move_cast_ rook}

(T2.4) Which functions either directly or indirectly call the function *side_index* (*eval.c*/line 483)?
Backward Calls/ Function dependency view (9 sec./ 123 n.)
{try_castling, find_kings, move_cast_rook, is_check, undo_move, do_move, main, game_end, eval_move, read_move, try_king, try_piece, find_moves, try_pawn, try_knight, try_bishop, try_rook, try_queen}

(T2.5) Which variables inside the function *eval_move* may have effect on the value of the variable *moves[ml][n].value* in line 182, column 8 (*eval.c*)? (intraprocedural)
Backward Slice (1 sec./ 54 n.)
{ml, n, same, x1, y1, x2, y2, k, gamemove, value, piece, kx, ky, kingx, kingy, sideindex, s, c, check_out, capt_piece, p2, nmoves, p1, board}