# Reverse Engineering: A Cognitive Approach, a Case Study and a Tool

By

Iyad Zayour

Thesis

Presented to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree

**Doctor of Philosophy (Computer Science)**

Ottawa-Carleton Institute for Computer Science
University of Ottawa
Ottawa, Ontario, K1N 6N5
Canada

# Acknowledgements

My supervisor, Tim Lethbridge, had a lot of influence on the ideas presented in this thesis. It has been a nice ride with him; we seem to agree always, my research just fitted nicely with his earlier research. He has been very supportive and helped a lot, particularly, at the level of quality of presentation and he never let me wait too long for any request.

My father, Dr. Ali, who is a professor of philosophy and psychology, gave me valuable ideas and hard to find pointers into these domains; not only throughout this thesis but through a life long education.

# Contents


Chapter 1 Introduction ..... 10

1.1 Introduction ..... 10

1.2 The reverse engineering tools approach ..... 11

1.2.1 The adoption problem ..... 11

1.2.2 The efficient tool ..... 12

1.3 The current state of RE tools ..... 13

1.4 Our research approach ..... 13

1.5 Research objectives ..... 15

1.6 Overview of the thesis ..... 16

Chapter 2 Background ..... 18

2.1 Terminology ..... 18

2.1.1 Cognition and cognitive ..... 18

2.1.2 Task analysis and modeling ..... 22

2.1.3 Software Maintenance (SM) ..... 23

2.1.4 Reverse engineering ..... 23

2.2 Main research in reverse engineering ..... 24

2.2.1 Program Analysis ..... 25

2.2.2 Slicing ..... 25

2.2.3 Plan Recognition ..... 26

2.2.4 Concept Assignment ..... 26

2.2.5 Design Recovery ..... 27

2.3 Dynamic Analysis ..... 28

2.3.1 Overview ..... 28

2.3.2 Basic activities in dynamic analysis ..... 29

2.4 Program Comprehension and cognitive models ..... 31

2.4.1 Bottom up ..... 31

2.4.2 Top Down ..... 32

2.4.3 Discussion about cognitive models ..... 32

2.4.4 The Integrated model ..... 33

2.4.5 The integrated model and tool capabilities ..... 35

2.4.6 Discussion ........ 37
2.4.7 Partial comprehension ........ 38
2.4.8 Tools and program understanding ........ 40
2.5 Methodologies to identify the problems in SM ........ 41
2.5.1 Empirical and ecological study of SM tasks ........ 41
2.6 TkSee and the work practice approach ........ 43
2.6.1 TkSee ........ 44
2.6.2 Work practices ........ 45
2.7 Examples of approaches and tools ........ 46
2.7.1 Debuggers and dynamic analysis tools ........ 47
2.7.2 IsVis ........ 47
2.7.3 Jcheck ........ 50
2.7.4 Jinsight ........ 50
2.7.5 RunView ........ 51
2.7.6 View fusion ........ 52
2.7.7 Run Time Landscape ........ 52
2.7.8 Software Reconnaissance ........ 53
2.8 Support for program execution comprehension ........ 53
2.8.1 Inter-process dynamic analysis ........ 55
2.9 Dynamic slicing ........ 57
2.9.1 Dynamic slicing and program comprehension ........ 57
2.9.2 Discussion ........ 58
Chapter 3 Justifications for our approach and case study ........ 59
3.1 Theoretical Justifications ........ 59
3.1.1 Tool adoption and the real problems of SM ........ 59
3.1.2 How to characterize SM tasks? ........ 60
3.1.3 What criteria can be used to rank the difficulties of SM tasks? ........ 62
3.1.4 Cognitive load (CL) as a criterion ........ 62
3.1.5 The notion of CL ........ 63
3.1.6 Efficiency increase and CL ........ 64
3.2 The approach ........ 65

3.2.1 The cognitive approach in summary ..... 67
3.3 Our approach at work: the case study ..... 68
3.3.1 Identification of a context ..... 69
3.3.2 SM process as we perceive it: the task view ..... 69
3.3.3 The difficulties model ..... 72
3.3.4 A framework for the explanation of difficulties ..... 76
3.3.5 Tool requirements ..... 78
3.4 Evidence and support ..... 78
3.4.1 Experiment and introspection ..... 78
3.4.2 Locating pieces of code ..... 79
3.4.3 Cognitive models and domains cross-referencing ..... 80
3.4.4 Support for the difficulties model ..... 82
3.5 Generalization for cognitively based assumptions ..... 82
3.5.1 Cognitive overload and its cost ..... 83
3.5.2 The implicit-explicit and cognitive distance ..... 85
3.5.3 Gap in perception between user and tool designer ..... 87
3.5.4 Problem solving ..... 89
3.5.5 Automated processes ..... 89
3.5.6 Increasing recognition vs. recall ..... 91
3.5.7 Facilitating meaningful encoding ..... 91
3.5.8 Generalization for code delocalization and deep nesting ..... 92
3.5.9 The gap effects ..... 93
3.5.10 Conclusion ..... 94
3.6 Generalization of the theoretical lessons ..... 95
3.6.1 Object oriented code ..... 95
3.6.2 Conclusion ..... 99
Chapter 4 Overview of DynaSee ..... 100
4.1 General functionality ..... 100
4.2 Instrumentation ..... 101
4.3 The unit of instrumentation ..... 102
4.4 DynaSee features ..... 102

4.4.1 Repetition removal ........ 103
4.4.2 Patterns ........ 104
4.4.3 Routine ranking ........ 107
4.4.4 Call tree ........ 110
4.4.5 Bookmarks ........ 113
4.4.6 Code window and TkSee ........ 114
4.4.7 Summary of features ........ 114
Chapter 5 Evaluation ........ 117
5.1 DynaSee features evaluation ........ 117
5.1.1 Trace test set ........ 117
5.1.2 Repetition removal ........ 118
5.1.3 Patterns ........ 119
5.1.4 Pattern Variations ........ 122
5.1.5 Routine ranking ........ 125
5.1.6 Combining features ........ 128
5.1.7 Conclusion ........ 130
5.2 Revisiting requirements ........ 131
5.2.1 Does DynaSee create new difficulties or overhead for interpreting the dynamic view? ........ 131
5.2.2 Does DynaSee Reduce the CL in search? ........ 132
5.2.3 Does DynaSee reduce the negative CL effects caused by the issues presented in the difficulties? ........ 132
5.2.4 Does DynaSee support domain traceability? ........ 133
5.3 Holistic evaluation ........ 133
5.3.1 Task 1: ........ 134
5.3.2 Task 2: ........ 135
5.4 Semi-holistic evaluation ........ 137
5.4.1 Task 3 ........ 137
5.4.2 Task 4 ........ 138
5.4.3 Conclusion ........ 139
5.5 User perception ........ 140

5.5.1 Debriefing results ........ 140
5.5.2 Concerns ........ 141
5.5.3 Discussion about the methods of evaluation ........ 141
5.5.4 Conclusion ........ 145
Chapter 6 Contribution, Conclusion and Future Work ........ 146
6.1 Executive summary ........ 146
6.2 Contributions ........ 148
6.2.1 Research paradigm ........ 148
6.2.2 Interface with psychology ........ 151
6.2.3 DynaSee ........ 155
6.2.4 A final note on contribution ........ 157
6.3 Conclusion & Future work ........ 158
6.3.1 About the conceptual issues ........ 158
6.3.2 Areas that need more effort ........ 159
6.3.3 About the tool ........ 160

# List of figures

Figure 1: Screen dump of TkSee: the exploration pane (left) shows symbols.. ............... 44
Figure 2: POET screen dump......................................................................... 55
Figure 3: The phases of our entire research approach. ................................................... 68
Figure 5: The SM phases in our task model ..................................................................... 71
Figure 6: A screen dump of DynaSee where the call tree is on the left ........................ 101
Figure 7: Processing phases in DynaSee ........................................................................ 103
Figure 8: The general algorithm to compute W............................................................. 109
Figure 9: DynaSee call tree, the P nodes belong to patterns while the N node ............. 111
Figure 10: Call tree collapsed to form a visual pattern.................................................. 112
Figure 12: the support for domain mapping in DynaSee............................................... 115
Figure 13: the diagonal shape of the call tree branches after W is set to 71.................. 127
Figure 14: Graph showing the compression rate after applying each feature................ 129
Figure 15: The same graph without repetition removal ................................................ 129

## Abstract

Software maintenance (SM) for large legacy systems is a very inefficient process; on average, 70% of software costs are spent on maintenance [Swanson 89]. The inefficiency of SM has been related to the difficulty comprehending software systems; therefore program comprehension is considered to be a key bottleneck of SM. Reverse engineering tools have been used to alleviate this bottleneck with lower than expected success.
We present a cognitively based approach for reverse engineering tool development. We use ideas from cognitive psychology and other disciplines to formulate the approach. We also describe a case study in which we applied the approach in a telecommunication company. The case study resulted in the development of DynaSee, a reverse engineering tool which helps software engineers analyze dynamic program information. DynaSee reads routine call traces, and provides several processing and visualization features that make the use of traces much more useful for software maintenance and program comprehension. Next, we describe and evaluate the various features of DynaSee that compress, abstract and augment traces to make them comprehensible and useful for SM tasks. Finally, based on our experience in developing DynaSee, we generalize the aspects of our findings and techniques that are based on psychology by relating them to the mainstream psychological literature and to other disciplines where similar techniques have been used.

# Chapter 1 Introduction

## 1.1 Introduction

It is well known that software maintenance (SM) is the most expensive part of the software life cycle. It is estimated that about 50-75% of software costs are spent on this phase [Swanson 89, Lientz 78]. The greatest part of the software maintenance process is devoted to understanding the system being maintained. Fjelstad and Hamlen [Fjelstad 83] report that 47% and 62% of time spent on enhancement and corrections tasks, respectively, are devoted to program comprehension

Maintenance is particularly expensive for large systems that are developed over a long period of time by generations of programmers. Such software programs are often described as "legacy systems". Legacy systems are old software systems that have stayed alive and are maintained despite their high maintenance cost because they are vital to their owners and/or users.

Part of the reason for the expense of maintaining legacy systems is that such systems are hard to understand and to learn. This is because they tend to be very large, to be poorly structured and to have little or no documentation [Tilley 96]. A software engineer (SE) typically has to spend a long time trying to find where a change should be made and to understand how the system is functioning so that his change can satisfy a requirement without creating new problems.

At a high level, this thesis deals with the problem of the high maintenance cost of large legacy systems by seeking means to facilitate the comprehension of these systems. In particular, we want to achieve that by providing a conceptual framework, in terms of an approach for RE tool design that increases their chances to be efficient and adopted. Given that comprehension is a cognitive activity, justifications and foundations of this approach will use, in addition to the software engineering literature, certain psychology literature.

## 1.2 The reverse engineering tools approach

The classical way to address the inefficiency of program comprehension has been to develop reverse engineering tools. The field of reverse engineering concerns how to extract relevant knowledge from source code and present it in a way that facilitates comprehension during SM.

Reverse engineering (RE) tools are developed with the goal of increasing the productivity of SM. However, the chain of causality from tools to increased productivity is non-trivial. MIS research deals extensively with this chain [Compeau 95]. In fact, research about this chain is even older than information science itself and goes back to research on the role of technology in productivity and on technology acceptance [Compeau 95, Davis 89].

From our perspective, we abstract this chain by assuming that to increase productivity in SM, an efficient tool has to be developed and this tool has to be adopted by its users. However, as we will discuss next, both designing an efficient tool and having a tool adopted are serious research questions.

### *1.2.1 The adoption problem*

The first problem to consider is that RE tools in general have “low adoption” by software engineers (SEs) in the industry. Despite the considerable amount of research in the field of reverse engineering, SEs still largely rely on using brute force approaches such as reading system logs and source code, or using primitive search tools like "grep" [Singer 98].

The issue of adoption is particularly relevant in the case of RE tools since their adoption is generally voluntary. Unlike most software, the users of such tools are not obliged to use them because they tend not to be necessary to complete a task. In other words, without a RE tool a software engineer would eventually be able to perform the required

maintenance. The trouble is that software engineers often do not know how much benefit tools could give them, or do not believe it will be worth their time to try out the tools.

As we said, adoption of RE tools is not something that we can cover completely; it is part of a larger and older *new technology acceptance* problem. The success of adoption is dependent on many factors often unrelated to the tool itself such as managerial, social, logistical and economic factors. Nevertheless, we want to focus on those factors related to the tools so as to design tools in a way that increases their characteristics that help in their adoption

### *1.2.2 The efficient tool*

The second problem to consider that is not totally separate from the first problem (because efficiency increases adoption) is that of designing efficient tools. The nature of the problem that the tool should solve makes the design of efficient tools a research problem by itself [Storey 97]. In general, to have an efficient tool means that some of the tasks that are done without the tool should be done with the tool with less cost.

Designing tools that increase the efficiency of tasks implies many research questions derived from the need to identify the tasks to be targeted by the RE tool. For example, how is cost in SM defined and measured? And, according to the cost definition, how do we find and identify costly tasks given the diversified and creative nature of SM?

In other words, it is the creative and implicit nature of SM that leads to many research questions. Unlike most of classical software development where the goal is to process explicit information, our task involves targeting the mental activities in humans; that is why we adopt a “cognitive” approach.

## 1.3 The current state of RE tools

Most existing RE tools belong to two camps, those that are based on static analysis of code and those that deal with graphical visualization of the software system components and their interactions. Before assessing these two camps, we note that they are both built on intuitive assumptions about how these tools will increase productivity. No systematic analysis of the factors that increase efficiency and adoption are typically considered during tool design; as Lakhotia put it, "we are not listening to the user, but we are instead building what we think the user needed." [Lakhotia 93].

In recent literature [Lethbridge 96,98], more attention to the users needs has been displayed, particularly by the study of their work practices. Lethbridge's work, however, belongs to the static program analysis camp. These tools only cover part of the information requirements of SM. Also the static analysis tools have reached their maturity in terms of possible additional research effort especially with the availability of many sophisticated commercial products.

We think that what is needed is a re-examination of the assumptions about what makes a RE tool efficient. New findings beyond the classical assumptions are necessary to leap beyond the two classical camps (visualization and static analysis) that are exhausted by now by so many tools that use the same paradigm but with different designs. These findings should relate to all influencing aspects that make a tool efficient and productive.

## 1.4 Our research approach

Unlike in computer science, rather than contributing to the solution of a problem that belongs to particular thread of research, we take a practical software engineering approach to provide a solution for an actual industrial problem. We do not restrict ourselves to any discipline or paradigm but rather we are goal oriented, seeking answers and solutions in order to reach satisfying results with respect to our goal of reducing SM cost.

Although the initial ideas for this thesis come from the difficulties experienced by the author in maintaining a large legacy system in an industrial setting, we do not begin with any assumption about how to design a productive tool. Instead, we backtrack to the basic high-level assumptions touching on human nature and behaviour.

Since we are result-oriented and some of the results we seek touch on human nature, we had to tap into psychology and certain other disciplines. For example to answer the question, "what makes a person adopt a new technology?", we had to go back to social psychology and MIS to conclude that it is the perception of positive outcome. Then later to find how to reduce memory load we had to tap into the memory literature in cognitive psychology. However, although the ideas we use come from different disciplines, they have a synergistic effect across all of our research effort to achieve the simple goal of increasing productivity in SM.

Assumptions taken about human behaviour have to affect all the steps from detecting users' problem, to generating tool requirements, to designing a tool, and finally to implementing the tool. For example, the assumption that perception is what matters in adoption caused us to target the tasks that are perceived as difficult by the SE so that facilitating these tasks will show up in term of improved perception.

Another consequence of being result oriented concerns the depth at which we deal with investigating problems. In order to report practical conclusions with industrial significance, we have not to spend too much time and effort at any particular problem or stage of the research cycle so that the cycle's end can be reached and results reported. We call this a breadth-first approach as opposed to depth-first approach. So, instead of going deep into any particular problem, we tackle all encountered problems with enough depth to have satisfying results. Yet, we keep the door open for future research on any of the problem that we dealt with.

For example, in the definition of the problem space, it is unrealistic to try to exhaustively define SM. Hence attempts that claim to define the entirety of the SM concept often end

up being oversimplifications of reality. Consequently, solutions based on such definitions rarely become useful. Lakhotia [94] writes: “While it is common practice for researchers and developers to claim that their tool is a panacea, it is equally common knowledge that this is very rarely true.”

Focusing on a limited context allows us to deeply analyze such context with an acceptable amount of effort and to achieve more certainty about the problem we are solving. Characterization is one phase of the solution cycle; spending too much time at one part may cause losing contact with the big picture of the solution.

In general, the trade-off between scientific certainty, methodology and generalization on one hand and the practical need to cover the entire research cycle, on the other hand, is evident throughout the thesis and is also revisited later on in the thesis.

## 1.5 Research objectives

The broader motivation for the thesis, at a high level, is to reduce the maintenance cost of large software systems. To contribute to this goal, we tap into the area related to the problems that stand in the way of reducing cost using reverse engineering (RE) tools; particularly we want to research methods that may increase the efficiency and adoption of RE tools. Efficiency of a tool increases the productivity of software engineers who use the tool; .i.e. the tool should let the SEs do their tasks faster and with less effort. Adoptability of a tool is about increasing its chance of being used. The quest for increasing efficiency and adoptability generates additional research questions that we need to address such as how to identify the real problems and difficulties of SM and how to rank these difficulties.

The vehicle to contribute in this area is to propose an approach for developing RE tools that incorporates guidelines, design principles and conclusions which, taken together, provide a theoretical framework for the development of efficient and adoptable RE tools. This framework that includes aspects from different disciplines will be illustrated through a case study and a RE tool that will developed within this framework. The evaluation in

the thesis will be oriented toward validating the internal assumptions of the approach and also those related to the role of approach with regards to the higher level goals.

Findings from the case study that contribute to the support of SM will be illustrated to the level where they begin to have a general utility beyond the case study. The tool will also be treated as an artefact that has a life of its own beyond its initial role as a product of the case study. All aspects that are required to produce a tool with practical value in the industry, including those unrelated to the case study or approach, will be addressed.

Finally, a special emphasis will be given to the use of psychology as a contributing science in computer science research. There is considerable interest among researchers to bring these two domains closer.

## 1.6 Overview of the thesis

The thesis begins by discussing the related literature. Then, justifications for our approach are presented, followed by the details of the approach itself. A case study, as an implementation of the approach, is described next. This includes a description of the RE tool and its features. Finally, evaluation of the tool will be presented.

**Chapter 2** is about defining the common terminology and describing the background of the main related concepts. We elaborate on the areas that are related to our approach such as dynamic analysis and program comprehension. Then we describe attempts in the literature to understand the SM process and its sources of difficulties. Examples of different existing tools and approaches will be provided.

In **Chapter 3**, we first present the theoretical justifications for our approach and defend the fundamental assumptions of this work. In the next part of the chapter, we describe the case study where we applied our approach in a realistic setting. We begin by defining a task view of how we perceive SM in the context selected by the case study. Within the problems identified in this model, we prioritise these problems and analyse them in

cognitive terms generating a difficulties model that forms the basis for the tool requirements. Possible generalization from the case study findings will also be discussed.

In **Chapter 4,** we describe DynaSee, the RE tool produced within the case study. DynaSee, among other things, is used to validate our assumptions about the users' problems and the provided solutions. The design rationale and functionality of each feature are discussed.

In **Chapter 5**, we evaluate DynaSee by measuring the success of each of its features using data analysis. The tool's effectiveness will be evaluated as a whole with respect to the high level goals in the same context from which we derived the original requirements. We also try to capture the users' perception of the usefulness of the tool and their feedback.

Finally in **Chapter 6**, we summarize the thesis, highlighting our contribution, presenting our conclusions, and discussing possible future work.

# Chapter 2 Background

In this chapter, we describe the commonly used terminology and the background of the main concepts related to our research. We survey the literature on reverse engineering in general and dynamic analysis and program comprehension specifically. We also survey various tools and approaches related to SM support.

## 2.1 Terminology

### *2.1.1 Cognition and cognitive*

Cognition is defined by Britannica [Britannica 98] to include, “every mental process that can be described as an experience of knowing as distinguished from an experience of feeling or of willing” . As such, it represents a large scope of study allowing for several interpretations.

In general, the confusion about the use of the terms cognitive and cognition relates to the existence of two disciplines that use these terms with different interpretations: cognitive psychology and cognitive science. The difference will be illustrated next based mainly on a paper by Wilson [01] .

#### *2.1.1.1 Cognitive psychology*

Cognitive psychology generally refers to the branch or paradigm of psychology that succeeded the *behaviourism* paradigm “which accepted that humans learn and act in the world as a result of stimuli which they encounter". For behaviourism, only the relationship between stimuli and response was a valid subject of argument, thus reducing the human into a black box with input (stimuli) and output (response or action).
Later, it was realised that behaviourism “has been too restrictive in the range of questions that it could address” opening the door for a new paradigm, cognitive psychology, that

“permitted arguments to postulate mental processes and intermediate representations between stimuli and actions.”

Cognitive psychology is concerned with information processing, and includes a variety of processes such as attention, perception, learning, and memory [UACSD]. It is also concerned with the structures and representations involved in cognition. The greatest difference between the approach adopted by cognitive psychologists and by the behaviourists is that cognitive psychologists are interested in identifying in detail what happen between stimulus and responses thus looking at human as a white box. With the cognitive approach, actions become related to how human *understand* the world rather than treating his action as an automatic response to a stimuli.

#### *2.1.1.2 Cognitive science*

Yet, even cognitive psychology was found to be still too limiting for providing answers to broad questions concerning mental life that require broad theories encompassing many aspects of that mental life. The limitations were mainly due to the restricting legacy of the experimental rigour of psychology. So, "the trend towards further softening the methodological demands on theory led to the development of cognitive science in the mid 70's.”

By definition, “cognitive science is concerned with the understanding of mental life, and the expression of that understanding in the form of theories and models” [Wilson 01]. It develops frameworks and models of human behaviour which can be used to provide insights into human computer interaction (HCI). So unlike cognitive psychology that deals with the mental life of human in general, cognitive science was coupled with a specific domain of cognition – HCI.

Although cognitive science was initially based on cognitive psychology, the theoretical demands of cognitive science, that went beyond its experimentally testable theories, necessitated the incorporation of findings from linguistic as well as modeling techniques

used in artificial intelligence. As such, cognitive science became a synergy of different disciplines.

*2.1.1.3 The cognitive approach and cognitive load*

By the term "cognitive" in the "cognitive approach", we aim at paralleling the concerns of cognitive psychology i.e. to look at the human as a white box, investigating the conscious mental life that takes place between the observable actions during SM. For a discussion of why we focus on the conscious parts of mental processes see sections 3.5.5 and 3.5.6. These sections explain why only conscious processes are relevant for optimization of human cognitive performance.

Our use of the term cognitive load (CL) follows the cognitive psychology view of cognition. Particularly, the view that the working memory (WM) is the primary workbench where all conscious, attention-consuming mental effort is applied [Baddely 74, 86] (see section 3.5.1.1 for an illustration of a psychological view of WM). Viewing the WM as the primary workbench makes the term CL very much equivalent to memory load. However, reducing conscious cognition to only WM may be too much of an assertion for two reasons. First, WM is a non tangible construct with no universally accepted precise definition. Second, even conscious processes interact with perception; e.g. one can solve a problem that he sees on a paper and thus it does not have to be maintained entirely in the WM (i.e. memorized). Later in the thesis, we argue that reducing the need to maintain pieces of information in memory by delegating it to the perceptual system can be helpful for general cognitive performance.

*2.1.1.4 ICS*

The cognitive psychology view of cognition, having the WM as the central processor and of limited capacity, is fundamentally different from some models of cognition in the cognitive science literature such as the Interacting Cognitive Subsystems (ICS) of Barnard [93].

ICS represents the human information processing system as a highly parallel organization with a modular structure. The ICS architecture contains a set of functionally distinct subsystems, each with equivalent capabilities specialized to deal with a different class of information representation (e.g. incoming sensory information and meanings abstracted from it). The ICS subsystems exchange representations of information directly, with no role for a central processor or limited capacity working memory.

This deviation from cognitive psychology has its reasons. ICS is concerned with low level details of perception during HCI which makes it feasible only for critical HCI situations such as air traffic control [Busse 98] . In such situations, the focus is put on the details in which the human acquires and deals with information – i.e. the sensory and perceptual information as well as their path to mental representation. As such, ICS "relates primarily to approximation over activity that occurs within the very short term dynamics of human cognition" [Barnard 93].

Such critical short term cognition makes sense for a perception-centred approach where more direct links can be made between perception and action. Accordingly, instead of dealing with general mental life, ICS provides a "framework for answering such questions as, "how many information channels can be used simultaneously?" [Wilson 01].

The use of CL in such special purpose models of cognition obviously implies different meanings than our use. This special use of CL, however, is not strong enough to invalidate our use of this term since classical cognition is about mental life and not about information flow within the perceptual subsystem. In fact, the separation of perception as a parallel unconscious process from cognition as a mental conscious process can even be seen in some cognitive science models such as that of Card [83] who, in his model of cognition, breaks human information processing into three parallel macro-level mechanisms – perception, cognition, and motor activity

### *2.1.2 Task analysis and modeling*

Task analysis and task modeling are not generic terms; they have been used with specific meanings in the literature. In general, task analysis is defined as, “the process of gathering data about the tasks people perform and acquiring a deep understanding of it” [Welie 01]. The process of structuring data and gaining insights into the data is called task modeling. As such, a task model is the product of task analysis then modeling.

According to Welie [01] task analysis and task modeling, “have always been relatively ill-defined activities, where no commonly accepted models, or representations were used.” Task modeling methods have been mainly formal. This formality “made them more powerful and at the same time less usable.” They often used textual representation and “large textual representations need to be constructed and it is often difficult to understand what exactly is being represented.” Tools that were needed to make these modeling activities feasible for practitioners hardly appeared.

#### *2.1.2.1 GOMS*

Due to the different variations used in the representation of task modeling, the term *task model* has had many interpretations. Also, specific formal methods for task modeling have been developed. GOMS [Card 83] is a well known task modeling methods. It is perhaps the most influential technique that, in addition to its followers, dominated the research on models in HCI [Welie 01]. GOMS is intended to estimate user performance based on a description of the system before the system is actually built; it is a task-based dialog description technique [Wilson 01].

GOMS helps in predicting interaction time for a specific interface with a user once a system has been designed. It also helps in highlighting some aspects of complexity for a clearly specified task.

GOMS is derived from cognitive science models of human information processing, but does not deals with the detail of processing itself, rather it only provides representation languages.

### *2.1.3 Software Maintenance (SM)*

ANSI [Chikofsky 90] defines software maintenance as: "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment". As such, this term describes a very large problem space. Our focus will be only on one of its sub activities related to program comprehension and the use of related reverse engineering techniques.

A common classification is to divide SM into Corrective, Adaptive, Perfective and Preventive [Pfleeger 98]. Another way to describe SM is to subdivide the maintenance space into sub activities; i.e. to describe a process for it. Many processes have been suggested, including the one in [Chen 90] that describes SM as the following steps:

1. Problem identification: Identifying the specific functionality to be added or modified and understanding how the new functionality is different from the old one.
2. Program understanding: Determining which program elements are involved in the functionality to be changed.
3. Program modification: Design and implementation of the changes, including assessing the impact of the changes.
4. Program revalidation
5. Redocumentation

### *2.1.4 Reverse engineering*

The most general sub domain that our work belongs to is reverse engineering. Reverse engineering is related to SM because it is the part of the maintenance process that helps to understand the software so one can make appropriate changes [Chikofsky 90].

Reverse engineering is loosely defined as the process of extracting critical information that is necessary for product maintenance and/or reengineering including design and specification [Erdos 98]. Many other definitions exist for reverse engineering: one of the

most cited is that of Chikofsky [90]: "Reverse engineering is the process of analysing a subject system to identify the system's components and their inter-relationships and to create representations of the system in another form or at a higher level of abstraction".

Chikofsky states that the main purpose of reverse engineering is to increase the comprehensibility of software systems. He then refined the objectives of reverse engineering in 6 points:

1. Cope with complexity
2. Generate alternate views
3. Recover lost information
4. Detect side effects (using observation to detect faults)
5. Synthesise higher-level abstractions
6. Facilitate reuse

The tool that we produced in our work is considered to be reverse engineering because it involves generating a view of the software – the dynamic view – in a way that supports answering many questions related to maintenance in general and program comprehension specifically. We focus particularly on coping with complexity when dealing with large legacy systems and with the dynamic information generated by their execution. Among other things, we synthesis higher-level abstractions out of the raw data.

## 2.2 Main research in reverse engineering

Research in reverse engineering belongs to several thrusts that are evolving in parallel. In the following subsections we will describe some of the most important of these thrusts. Note, however, that reverse engineering is not an exact science, and its terminology is not always consistent [Tilley 96].

### *2.2.1 Program Analysis*

Most commercial systems focus on source-code analysis and simple code restructuring using the most common form of reverse engineering: program analysis. Program analysis is performed by a parser that generates a new representation out of the source code [Rugaber 95]. Generated representations are usually stored in a database and vary in the scope of the captured information. Abstract Syntax Trees (AST) are the most comprehensive representations, and permit other sophisticated program analysis operations.

Several analyses can be performed to generate alternative views of software [Rugaber 95]. Control flow analysis can be intraprocedural and thus provides a determination of the order in which statements can be executed within a sub-program and yield a *control flow graph*. A control flow graph is a graph whose nodes represent *basic blocks* and whose edges represent possible control flow between blocks. The basic block is a maximal set of statements or instructions with single entry and single exit (also called branch since it does not contain branching). Interprocedural analysis determines the calling relationships among procedures and yields a *call graph.*

Data flow analysis is concerned with answering questions related to how variable definitions flow to uses in a program. A structure chart is a call graph in which arcs are annotated with the names of the formal parameters and an indication of whether the arc is supplying values to the called procedure or returning them.

### *2.2.2 Slicing*

Slicing – the process of extracting a program slice – is another derivative of data flow analysis.  A program slice consists of a valid program containing only those statements in program P that may affect the value of variable V at some point [Korel 97]. A program slice can either be a static slice that preserves the program behaviour with respect to a variable for all program inputs, or a dynamic slice that preserves the behaviour for a particular program input [Korel 98].

### *2.2.3 Plan Recognition*

Plan recognition is a technique that involves searching the program text for instances of common programming patterns [Tilley 96]. Those patterns that are common and stereotypical are known as clichés. Tools that use this technique provide and accumulate a "plan library" against which they automatically perform the search.

Patterns can be structural or behavioural, depending on whether one is searching for code that has a specified syntactic structure, for code components that share specific data-flow, control-flow, or dynamic (program execution-related) relationships.

Program plans are abstract representations of source code fragments. Comparison methods are used to help recognise instances of programming plans in a subject system. This process involves pattern matching at the programming language semantic level.

Due to the variety of ways a pattern may be represented in the program, plan recognition is a difficult research problem. In fact, Woods and Yang [Woods 96] proved that complete program understanding using plan recognition is NP-hard.

### *2.2.4 Concept Assignment*

The need to locate source code of interest to programmers is widely recognised [Wilde 95]. In particular programmers need to find the correspondence between high-level domain concepts and code fragments.

One approach to this problem is concept assignment. Concept assignment is the task of discovering individual human-oriented concepts and assigning them to their implementation-oriented counterparts in the subject system [Rugaber 95]. This type of conceptual pattern matching enables the maintainer to search the underlying code base for program fragments that implement a concept from the application.

Concept recognition is still at an early research stage, in part because automated understanding capabilities can be quite limited due to difficulties in knowledge acquisition (e.g. the identification of comments) and the complexity of the matching process.

### *2.2.5 Design Recovery*

Design recovery [chikofski90] is a subset of reverse engineering in which domain knowledge and external information, as well as deduction or fuzzy reasoning are used to identify higher levels of abstraction beyond those obtained by examining the system itself. Thus design recovery is characterized by utilising multiple sources of knowledge that may include knowledge from the heads of human experts to reconstruct vital design information. An especially important type of design recovery is architecture recovery.

Documentation has traditionally served an important role in aiding program understanding [Tilley 96]. However, there are significant differences in documentation needs for software systems of large size compared to the small programs. Most software documentation is “in-the-small,” since it typically describes the program at the algorithm and data structure level. For large systems, an understanding of the structural aspects of the system’s architecture is more important than any single algorithmic component.

Architecture recovery is particularly relevant in large legacy systems because such systems tend to have poor structure and out of date documentation [Tilley 96]. Even if documentation of lower-level details is still good, the size of the system still makes it difficult to navigate.

Architecture documentation provides a compact functional overview of the whole system. Typically, architecture is represented using diagrams where boxes represent functional subsystems and arrows connect boxes to show how the subsystems interact.

## 2.3 Dynamic Analysis

Most reverse engineering work is based on examining static aspects of software. Less studied are approaches based on examining dynamic aspects; such approaches are called dynamic analysis. One reason of why dynamic analysis attracts much less attention is because of limitations in the data it provides. Such data may not be trivial to collect, usually are of large size, and are typically hard to comprehend [Jerding 97].

Dynamic analysis involves collecting information about the software during execution. This information should describe aspects of its dynamic behaviour such as control flow, data flow and event sequences. Testing and profiling are the most common domains that use dynamic analysis [Rugaber 95]. In the context of reverse engineering, the goal of dynamic analysis is to understand dynamic characteristics of a design (execution sequences and relative time ordering of events) [Wilde 95].

### *2.3.1 Overview*

In general, dynamic analysis approaches are based on running the software, collecting information about the execution, analysing this information and potentially visualising it. The classical way of generating dynamic information is to instrument the source code i.e. to insert probes (print statements) at interesting points in source code. The system is then run for some scenarios where probes execute, directing their output to a trace file. The trace file is then the input into dynamic analysis tools where its data is analysed and visualised.

Generally speaking, a trace file contains a recording of the execution sequence and relative time ordering of events [Wilde 95] during program execution. Events are visible points in the execution; they can be on several levels of granularity and usually reflect important points in the control or data flow. For example, if inter-process communication is the focus, probes would be inserted before inter-process operations so that the generated traces reflect inter-process activities. The trace entry can contain more information than just the fact that an event occurred – it might contain the value of certain variables.

The proposed thesis research mainly concerns dynamic analysis, although some static information is used to augment the analysis of dynamic data.

### 2.3.2 Basic activities in dynamic analysis

Tilley [1996] identified three basic activities involved in the general reverse engineering process:

- Data gathering – either through static or dynamic analysis
- Knowledge organisation – creating abstractions for efficient storage and retrieval of the raw data
- Information exploration – through navigation, analysis, and presentation.

Similarly we describe the general approach for dynamic analysis to involves 3 phases: Data gathering, data processing and analysis, and data visualisation.

#### 2.3.2.1 Data gathering:

Several design decisions at the data gathering phase determine the nature of the dynamic analysis. One of the most important is the choice of events or the interesting points to capture.

At one hand, the classical choice of event granularity in dynamic analysis related to profiling and testing is the basic block. Such capturing may require instrumenting at every branch of the program (if and case statements). On the other hand, most work on dynamic analysis for program comprehension deals with process-level granularity [Kunz 94, 97].

Another design decision concerns the method of producing the data. As mentioned above, the most common way of gathering dynamic data is to instrument the source code of the target program by inserting probes (print statements) and generating a corresponding trace file. This is an example of a *code-intrusive* approach because it involves modifying the source code. Such modifications will not affect the function of the software (i.e. perturb its

execution) except in a timeshared (e.g. multithreaded) system where race conditions could be present.

Non-invasive approaches are those that do not affect the source code. Data can be generated, for example, by polling the runtime compiler/executor for information on the state of control or variable contents.

#### *2.3.2.2 Data Processing and Analysis*

Data processing and analysis involves acquiring knowledge that is not directly evident from the data and changing the data form to be more suitable for particular tasks. The most common form of data processing deals with compressing the trace or extracting from it higher level abstractions to facilitate its navigation and comprehension.

Trace compression is well studied in the contexts where traces are used for performance monitoring of distributed or parallel systems or for program simulation [Larus 93, Elnozahy 99, Yan 98].

#### *2.3.2.3 Data visualisation*

A rich area of research is the visualisation of dynamic data. One way of dividing visualisation approaches is by their time relation with the generation of data. Run time approaches involve visualising data during the program's execution (as it happens) while post-mortem visualisation approaches look at previously recorded execution traces.

Other dimensions for categorisation involve the unit of data to be visualised (e.g. at what granularity) and the method of interaction between units (e.g. animation versus still display.)

Section 2.7 will describe several tools that use the different approaches discussed above.

## 2.4 Program Comprehension and cognitive models

Researchers in this domain have tried to define and validate cognitive models that describe how a programmer understands code during software maintenance and evolution. A *cognitive model* describes the cognitive processes and knowledge structures used to form a mental representation of the program under study [Rugaber 95]. It is a theory about the various cognitive activities and their interrelationship used as a programmer attempts to understands a program.

Several cognitive models have been described for program comprehension. Most models have been developed based on observational experiments. Various empirical works have been conducted to explain and document the problem-solving behaviour of software engineers engaged in program understanding. Von Mayrhauser and Vans surveyed this area in [Von 95] and compared six cognitive models of program understanding.

In the following section we will briefly go over the earlier models, then concentrate on the integrated model that encompasses the crème of several models. Cognitive models are usually classified as either bottom-up or-top down:

### *2.4.1* ***Bottom up***

Bottom-up comprehension models propose that the program understanding is built from the bottom up, by reading source code and mentally chunking together low level programming details to build up higher level abstractions. These abstractions are also grouped into higher-level concepts until a high-level understanding of the program is reached [Storey 99].

In the Pennington model [Pennington 87], a program model is first constructed by the comprehender. This construction involves chunking the microstructures of program text (statements, control structure and relationship) together into macrostructures that correspond to control flow abstractions that capture the sequences of operations in the

program (i.e. programmers follow the execution flow in code and assimilate it into high level meanings). Once the program model has been fully assimilated, the situation model is developed. The situation model involves data-flow abstractions and functional abstractions. Functional abstractions are the cross-referenced mappings between source code and corresponding domain concepts such as program goals.

### *2.4.2 Top Down*

Top down models suggest that comprehension begins with the formulation of general hypotheses about the program functionality and then the hypotheses are refined into sub-hypotheses to form a hierarchy until sub-hypotheses can reach a level where they can be matched with code. The top-down model is considered to be behavioural because the comprehension process begins from external program behaviour.

Brooks [Brooks 83] proposes a top-down model that describes program understanding as a hypothesis-driven reconstruction of the mapping between domain level concepts (application domain) into the low level programming constructs (source code) using intermediate domains. To achieve this, a global hypothesis describing the program is defined. The comprehender then tries to verify this hypothesis. This in turn can cause further hypotheses to be created, building up a hierarchy of hypotheses to be verified. This continues until a level is reached where the hypothesis is matched with code and thus can be verified or proven to be false

### *2.4.3 Discussion about cognitive models*

The early cognitive models, in our opinion, are of little practical value in providing useful solutions for supporting software maintenance. They are more of theoretical value and are only loosely related to the real problems experienced by engineers in industry. Pennington for example used earlier psychological work on text comprehension while Brooks used work on problem solving in other domains such as physics and thermodynamics. They all used observational experiments to develop their models (except for Brooks). These experiments have been limited by the fact that they were on programs of small size

(between 20 and 500 LOC). As such, there is little evidence the models' results can be generalized to explain the task of programmers in real life situations where typically systems are much larger.

We argue against the validity of experiments alone to theorize about program comprehension during SM because there are too many variables to be considered. Storey [99] notes that application and programming domain knowledge, maintainer expertise and familiarity with the program, all affect the comprehension strategy used by the programmer during program comprehension. Jorgensen [95] empirically confirms the influence of many other variables such as the language used, the age and education of the programmer, and the size of code needed to be modified.

In fact, these variables largely explain the difference between the different models [Storey 99]. Brooks however theorized without using experiments; we along with other researchers [Lakhotia 93-b] heavily use and support his findings.

#### *2.4.4 The Integrated model*

Recently, the work of Von Mayrhauser and Vans [Von 93, 96 98] moved cognitive model research closer to the software engineering domain and solutions. Their subject systems were about 50-80 KLOC as opposed to early work that was in the range 20-500 LOC, and involved industrial engineers as opposed to students.

In their integrated model, Von Mayrhauser and Vans integrated ideas from the major theories in program comprehension into one consistent framework that can explain a significant portion of SM activities. They also explained how each component of this model complements the others. Their suggested cognitive model uses the work of Pennington and Brooks the most and it consists of:

1. Program model (bottom up)

2. Situation model
3. Domain model (top down)
4. Knowledge base

The first three corresponds to different cognitive comprehension processes that are used by the programmer, and the fourth is used by the first 3 models. Each process has its own mental model of the program being understood. During maintenance, any of these 3 processes can be activated from any of the other processes. Maintainers during experiments have been observed to switch frequently between these different models depending on the problem-solving task at hand and their familiarity with the code. The switching is related to beacons (clue recognition), hypotheses, and strategies.

In particular, when the code is familiar, the top-down model is used, and when the code is unfamiliar, the bottom-up strategy is used. For example, when a maintainer is reading a program in the top-down manner, he may come across an unknown section of code; then the maintainer switches to the bottom-up strategy to determine what this new section does. Typically, switching from program to situation models happens because the engineer is trying to link a chunk of program code to a functional description in the situation model. Switching from situation to program model occurs when the engineer may be looking for a specific set of program statements to verify the existence of certain functionality.

The integrated model describes each sub model and the major tasks exercised within its context as follows:
The domain model is a top down model that is invoked when the code is familiar. The domain model incorporates domain knowledge that describes program functionality as a starting point for formulating hypotheses. It guides understanding. The major sub-tasks in this model are:
1. Gain a high level view of the system
2. Determine the next statement to examine
3. Determine the relevance of code to current hypotheses or mental model
4. Generate hypotheses about the program statements under study.

The program model is invoked when the code is new to the SE. This is almost the same as the Pennington model described earlier. The major sub-tasks in this model are:

1. Read comments or other related documents
2. Examine next statement
3. Go to next module
4. Examine data structure
5. Data slicing and tracking changes to variable contents during program execution
6. Learning by chunking information and storing it in long term memory

The situation model (bottom up) is an intermediate model that uses the program model to create a data flow or functional abstraction. This model is constructed by mapping functional knowledge to high level plans or by mapping from the program model bottom up. It is concerned with algorithms and functional data structures. The previously acquired situation knowledge together with hypothesis-generation drive the acquisition of new knowledge and help to organize it as chunks of information for storage.

For example, at the domain level of an OS, we may have a picture of the process control block (PCB). At the situation model, a PCB corresponds to a table, while in the program model we will be concerned with a C struct and how it is used and updated in the code.

The knowledge base is some of the long-term memory and is usually organized as *schemas*. It contains different schema for the 3 processes in 3 separate models. The knowledge base acts as a repository for any newly acquired knowledge. Newly acquired knowledge is associated with the corresponding models.

### *2.4.5 The integrated model and tool capabilities*

Von Mayrhauser and Vans maintain that "the first step in satisfying a maintenance engineer's information needs is to define a model of how programmers understand code"

[Von 93]. To validate and refine this model they produced a large body of empirical results based on several experiments that cover different variations of software maintenance, such as adaptive maintenance, corrective maintenance and reengineering [Von 98].

Most of their experiments were based on the integrated model that they developed; results were arranged according to the model's classification of tasks and models. Their work yielded a large harvest of empirical data that ranked task's and subtask's importance depending on experimental variables such as the type of maintenance and the expertise of the participants.

Unfortunately however, the size and the granularity of their tables that serve to confirm or disconfirm some hypotheses made it hard to convert this data into directives or guidelines for tool development. Additional valuable information can be found in the informal conclusions that they wrote at the end of their many papers. We present next a compilation of their main conclusions:

1. A tool should support the natural process of understanding and not enforce a new one. This can be done mainly by satisfying the information needs and the cognitive processes of programmers.

In particular, there is a need to <u>support the switch between different mental models and the connection between them</u>, since this connection is an integral part of comprehension. They observe that most available tools support one model only. They suggest the use of bookmarks or hyperlinks to establish cross-referencing knowledge between different models.

2. <u>Memory</u>, and in particular short-term memory – an old name for working memory (WM) – is vital during comprehension, so a tool should <u>alleviate its capacity limitation</u>. Incorporating more domain and specialized knowledge will support the WM: "<u>domain knowledge facilitates retention and retrieval of knowledge</u> about the code," as it provides a

"motherboard into which specific product knowledge can be integrated more easily."

3. Bottom up comprehension should be reduced. They write: "Chunking of small pieces leads to many layers of program model until the situation model level can be reached. This is cognitively taxing." They argue that incorporating more domain and specialized knowledge will reduce bottom up comprehension, saying: "lack of specialized knowledge leads to slower code cognition because understanding has to proceed bottom up".

Von Mayrhauser and Vans summarise what a tool is required to do to speed up the process of comprehension: "Tools must quickly and succinctly answer programmer questions, extract information the programmer asks for without extraneous clutter, and represent the information at the level at which the programmer currently thinks."

### *2.4.6 Discussion*

The approach of von Mayrhauser and Vans is based on defining the information needs of programmers during SM. The definition process begins by defining a comprehension model with a level of detail sufficient to identify and observe these details.

The authors, after generating a template out of their model of what the activities of SM are, begin to experiment and observe in order to allocate frequency and spent-time numbers for each of the activities in their template. Their results fill many tables, each containing between five and twenty rows with multiple columns full of quantitative data.

We argue that such work can be valuable per se but gives little help to someone trying to develop a tool. A large conceptual gap exists between these tables and tool design concepts. The authors were not developing a tool but rather collecting data about what programmers do and need. However expressing what someone is doing or needing can be greatly dependent on what the inquirer needs to know or do. The choice of the abstraction level (what activities to observe) and the language in which the observation is expressed can be critical. The authors' shortcomings are in their assumption that the mere collection

of data would be enough for a solution to be developed without considering the challenges of tool developers.

We think that better help can be provided by generating comprehensive theories about how to alleviate the difficulties of SM in a way that is tractable by a tool. In other words, what is needed is information about how certain big user problems can be solved with a solution that can be translated into a working software tool instead of detailing quantitatively what the maintainer actions are and how much they spend time on each action.

### *2.4.7 Partial comprehension*

Erdos and Sneed [Erdos 98] propose a simplified model that covers the information requirements and tool capabilities necessary to support SM activities. They base their proposal on their vast experience in SM (40 years).

The authors begin by arguing that researchers in program comprehension have paid too little attention to the real requirements of maintenance programmers. In their opinion, complete program comprehension is not always necessary. They note that tools that assist with complete program representation are not useful or usable for two reasons. First, most of the tools focus on one dimension of a program such as control flow only or data flow only, while a program is an interwoven multidimensional artefact. Second, in their opinion, due to the multidimensional nature of a program, a full representation of more than 10 KLOC becomes impossible.

Instead, they argue in favour of a question-answering model as the basis for tool design to support SM activities. Program representation should be localized around a certain type of concern, i.e. enough to answer individual questions raised by the SE. They argue that an SE needs, “local views of subsets of procedures, data structures and interfaces affected by a particular maintenance request i.e. local comprehension “. They proceed to note, “local representation should only be produced on demand and in response to a particular

question."

They suggest that most SM needs can be met by answering seven basic questions. These questions are:

1. How does control flow reach a particular location?
2. Where is a particular subroutine or procedure invoked?
3. What are the arguments, results of a function?
4. Where is a particular variable set, used or queried?
5. Where is a particular variable declared?
6. What are the input and output of a particular module?
7. Where are data objects accessed?

The authors then describe a static program analysis tool that they claim helps in answering these 7 questions. However, they admit that maintenance is a complex task that has an unlimited need for information about the problem domain and on the solution space so their solution is also a partial solution. However, by answering the most common questions they hope their tool will increase the productivity of maintenance work by a significant percentage.

They also concluded that finding a starting point in the code remains a necessary prerequisite before asking or answering any of their questions. The authors approach in defining a model of the difficulties of SM then developing a tool that directly addresses these difficulties is a step in the right direction. However, they did not provide a methodology or a framework for the path beginning at identifying difficulties to producing a solution; they presumed that it is enough to trust their experience and believe their assumptions. More importantly, they limited themselves to static analysis techniques that, as we discussed in the first chapter, only covers part of the problems of SM.

### *2.4.8 Tools and program understanding*

Storey et al. [97] conducted an experiment on three different program understanding tools. The experiment involved 30 students who were given a set of maintenance problems to solve on a subject system of 17,000 LOC. Participants were asked to think aloud and they were videotaped. The videotapes were later analysed by the authors.

The aim of the experiment was to obtain an insight into, "how do program understanding tools affect how programmers understand programs." The authors produced a set of observations and conclusions out of their experiment.

They observe, "for larger software systems, the true strength of a program understanding tool lies in its abilities to manage the inherently large amounts of information. Although our test program was relatively small, there were several issues related to managing complexity, minimising disorientation, and reducing cognitive overhead."

Their observations showed that participants spent considerable time looking for the code that needed to be modified. As a result, they conjecture that there are two aspects that need to be addressed in a program-understanding tool. First, there should be integrated capabilities to search the code. Second, there should be support for a combination of comprehension strategies and for switching between them as well as, in general, to reduce the cognitive overhead in program explanation.

Also in [Storey 99], Storey and her colleagues developed a hierarchy of cognitive issues to guide the development of tools to help in software exploration and development. The hierarchy has two main branches, one for cognitive processes and strategies during software comprehension and the other for cognitive overhead related to how could the user interface reduce disorientation during browsing and navigating the visualisation of software structure.

Storey et al. examined various cognitive models, and highlighted many issues that should be a concern during tool design. They examined existing tool capabilities, and how they

address these concerns. They also applied software exploration techniques that reduce disorientation during exploration of hyper documents.

The hierarchy, however, does not provide sufficient insight as to how the identified issues can be resolved, nor does it tell when certain information should be made available. Nevertheless, the authors draw useful conclusions regarding supporting cognitive processes; they write, "more support for mapping from domain knowledge to code and switching between mental models would be useful." Regarding cognitive overhead in navigation they write, "better navigation methods which encompass meaningful orientation cues and effective style for browsing large software systems are also needed".

## 2.5 Methodologies to identify the problems in SM

A major theme in this thesis concerns understanding SEs problems and tasks in a way that ensure good tool design. In this section, we investigate various approaches cited in the literature that tackle this issue.

### *2.5.1 Empirical and ecological study of SM tasks*

Most empirical work on SM tasks is considered to be of the *theory validation* type [Jorgensen 95]. These classical empirical studies are based on setting a hypothesis and then gathering a large amount of information from many contexts to confirm or disconfirm the hypothesis. We believe that empirical work should have a more active role, generating ideas and hypotheses instead of passively validating pre-set ones.

Fortunately, a new paradigm that satisfies these requirements is evolving. Such a paradigm is sometimes called the *ecological* study of programmers, a term introduced by Shneiderman and Carroll [88]. Shneiderman and Carroll define ecological study as, "the thrust in software psychology where usability researchers are direct participants in the definition and creation of new software artefacts."

The ecological study of programmers represents a new paradigm in empirical studies where the emphasis is:

1) Realistic software situations should be confronted on their own terms

2) The work is directed toward design results, not merely toward evaluation and description in the service of design goals. The trajectory of this paradigm is the development of ideas that directly impact productivity and quality in software.

Shneiderman and Carroll argue that such a realistic approach is more suitable to generate good results compared to research based on experiments, "because software design takes place in software shops not in psychological laboratories." They describe ecological studies as allowing a richer style of task analysis that, "can not stem from simply hypothesis testing procedures." It has to be, "imperative, inductive, it seeks to discover, not merely to confirm and disconfirm".

The main research setting for ecological studies is the case study. It is based on collecting lots of information from perhaps a small number of individuals. They argue that studying unique situations at a fine grained level will not be useful for statistical analysis but it will help understand the most important events in the use of software: the meaningful interactions and how they break down. We fully agree with and adopt the ideas of this paradigm that is clearly influential on our approach.

Soloway et al. [88] used an ecological design approach to studying software professionals. They dealt with the design of software documentation for maintenance. Their goal was to investigate the types of representation that might help programmers when they attempt to make changes to a program.

They used a research methodology based on: a) collecting data on realistic and real software tasks, b) characterizing the behaviour they observed in cognitive terms, and then c) developing some artefacts based on the theoretical analysis in order to enhance the programmers' productivity.

A set of software professionals was chosen and was asked to think aloud as they carried out tasks in their realistic setting. Audio records were later analysed. Subjects were given a 200-line program and were asked to solve several maintenance questions.

The observation and analysis of the authors focused on one major source of difficulties, the delocalised plan. They define such a plan as, "pieces of code that are conceptually related but are physically located in non-contiguous parts of a program." They note that a programmer who fails to develop a correct modification is often the one who fails to understand the causal interactions inherent in one of the delocalised plans.

The authors' work represents an improvement over methods based on laboratory experiments to understand the sources of difficulties in SM. They went to an industrial environment and experimented with professional developers. However, their subject system was not fully realistic and neither was the approach of directing their work by providing tasks instead of watching the subjects' own tasks. Yet, we agree with their self-evaluation when they define the value of their work as: "identifying specific cases in which programmers have particular difficulties." In our opinion, such a not-too-ambitious approach would eventually yield more reliable and valuable results than trying to solve everything at one time.

## 2.6 TkSee and the work practice approach

In this section we present the background of the research in the KBRE group to which the author belongs. The work described in this thesis can be considered as a continuation of the KBRE work described in [Lethbridge 98]. This thesis will augment their work and use the data they collected from studying work practices. All the programming work of this thesis will eventually constitute an extension to the TkSee tool developed by the KBRE and which is serving as the infrastructure for various studies of program comprehension.

### *2.6.1 TkSee*

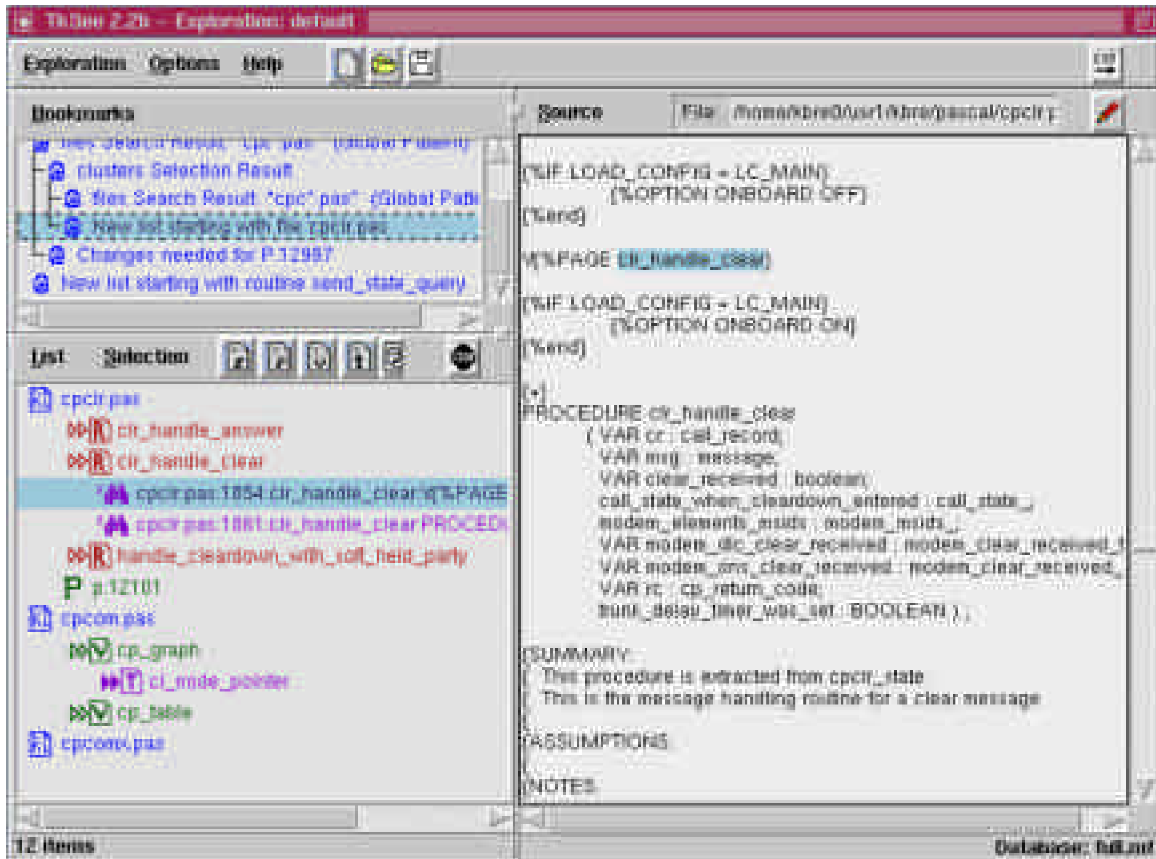


**Figure 1: Screen dump of TkSee: the exploration pane (left) shows symbols. The source code for the selected routine is displayed in at the right pane. The top-left history pane shows the current task.**

TkSee is a software exploration tool that allows software engineers to explore and understand source code. TkSee provides two major types of functionality: software object based exploration and string based search facilities. The former facility allows the users to pose queries about "software objects". The software objects can be such things as files, routines, variables, types, and lines in a file.

Query results form a hierarchy where indentation in the hierarchy represents any kind of relationship between the parent software object and the child (indented) software objects.

Indentation in the hierarchy represents relationships such as: 'routine calls routine', 'routine called by routine', 'variable defined in routine', 'type used in routine', 'line found in file', etc.

Users can perform many operations on the hierarchy such as deleting branches or expanding others by requesting new displays of related objects. TkSee is designed to satisfy many of the information needs of a SE, particularly those related to static relations between software objects.

The string based search facility is a sophisticated way of performing search that would normally be done using grep. An advantage of TkSee over grep is that a search result can be saved and refined.

### *2.6.2 Work practices*

The importance of TkSee to our research is not only that it will be the infrastructure on which we will build DynaSee but also because we benefit from the research methodology used to develop TkSee.

TkSee has been designed with the goal in mind of having a tool that is highly likely to be adopted. As such, the design requirements were generated using a new approach for the identification of tool requirements. This approach is based on studying the work practices of SEs. Singer and Lethbridge [Singer 97] argue, “By focusing on workplace activities, the study of work practices increases the likelihood that tools can be smoothly integrated into the users' daily activities. This, in turn, should increase the acceptance and use of software tools designed on the basis of work practices.”

In studies of work practices, data are generally collected by following and recording the work that people do. Singer and Lethbridge [Singer 98] used various techniques to collect SE work practice data such as using web questionnaires that asked the SEs what they do, observing an individual SE doing his work for several weeks, interviewing many SEs and

obtaining company-wide tool usage statistics.

These studies enabled to determine the time spent on different activities. Activities with high time allocation were targeted in the requirements. For example, search was found as an important component of real, day-to-day, software engineering. They observed that SEs repeatedly search for items of interest in the source code, and *navigate* the relationships among items they have found. This observation was addressed in requirements [Lethbridge 98] to provide advanced search capabilities that in the same time allow software engineers to keep track of search results across different search sessions.

The TkSee tool that was designed to satisfy the requirements did not cover all the activities with high time allocation. Lethbridge and Singer [Lethbridge 98] observed that significant time is spent on dynamic analysis and in investigating the routine call hierarchy. Our work in this thesis targets this part of dynamic analysis.

Singer and Lethbridge [Singer 97] present the work practice study approach as an alternative to the cognitive models approach as a means to identify design requirements for tools. They write, the “work practice study approach can reduce, or perhaps even eliminate, the need to study cognitive processes and mental models”. In section 0 we discussed the cognitive models approach and its shortcomings that included the presence of a gap between the cognitive findings and tool requirements. Lethbridge et al. emphasise this point and write [Singer 97], “it is not at all obvious how to design a tool given a specification of the programmer’s mental model”. Their contribution is not only in implementing a new way to derive tool requirements but also in actually implementing the derived requirements in a tool (TkSee) that is operational and is under evaluation.

## 2.7 Examples of approaches and tools

In this section we describe actual implementations, rather than the theories, of various tools and systems that can be considered related to the tool that we describe in this work.

### *2.7.1 Debuggers and dynamic analysis tools*

Classical debuggers such gdb, dbx or those integrated within the development environment of most modern languages can be considered as dynamic tools that help in program comprehension (to understand the execution of a program)[Korel 98]. They can be used to bridge the gap between source code and the sequence of execution by projecting the order of execution onto the static order. Classical debuggers can be either interactive, e.g. highlighting the statements that are executing, or post-mortem, e.g. generating traces of execution.

Interactive debuggers are useful to get an insight into the dynamics of a program but suffer serious limitations. They can be useful for limited exploration when the problem is finely localized within a small part of the code. They are, however, considered as an inefficient and time-consuming approach to understanding program execution [Korel 98]. Attempting to follow the execution of a program in a debugger for long periods causes disorientation. Interactive debuggers are perfect examples illustrating the "lost in hyper space" [Marchionini 88] syndrome, since they involve continuous bouncing to different locations in the code. On the other hand, traces generated by debuggers are hard to comprehend because they are low-level, large in size and flat. The problem of traces will be discussed later as one of the problems in dynamic analysis.

### *2.7.2 IsVis*

Jerding and Rugaber [Jerding 97] describe a tool, IsVis, which combines static and dynamic analysis in order to aid in the process of program comprehension. Jerding observes that, "program executions are made up of recurring interaction scenarios and that these interaction patterns occur at various levels of abstraction." His general thesis statement was: "Visualising interaction patterns in program executions can facilitate program behavioural understanding during design recovery, design/implementation validation, and reengineering tasks". Part of Jerding's work was to the creation of the IsVis tool.

Under IsVis, the target system is instrumented to record interaction events such as object instantiation and function calls and returns. The system is run for several scenarios and the generated trace is loaded into IsVis. Using the trace data, the analyst begins by interactively creating "interaction scenarios" that visually describe the relation between different software components in term of events between software components. Components are also defined by the human analyst out of the software artefacts involved in the trace. The interaction scenarios can be visualised using an event trace diagram based on temporal message flow diagrams (TMFD).

IsVis also provides several aids to help analysts locate useful recurring interaction scenarios by using several pattern matching techniques. These techniques include a visual "Information Mural", regular expression matching to find similar patterns corresponding to an identified one, and automatic detection of repeated sequences of interaction.

The Information Mural exemplifies a breed of dynamic data visualisation based on visual pattern identification. The information mural is a two dimensional graphical representation used to visualise large amounts of information. On one dimension, the components are shown (e.g. classes); messages between components are shown on the other dimension (axis).

The real innovation in the Information Mural is in the visual techniques that allow creating a global overview of message traces containing hundred of thousand of messages. The technique utilises grey scale and colour shading along with antialias techniques to create a miniature representation of an entire large information space. Using these techniques, areas that are brighter in the Mural are denser with information, conveying the same visual patterns that would be apparent if a huge trace diagram of the entire program was observed from a distance.

In IsVis, the information mural can be used to navigate through the interactions in time order, or can be used to spot patterns. Groups of interactions can be selected, highlighted and coloured. Then the graphics display can be updated to show how the interactions look

with the selected groupings, ideally simplifying the interactions, producing more patterns and greater understanding of the executing program.

In [Jerding 97-b], Jerding et al. describe the details of the trace compressing technique used in IsVis. The compression is based on a call trace representation that they developed as a middle ground between call graphs and dynamic call traces. They note that a call graph is a very compact representation of the call trace because it only shows a summary of the actual call sequence. It thus hides individual sequences of calls and their temporal order. On the other hand, a dynamic call trace is an unbounded data structure that creates many problems due to its large size, such as finding relevant information. The distinction is similar to the distinction between sequence and collaboration diagrams in UML.

Their developed data structure is a directed acyclic graph (DAG) and its construction is based on removing redundancy caused by loops, recursion and reuse of repeated subtrees. Hashing is used to guarantee that each identical tree structure is only represented once. In such a case, only a new edge is added from the new parent to the root of the shared structure. This representation allows preserving information about the unique call sequence and facilitating detection of repeated patterns. However, this representation does not preserve the temporal order of call sequences that is instrumental for many program comprehension activities.

In general, IsVis is more oriented toward a "global understanding" of the software. It provides an architecture level description of the behaviour, focusing on components and connections. As such, it provides an object-oriented description that is created by humans and only aided by the tool. The literature about IsVis, like most of the RE tools, does not even discuss why the tool is supposed to be useful. The author presents the tool as an achievement per se without any discussion about where it is needed and what is its role in the actual practice of SM.

### *2.7.3 Jcheck*

JCheck from NUMEGA [Jcheck] is a visual debugger used to analyse and debug multi-threaded programs written in Java. It builds a real-time graphical model of a program as it executes, and provides information on Java threads, program events and synchronization objects. Different threads and their interactions are animated and colour coded to depict their state (waiting, sleeping, etc.)

The graphical model allows one to visually analyse program behaviour and diagnose difficult runtime problems, such as thread deadlock, live lock, starvation, thrashing, and synchronization problems.

JCheck provide an excellent insight into the dynamics of a program but it emphasises the performance issues rather than program comprehension.

### *2.7.4 Jinsight*

Jinsight [Jinsight] is a Java visualisation tool created by IBM that shows the execution behaviour of Java programs. Jinsight gets its information from traces of previously executed programs. The Java program is compiled as normal, and then a modified Java virtual machine (VM), which is supplied with Jinsight, is used to run the code. As the program is running, the modified Java VM creates trace information and stores it in a trace file. This file can then be loaded into Jinsight, where different views are available to analyse the trace information.

Jinsight has the ability to show object populations, messages, garbage collection, CPU and memory bottlenecks, thread interactions, and deadlocks. Jinsight has several different displays available from which to view program execution. The Histogram view displays calling and reference relationships among objects. The Execution, Invocation Browser, and Execution Pattern views display sequences of messages among objects as a function of time. There is also a Reference Pattern view, which displays patterns of references among objects. This view can also be used to help find memory leaks. In this view, it can

be seen which objects are holding references that are causing problems for the garbage collector.

To help alleviate the problem of information overload with very large traces, Jinsight provides a pattern extraction facility. This allows recurring patterns to be displayed in a single view. This can help remove large amounts of redundant information, simplifying the display of interactions.

Many of Jinsight's views involve sequences of messages. A primitive abstraction mechanism is provided to clear up the cluttered and sometimes confusing views. No conversions between graphics and code are available, but several presentation modes for visually displaying how the program executed are provided.

Jinsight like Jcheck provides several ways to examine the dynamics of a program but its emphasis is on performance issues rather than program comprehension.

#### *2.7.5 RunView*

RunView [McCrickard 96] is a program visualisation tool designed for understanding and debugging large programs. It provides the user with an overview of programs by replacing program elements with graphical objects.

The code-display window shows each file in the program as a column and each function in a file as a block within the column. The size of the block corresponds to the size of the function. By highlighting files or functions of interest with different colours, the user can run the program and visually trace its execution. The execution-display window shows the current call stack at each function call with the right portion of the window. Each horizontal line in the display represents the call stack at some point in time, with each function in the call stack appearing as a line segment.

To semantically connect the windows, each line segment in the execution display has the same colour as the corresponding function in the code display. At any time, the user can highlight a function representation in either view to determine when the function was called, which functions called it, and which functions it called.

The RunView Dynamic run time view of module activity proves particularly helpful in interactive applications. At each interaction, RunView shows which portions of the code were executed.

### *2.7.6 View fusion*

Pal [98] used dynamic information to augment a static architecture diagram. This approach integrates high level architectural knowledge with call relationships captured during critical moments of code execution. His technique can be discussed in terms of view fusion, with the goal of creating a diagram that illustrates the dynamic interactions of high-level software components. Dynamic interactions between components are represented using numbered and directed edges between high level components.

Pal uses a non-intrusive dynamic data collection method. Using a debugger, low-level interactions are gathered by examining debugger call stack information produced by placing breakpoints on all procedures and functions in the software.

### *2.7.7 Run Time Landscape*

The Run Time software Landscape (RTL) [Teteishi 94], like the approach by Pal, belongs to the group of tools that do program visualisation at the architectural level. Dynamic information is visualised using view fusion with a static layout of software components.

The Run Time Landscape is intended to, "manage and present run-time information about large software systems in a manner that is comprehendible to a programmer". It is an extension of an architecture view of software called the Software Landscape. The Run Time Landscape uses a hierarchical containment view of software hierarchies provided by

the Software Landscape to represent the flow of execution as a moving dot called the execution marble.

Animation techniques are used to make the marble easier to follow as it moves between viewed components as each component receives the control locus. At any point of execution, additional windows can be opened that display various program-state related information such as the program stack or values of different variables.

### *2.7.8 Software Reconnaissance*

Wilde [92, 95] describes a technique that uses program traces to localize the source code that corresponds to a piece of functionality. Software Reconnaissance is based on the comparison of traces of different test cases. The target program is first instrumented so that a trace is produced of the components executed in each test. Then test cases are run, some 'with' and others 'without' the desired functionality.

For example, a programmer who wants to find where *call forwarding* is implemented might run one or two tests that involve forwarding a call, and then one or two tests that are similar, but do not forward a call. The traces are then examined to look for software components that were executed in the first group of tests and not in the second.

Software Reconnaissance (SR) instruments each conditional statement in the target system so that the trace records each executed branch (basic block). When a branch is found in the trace generated by some test cases that exhibit a feature *f* and not in the trace of test cases that do not exhibit *f*, it is designated as the *marker* branch for *f*. While marker branches were not always directly related to the feature in question, Wilde reports they were mostly considered as good starting points to find the relevant code.

## 2.8 Support for program execution comprehension

Korel [98] notes that the typical tool to understand program execution and behaviour is the conventional debuggers that support breakpoints and stepwise execution. He, however,

notes that debuggers are inefficient and time consuming ways to understand large program behaviour.

He presents a tool that describes program execution at a higher level of abstraction than statements. His tool represents a survey for methods that support comprehension of program execution, a subset of dynamic analysis and reverse engineering. The tool includes the following techniques for program execution presentations:

1. Execution on the call graph level where executed modules (e.g. routines) are highlighted on a call graph or a sub call graph that displays only the executed modules.

2. Execution on the call tree level where each executed module is displayed as a rectangle and a line connecting the calling and called module. Korel notes, however, that this technique only works for short execution sequences and that it is “impossible to visually represent the program execution; especially, for large programs and long program execution”. On the other hand, he suggests that some techniques can be used to alleviate this problem such as removing repetitions caused by loops, zooming in/out and collapsing the parts of execution that are not of interest to the programmer. He did not mention, however, any support in his tool for these techniques.

3. Execution on module trace level, where the executed modules are shown as corresponding bars in a viewing window (looking like a linear bar chart). The height of each bar represents the length of the execution race during each module execution (measured by the number of statements executed). The viewing window is of fixed length with all bars having the same width. So the more modules are included, the thinner the bars become in order to all fit inside the window.

Korel’s tool allows for synchronized viewing among all different views. A highlighted module in call tree view, for example, will cause its corresponding module to be highlighted in the graph window and in the linear module-viewing window. Also, it would

be possible to open up a module in these windows (represented by a rectangle) to see its actual source code with the executed statements highlighted.

### 2.8.1 Inter-process dynamic analysis

Dynamic analysis for inter-process interactions is a well-studied area. However, most of the work on dynamic data focuses on evaluating performance [Wilde 98]. In recent years some research has emerged to utilise inter-process interaction traces in program comprehension. We describe some of this research below:

#### 2.8.1.1 POET

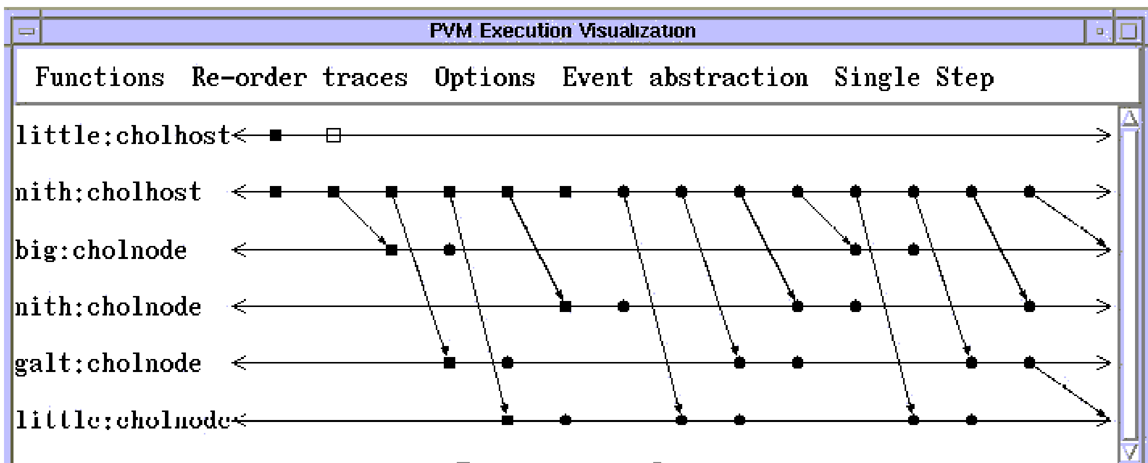


**Figure 2: POET screen dump**

POET [Kunz 97] is described as a tool for collecting and visualising event traces from the execution of distributed or parallel applications. It displays the events as an online form of a process-time diagram. In the display, each horizontal line, called line trace, corresponds to one process. Time flows from left to right, and a scroll bar allows scrolling in the vertical (process) dimension. An event is shown as a circle or square on a trace line, different kinds of squares and circles correspond to different events such as process creation, termination and message sending or reception. Interaction between processes can be shown as pairs of events connected by an arrow. Different kinds of arrows correspond to different kinds of interactions.

#### *2.8.1.2 Event abstraction*

Kunz [94] describes a tool that automatically abstracts inter-process events (e.g. send, receive) that are recorded in a trace file. Trace events are considered *primitive events* and the tool derives out of them *abstract events*. An abstract event groups together several primitive events, thus it “abstracts” them. A hierarchy of events can be constructed where the lower level abstract events are abstracted into higher-level abstract events.

The abstraction process begins by matching traced events with the corresponding source code that produced them. Once matched, several rules that use information about the control flow and data flow of the code are applied to group events into higher level ones. For example, if several events are produced by a set of statements belonging to one basic block, then these events are considered as one abstract event because these events will occur always together thus they are likely to correspond to one higher-level abstract event. Similarly, if several abstract events are derived from statements of one routine then they can be grouped in one higher-level event because a routine is likely to achieve a higher-level functionality.

While Kunz claims that it is possible to assign meaningful interpretations for derived abstractions, he notes that significant time is required to understand the derived abstractions. He also suggests allowing the user to modify such abstractions because they don't always match the user's high-level definition of abstract events.

#### *2.8.1.3 Pattern detection in traces*

Kunz and Seuren [97] describe an application for compressing traces of inter process communication by detecting patterns of communication in the traces. Patterns are first defined by the user using a graphical interface and stored in a patterns library. Then a search is performed on the trace to locate any occurrences of any patterns in the library. The user can also automatically define a pattern by inserting special probes in the source code. On execution, these probes produce annotations in the trace that group a set of trace entries and constitute a user-defined pattern.

## 2.9 Dynamic slicing

An interesting approach that can be related to our work is dynamic slicing. A dynamic slice is a valid program containing only those statements in program P that may affect the value of variable V at some point [Korel 98] for a given input. A dynamic slice is computed from the backward analysis of a program trace.

### *2.9.1 Dynamic slicing and program comprehension*

Originally program slicing has been proposed to guide programmers during debugging but in many cases it may also be used for program comprehension during SM by reducing the amount of details a SE (or programmer) see.

Korel [98] pushes the idea of dynamic slicing as a mechanism for program comprehension by assuming that, "each program function can be represented by a variable or a set of variables at a certain program point" and therefore by slicing for the representing variables we can get a slice for a function. Seeing only the details related to a function of interest to a SE is very desirable especially in large systems.

Dynamic slicing is more appropriate at this task because it produces a much smaller slices than static slices, only those statements that contribute to the computation of a selected function for a given program execution (input) are included in the slice.

Dynamic slicing was used to improve comprehension of program execution by limiting the execution presentation to the contributing part only. It is important to understand that contributing statements for a function are not equivalent to executed statements; not all executed statements contribute to the computation of a function of interest that is represented by a variable. Using dynamic slicing algorithms, it is possible to identify statements that affect the variable and step only through these statements or through the modules that contain them.

Alternatively, dynamic slicing can be applied on all program execution presentations (see Section 2.8). For example, contributing modules can be highlighted in a module level presentation of a program execution; alternatively, non-contributing modules can be simply removed from the presentation. Moreover, the degree of contribution can be computed from the number of contributing statements and this degree will be reflected in the visual appearance of the module representation.

### *2.9.2 Discussion*

The use of dynamic slicing for debugging is very likely to be of great use. It is more controversial to use it to support program comprehension for large programs. A dynamic slicing algorithm that requires statement level tracing is likely to suffer from scaling problems.

Also, there are questions about the validity of the assumption about whether all or at least most program functions that are of interest to a software maintainer can be represented by a practical number of variables. More importantly, the identification of such variables, if they exist, is not a trivial task and Korel [98,97] did not mention anything about supporting this activity.

According to our task views presented in the next chapter, typically a software maintainer needs to have a good comprehension of the program *before* he can identify the variables that represent his function of interest, making this activity a vicious cycle. However, we think that dynamic slicing, as a way to augment techniques for comprehending program execution, is of definite value.

# Chapter 3 Justifications for our approach and case study

In the first part of this chapter we will discuss the theoretical justifications for this thesis. Based on these justifications, we propose the general idea of our cognitively based approach for RE tool design. In the second part of the chapter, we describe a case study where we applied our approach on a realistic industrial setting.

## 3.1 Theoretical Justifications

The first theoretical justification for our approach is that the low adoption rate of reverse engineering (RE) tools is largely due to a gap in perception of software maintenance (SM) problems between tool designers and the software engineers (SE) who play the role of end users. This generates the following research question: given that there is no universal characterization of SM, how can one identify the real problems of SM? That in turn can be divided into two questions: Q1: How to characterize SM tasks, and Q2: What criteria can be used to rank the difficulties of SM tasks.

The second justification is that a RE tool can increase efficiency by reducing cognitive load (CL) spent on performing SM tasks. This leads to the need to define a theoretical framework based on the nature of CL and how it can be reduced.

### *3.1.1 Tool adoption and the real problems of SM*

The problem of tool adoption is not a new one. It relates to an older problem that has been addressed in MIS research since the mid-1970's, regarding the factors that influence an individual's use of information technology [Compeau 95].

The major theoretical thread in this domain is derived from social psychology theories and relates to behaviour and attitude toward technology. It conjectures that adoption results from the individual expectations of positive outcomes. This thread matured with the well-known and validated *technology acceptance model* [Davis 89] that states that adoption of

new technology takes place when there is *perception* of usefulness and ease of use in the candidate adopter. We rely on this model to assume that, to be adoptable, a tool that supports SM has to convey a perception of increased efficiency and reduced complexity of the tasks.

More specifically to our concern, it is largely accepted that a major source of low RE tool efficiency is due to a gap in perception of the real difficulties in SM between tool designers and their users (the SEs) [Storey 99, Lakhotia 94]. When tool designers have a different perception of a user's tasks and problems, they will design a tool to solve irrelevant problems. The success of a tool's adoption is dependent on how much the tool solves real and hard problems, and how much the sources of difficulties are eliminated or alleviated so that the reward (positive outcome) in using the tool can be perceived by the end users.

A detailed SM characterization is needed in order to be able to identify and understand the main problems and difficulties of SM in a way that allows for the design of an efficient tool. This understanding should clarify the details of SM at various levels, particularly the cognitive level, as program comprehension is mainly a cognitive process. Even after a characterization has been performed, a proper indicator of what makes a certain task "difficult" should be identified and used.

#### ***3.1.2 How to characterize SM tasks?***

Several approaches have been used to tackle the problem of characterizing SM (see Chapter 2 on some of the details of different approaches). Empirical methods have been used frequently; however, most empirical work on SM tasks is considered to be of the *theory validation* type [Jorgensen 95, Shneiderman 88]. These classical empirical studies are based on setting a hypothesis and then gathering a large amount of information from many contexts to confirm or disconfirm the hypothesis. We believe that empirical work should have a more active role to generate ideas and hypotheses instead of passively validating pre-set ones.

Our approach for the characterization of SM can fit under the ecological study of programmers paradigm. As discussed in Chapter 2, in this paradigm proactive methods of investigation are used seeking to discover big problems rather confirming or disconfirming hypotheses. The methods are applied on a limited context or a case study where the targeted context is thoroughly analyzed and the results are oriented toward designing a solution in terms of software artefacts.

The decision to work on a limited context that is also narrowed down to a difficulty model, as opposed to supporting all the activities performed during software maintenance, is not only based on our preference of approaches but also has additional rationale related to the nature of SM. Software maintenance is a sophisticated creative process that is far from being fully understood [Ducasse 99]. In fact, SM [Pfleeger 98] is defined as any work done to change the system after it is in operation. As such, SM defines a large space of tasks and problems. The nature of SM differs dramatically between different contexts, so SM means different things to different people in different contexts.

In our opinion, neither a universal characterization nor a universal solution is possible or, at least, it is not possible to get everybody to agree on a universal description. Focusing on a deep but narrow problem has more chance of quick success, and can contribute to knowledge that will eventually help people to build up better general SM understanding and tool support.

This approach also allows us to focus better on a manageable space and thus better determine the usefulness of any of the SM support techniques. In the context of a full program comprehension tool, it would be hard to isolate the strengths and weaknesses of individual approaches.

In other words, instead of designing panacea tools for all the SM, we use the craftsman's tool approach. A tool becomes useful and needed when it helps in some settings (task view) that are difficult enough to justify its use and eliminate particular difficulties in these

settings (problem model). Although a tool may be developed within the context of a particular problem, most tools ends up being used in many other contexts. The magical tool that solves the entire set of craftsman's problems in all settings is unrealistic, and so is the tool for all the SM problems.

### *3.1.3 What criteria can be used to rank the difficulties of SM tasks?*

As we said before, after a clear model for SM is defined, the next phase is to analyze this model in order to identify the sources of difficulties that make SM such an inefficient process. A prerequisite for this identification is the adoption of a criterion to rank activities and difficulties.

Classically, the amount of time spent on an activity or the frequency of performing the activity was used as the indicator to evaluate its efficiency. While time can detect many of the inefficiencies, it hides many underlying factors affecting cost and productivity. Reliance on time alone means that all units of time spent have an equal cost in terms of productivity, regardless of the cognitive effort required (e.g. an hour reading a newspaper would be equal to an hour of reading a math proof!).

### *3.1.4 Cognitive load (CL) as a criterion*

We argue that a better indicator of efficiency is the amount of cognitive load (mental effort) spent; the more cognitive load (CL) is needed, the more inefficient is an activity.

We hypothesize that CL is a better indicator (a more accurate gauge for efficiency in reducing the SM cost) than time because we believe it accounts for more aspects affecting the successful adoption of tools (and thus productivity) than time alone. One of these aspects is the likelihood of succeeding in a task: it seems reasonable to state that as tasks increase in CL, the chances of errors will increase, and the performer's confidence in the correctness will decrease. The reader can verify this by introspecting about the possibility

of making an error on a simple task such 1000 * 30 versus a more complex task such as 6973 * 31. Some necessary tasks during program comprehension of very high CL may be ignored or approximated, thus negatively affecting the efficiency of SM. A time-only based metric will not capture such implicit inefficiency.

Of course, CL also affects directly the time required to perform a task. Note however that we don't claim that all sources of inefficiency are detected by CL, rather CL increases the accuracy of such detection. It can be complemented by other measures such as time and frequency of use.

### *3.1.5 The notion of CL*

The notion of CL is well grounded in cognitive psychology. The human cognitive system has limited resources and using its resources near their capacity causes a perception of difficulty. But as the load increases, errors begin to occur and finally a complete failure in performing the task occurs. In decision-making tasks, it has been proved that as information increases beyond certain limits, the performance decreases parabolically [Umanath 94].

The most important and scarce resource of the cognitive system is in working memory (WM). WM is considered to be the bottleneck of the cognitive system [Card 83 p. 392]. WM has a well-characterized limited storage and processing capacity. Storage and processing resources are hard to differentiate, since processing requires space to store intermediate results. For simplicity, we will combine both kinds of loads under the term CL (see the discussion about the definition of CL in terminology section of Chapter 2).

CL, even when there are no overloads, will have negative effect on performance. The support for this assumption comes from the work of Steinberg whose experiments demonstrated that the retrieval time from the WM is linearly proportional to the number of items stored in the WM [Ashcraft 98 p.116]. In other words, the more WM is occupied, the longer the retrieval time is.

#### *3.1.5.1 Measuring CL*

Retrieval time in laboratory experiments signifies the mental effort and thus CL under our definition. In fact, the only way to quantitatively measure CL in experimental settings is by measuring the *mental time* to achieve a task. However, for this to be valid, the task should be small and well determined to prevent multitasking and to ensure that the elapsed time was purely spent on mentally processing the task. Examples of such experiments are measuring the time a subject takes to find an answer for a mental task such as rotating objects (note the mental rotation experiment of Shepard described in the next section) or by measuring eye fixation during reading to determine which words are harder to comprehend.

Outside the laboratory setting, such as in our case, the time spent on a task has to be carefully used as an indication of CL since it is hard to isolate atomic tasks in realistic settings dominated by multitasking. With respect to our own objective where perception is central, we don't need to quantitatively measure CL but rather just identify levels of CL usage(e.g. identify cognitive overload). Humans can easily perceive their mental level of difficulty [Turner 96], especially when asked to make a conscious effort to do that (introspection). For example, the user may be following a call hierarchy, he will be more able to describe that the task of following a deep tree is disorienting and exhausting.

### ***3.1.6 Efficiency increase and CL***

As we mentioned, our second theoretical assumption is that an RE tool can increase efficiency by minimizing the CL required for the SM tasks. We next describe the theoretical framework of how a tool can do that.

The general theoretical framework that describes how a tool can reduce CL is the distributed cognition model [Flor 91]. If we consider the reverse engineering (RE) tool and the human cognitive system as two processing/storage nodes in one system, with the tool

as the resource-abundant and inexpensive node and the WM as the opposite, then the problem will be reduced to moving load from WM to the tool.

In the design and implementation stage of a tool, the goal should be that the tool "sub-contracts" from the WM whatever possible sub-activities it can. This can be compared to using a hand held calculator as an external aid to sub-contract some of the processing load (the arithmetic) of a larger mathematical problem. Another example is using paper to store intermediate results of multiplication, instead of storing the results mentally.

Cognitive task analysis should identify, among other things, the implicit processing constructs and operations that go on in the WM, (e.g. mentally constructing a call hierarchy). The tool should take over some CL by explicitly representing the mental constructs using its processing power and the screen display (e.g. constructing and displaying the call tree on the screen).

## 3.2 The approach

Based on the above considerations and justifications, we propose a cognitively based approach for RE tool design that we argue will increase the efficiency and the potential adoptability of RE tools, therefore reducing the SM cost.

The first step in our approach is to define and focus on a certain context and deeply analyze it so a precise problem space is defined in terms of a set of inefficient SM activities within it. The context definition requires sampling in an industrial setting in order to reduce the complexity of the problem space into a manageable form.

Since our goal is to improve adoption by improving perception, the choice of context (and the case study in general) has to be one in which there is a strong perception of inefficiency in SM tasks. Such situations are identifiable by empirical/ecological proactive techniques such as external observation of SE work practices, by asking the SE to identify them, or even by introspection [Lakhotia 93].

Out of the context definition and analysis, we define a model of SM to be the basis for further steps. We call this model a *task view,* which is an abstraction of the chosen context that describes the major activities of SM as we perceive them. In general, a model is a simplification of reality in a way that facilitates analysis and keeps the focus on relevant details. Both the abstraction level and the focus of our model would be geared toward the identification of major difficulties in SM.

A special focus in the construction of the task view is given to the tasks associated with cognitive overloads (situation where the usage of memory resources is approaching capacity) in order to reduce these overloads. Cognitive overloads can be identified by external observation because they can be perceived as such. Even if one argues that some difficulties cannot be perceived, we say that we are interested mainly in what is perceived. This is because it is our goal to enhance adoption that, in turn, dependents on the perception according to the technology acceptance model of Davis [89].

After the identification of the task view, the tasks of the model should be cognitively analyzed to produce a difficulty model. Such analysis is done by breaking down the targeted tasks into their elementary micro activities to the level where they begin to have psychological significance (in terms what is discussed in the literature of this science). At such a level, psychological knowledge can be used to identify the root causes of overload.

From these causes, we also use psychological knowledge to derive an explanation of the difficulties. This explanation will be the central design theme that will shape the solution. Here, the difficulties and their cognitive sources have to be analyzed in terms that explain why they exist and how they can be alleviated --the cognitively based remedies for these causes. It is also here where all the theorization will take place, where the information collected in previous steps are put together in one meaningful theory that will embody within itself the rationale of the requirement and design of the solution. We call this phase the theoretical explanation framework; “framework” is better term than “model” because a model is an abstraction of complexity while a framework is a body of guiding knowledge.

The suggested explanation of the source of difficulties would be, then, easily translated into some requirements for the tool to satisfy. The requirements should also cover additional issues to ensure that the tool will meet its high-level objectives.

Finally, an RE tool would be designed with features to satisfy the cognitive requirements. The design of the tool does not map directly to the requirements. Rather it is a creative process similar to any other software solution for a problem, except that the cognitive factors should be highly influential when making various design decisions.

Evaluation and iteration based on evaluation findings are classical steps in software development that we also include in our approach. Note however that since tasks and difficulties models are documented, their details will also be revisited in any iteration so as to investigate if any of their assumptions could have caused weakness that manifested in the evaluation. That is, if the modeling is inaccurate then the solution will also be – what needs to be fixed is the modeling so the tool can be improved.

### *3.2.1 The cognitive approach in summary*

In summary, as illustrated in Figure 3, our cognitive approach involves the following steps or phases:

1. Sampling a context from an industrial setting where there is perception of difficulties.
2. Abstracting the tasks in this context into a task view that focuses on the tasks that most proclaim the perceived difficulties.
3. Performing cognitive analysis on the tasks of the task view to identify a set of conjectures about the sources of the difficulties in the task view. The result would be the *difficulties model,* which represents our perception of the major cognitive difficulties in the task view and their cognitive origins.

4. Establishing a theoretical view of the explanation of the causes of the difficulties model that acts as a framework to infer a set of requirements to be included in the tool's design.
5. Beginning the tool development cycle by generating requirements, implementing the tool, evaluating it and iterating through the steps of the approached based on the evaluation result.

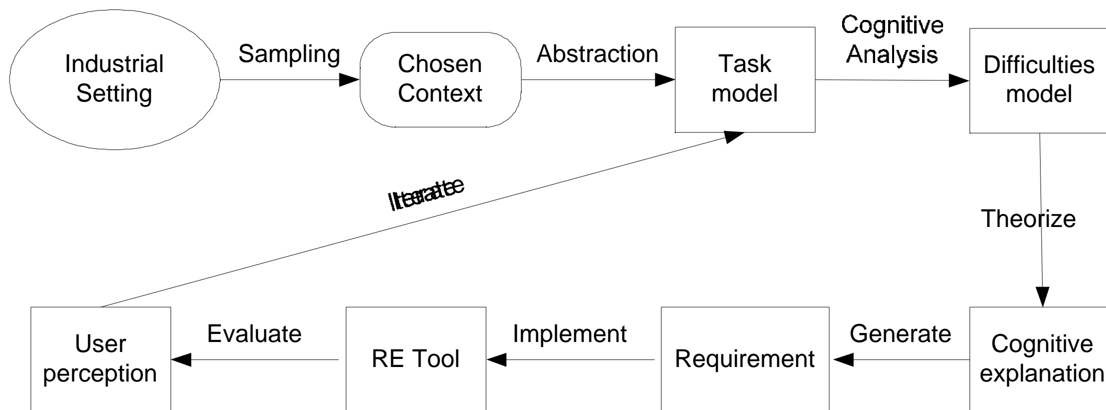


**Figure 3: The phases of our entire research approach.**

The case study presented in the next part of the chapter will best illustrate the approach.

## 3.3 Our approach at work: the case study

We applied our approach in designing a RE tool called DynaSee that we will describe in Chapter 4. In this section we will describe how we executed the approach as a case study in a telecommunications company that maintains a large legacy software system. The company suffers from the high cost of maintaining the system, thus we want to produce a reverse engineering tool that reduces that cost.

### 3.3.1 Identification of a context

As our approach suggests, we have to face problems in their own realistic setting. Within the complicated SM environment we faced in the targeted company, we used sampling to define a context that corresponds to a manageable problem space. We wanted this space to represent the worst-case mixture of environmental parameters.

First, we chose a specific system, a set of SEs to observe, and a subset of activities that these SEs perform on the selected system. This is because the size and quality of the system, in addition to the degree of expertise of the SEs, affect the nature of maintenance tasks.

The system we chose is a telecommunications system, initially written in 1982, that includes a real-time operating system and interacts with a large number of different hardware devices. The system is written in a proprietary structured language and contains several million lines of code. The chosen tasks are small corrective maintenance tasks. Such tasks are often assigned to an entry level SE who has little understanding of the structure of the system.

Our choice for this context of tasks and persons coincides with Jorgensen's findings [Jorgensen 95]. In his empirical study of SM, he concluded that the combination of corrective maintenance and less-experienced programmers working on a large system generates the lowest level of productivity in maintenance work.

### 3.3.2 SM process as we perceive it: the task view

In order to characterize SM in the chosen context, we applied proactive methods as our approach suggests. We began by observing the work practices of the targeted SE doing targeted tasks on the targeted system. We discussed with them their issues and problems and validated our conclusions. The author's experience in a similar context was also useful in the definition of the task view. It is important to note that this view, which we will present next, does not relate to any formal modeling nor does it represent a general or

complete description of SM. It just represents our choice of a set of hard and typical SM tasks identified in our investigation that are abstracted to a level appropriate for the case study.

In our task view, an SM activity is typically initiated by a maintenance request. The first maintenance activity is to understand the maintenance request by reading its description and/or by reproducing the problem.

Next, the code relevant to the problem has to be located. Given that maintenance requests are usually expressed in a very domain-oriented language, a process of converging (homing in) on the relevant code begins.

The primary method for converging is that a starting point in code has to be located. The starting point typically is a snippet of code that is part of the execution path of the current problem.

Locating a start point can be a difficult task if not enough pointers or cues are provided in the problem logs (e.g. call stack dump if the maintenance request is a trap) or if the SE has no information about the possible location of the relevant code. The SE searches for different cues in the source code; such cues include strings from the user interface, strings in comments and symbols names (e.g. variables and routines). Returned matched lines from the search are evaluated for relevance, not only by examining the lines but also by examining their neighbourhoods.

Alternatively, if the SE has an idea of about the possible location, he may scan the suspected modules looking for indications in the comments and in the symbol names that indicate or refute relevance.

Once the starting point becomes known, further convergence on the problem is done. Typically, the execution path is followed (beginning from the starting point), by continuously following (synchronizing) the events in the program behaviour inside the

code (mapping behaviour to code) until the events directly preceding the maintenance request are identified and thus the problematic code is localized. In fact, the synchronization must identify all relevant code, not only that which needs to be modified but also all the code that needs to be comprehended for a proper modification.

A particular observation at this phase is that control flow is traced at the level of routines more than at the level of individual lines (statements). The SE follows the routine call hierarchy at varied levels of depth, drilling down when a routine is not clear or when it is particularly relevant, and moving forward otherwise in the same level of the call hierarchy. The exploration focus is on routine names while individual statements and code comments are only looked at when the problem is identified within a routine or when the routine name is too ambiguous to tell about its functionality.

Once relevant code is identified, different strategies of code comprehension are used. In particular, bottom up comprehension is used for the statements suspected to be directly responsible for the maintenance request. The code statements are mentally visualised as executed (symbolic execution) and mapped with the problem behaviour.

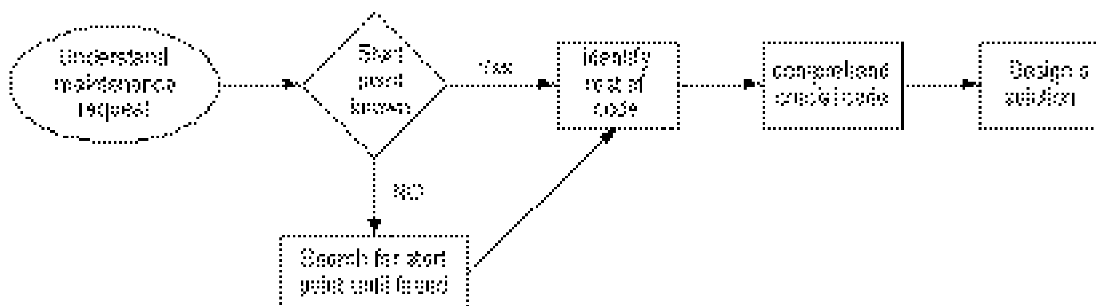


**Figure 4: The SM phases in our task view**

### *3.3.2.1 Analysis of the task view*

SM is a problem-solving process driven by the practical needs of the problem. Basically, the cycle begins with a maintenance request that is typically written in a domain language, i.e. a certain abnormal behaviour that needs to be altered into another desired behaviour.

The change of the program behaviour (the solution), will be ultimately made in some parts of the code. So a mapping between the problem behaviour and the corresponding code causing the problem is the obvious thing to do. This mapping, however, is by itself a challenging and typically an inefficient process especially when dealing with large software systems.

Moreover, with large systems, new constraints are added to the mapping and comprehension tasks. A full comprehension of the system is impossible; comprehension is done partially for the parts that directly affect the problem at hand.

The difficulty of mapping and comprehension in large systems implies an increase in the role of intermediate tasks that has to precede and guide partial comprehension. Namely, there is a central role for the process of locating relevant code for a maintenance request, what we designate as *manual slicing* (as opposed to slicing done by tools as in section 2.2.2). Manual slicing (hereinafter slicing) of code for a problem achieves the mapping requirement and thus identifies the parts that need to be comprehended. Typically, the path of execution (the executed code) related to the maintenance request represents most of the slice, or at least are good start points to find other related code.

### *3.3.3 The difficulties model*

The task view clearly present slicing as the major task that needs attention since it is a key activity that is difficult and different in large legacy systems. At this phase of defining the difficulties model we identify why slicing is difficult.

Slicing is done by synchronizing program events with code. In turn, synchronization is composed of two interwoven processes: locating delocalized pieces of the slice and comprehending the slice. In this section we try to analyze why slicing is such an inefficient task.

#### *3.3.3.1 Code location*

Locating pieces of code that are relevant is done in large legacy systems (LLS) using search, either for the first piece of code (starting point) or for next piece on the control flow. However, search is inefficient in large legacy systems and when combined with other tasks, it weighs on their efficiency in a very negative way. Search is heavily needed in large legacy systems because complete comprehension, exhaustive reading or scanning of code is non-feasible.

The main cause of the high inefficiency of search is that there is no explicit mapping between behaviour and code (static to application domain). Most often, because of the lack of reliable structure in LLS, the whole system needs to be searched for cues, and the search yields too many false hits (low precision). An excessive amount of time and effort is consumed to evaluate the results. This degrades the performance of the higher-level slice comprehension process, as we will discuss next.

#### *3.3.3.2 Slice comprehension*

The biggest problem in SM activities during slicing by search is the WM capacity. Tasks are more demanding than can be easily handled by the limited WM, thus causing frequent forgetting of vital information. This is because the information requirements (that need to be together in the WM) are extensive, thus cognitive overloads occur and degrades the performance of SM.

The scale of the overload problem was very clear from observations. In [Lethbridge 2000], Lethbridge describes several problems with exploration by searching. Most of these problems were related to the ability to remember where different search results (snippets of code) were stored. Lethbridge observes that some users (the SEs) use paper or editor buffers to save search results; however, he concludes that, “neither of these solutions is entirely satisfactory due to the overhead of doing the saving, and then finding the data again later.”

Additional signs of memory overload were shown by Lethbridge when he notes that users lose track of what they have to do and of their mental models when changing the context of work from one task to another, before completing the first. He points out that users have problems recalling their plans and models upon returning to an earlier context.

We identify the following cognitive causes of difficulties in slice comprehension that result in cognitive overloads:

#### *3.3.3.3 The delocalisation of code*

Delocalisation of code occurs when different pieces of code are logically related but physically separated. For example, comprehension of code that executed in a certain run of a program requires that comprehension follows the logical relation created by the execution flow between routines that are stored all over the software system (in different files and directories). The linear order of code execution has to be mentally reconstructed from delocalized static code.

Unlike text comprehension, where information exists in largely linear order, code comprehension requires an understanding of not only the looked-at artefacts (code objects that are the focus of reading), but also other delocalized artefacts. The comprehension process requires finding each one of the delocalized artefacts, retaining it in the WM, and then finding the rest so an overall map of the relationships can be mentally constructed. In moving from the acquisition of one artefact to another, the big problem is in the limitations of the WM, it either fades or simply cannot keep all of the required artefacts. The psychological support for this claim will be presented in Chapter 6.

#### ***3.3.3.4*** *Deeply nested relations*

Often, to understand a code artefact, it is necessary to understand some of its nested components that in turn depend on their own nested components. Deep nesting of this kind is particularly challenging and occurs mainly in routine call relations, type definitions, and inclusion relations.

For example, understanding a routine typically requires understanding the function of some of the routines it calls, and the same applies recursively for the called routines. Similarly, understanding a structured type definition requires an understanding of some of its fields, which may also be of compound types. The depth of routine call nesting is made even worse because, in our subject system, parts of the operating system are indistinguishable from other source code. Therefore, tracing functionality that requires an operating system service may also involve tracing the operating system routine calls that are used.

Deeply nested relations become particularly problematic because the atomic meaningful unit that needs to be formed from the nested relations is often larger than the WM capacity. The notion of a cohesive and connected structure of information is well studied in text comprehension literature. Turner [96] in his model for text comprehension, describes the input cycle – the number of propositions accumulated in WM before moving it to the long term memory (encoding). He notes that the reader will keep inputting propositions until a connected chain of information has been constructed (so that its gist can be extracted for encoding) or the WM limit has been reached.

#### *3.3.3.5 Uncertainty*

Another cause of difficulties in slice comprehension is manifested in the exploration aimed at following the execution path in code (control flow), which is a major activity within slicing. The difficulties stem from the uncertainty involved in determining the path that is actually followed.

While the SE is only interested in the path that produced the behaviour under investigation, static source code does not correspond to one possible control flow but normally allows for many different execution paths. It is often not possible to identify the desired path from source code with complete certainty because control flow depends on variables whose value may not be known until run time. This makes code exploration harder because it has to be carried out under uncertainty, thus many alternative paths have to be considered. This creates a need to keep track in the WM of additional information

required to backtrack from one path of exploration to another when a path is judged to be unlikely to correspond to the actual execution path.

According to Lloyd [99], who referenced classical work in psychology: “Results from applying information theory on human psychology strongly indicate that the number of alternatives, and the amount of information are crucial to both the time required to comprehend information and the degree of success attained in understanding information.”

##### *3.3.3.6 Low meaningful encoding*

The cost of symbols (code artefacts) that need to be maintained in WM can depend to a greater or lesser degree on how or whether they can be related to existing (learned) knowledge. By encoding the new information in terms of existing knowledge, meaningful encoding dramatically reduces the WM load [Ashcraft 99 p.103]

What makes meaningful encoding particularly low in our context is that in slicing, code needs to be dealt with outside its context. The delocalization of code, and the need to deal with search results that can come from anywhere in the system, lessens the ability to use contextual knowledge (e.g. functions of the current module) to relate to existing knowledge.

### 3.3.4 A framework for the explanation of difficulties

We see that the general framework that explains comprehension tasks and thus their causes of difficulties is the three domain mapping theory that we will explain next. We look at program comprehension as navigation in a software knowledge space within and across its domains. In our view, the software knowledge space can be seen as having three domains, each one with its own set of useful information:

- The static domain, which consists primarily of source code including comments plus any additional documentation that describes the design and implementation of software. The information in this domain is always available, fixed, and explicit.

- The dynamic domain, which concerns the information and knowledge about what is taking place between code artefacts during program execution, e.g. control, call and data flow. Inserted print statements, program traces and debuggers are the primary ways to obtain an insight into this domain. However, most of the time the knowledge of this domain is mentally induced from the static source code.

- The application domain, which concerns the external behaviour and the functionality of the program. This domain includes whatever is visible to the user, such as the user interface and the program output as well as any detectable event in related application software or hardware.

During SM, the information requirement for maintainers spans the 3 domains in an opportunistic manner. Maintainers need to cross reference information from any domain into the other two domains. The cognitive difficulties of mapping between domains stem from the natures of these domains. While both the application and static domains are explicit and visible, the dynamic domain is implicit and invisible.

The dynamic domain is the intermediate domain that allows cross referencing between the other domains, hence it has to be mentally constructed for the other domains to be bridged. The mental construction is done, in part, by the mental reordering of code to the sequence that it follows during execution. For example, to find what code (static domain) produced a certain functionality (application domain), the execution path (dynamic domain) corresponding to the functionality has to be identified.

Slicing and slice comprehension very much correspond to the mental construction of the dynamic domain. This is because they involve finding and comprehending the parts of codes that executed to produce the behaviour of the problem.

### 3.3.5 *Tool requirements*

The tool requirements are the results of what we discussed above. The tool should in general reduce CL by subcontracting the mental construction of the dynamic domain by reordering the code according to the dynamic order and by explicitly presenting it in a way that supports comprehension. However, the new form in which the dynamic domain is presented should be usable and should not incur significant additional overhead that may mask its utility (create new sources of load that significantly increase the overall CL).

In particular, the tool should address all the sources of overloads identified in the previous steps in order to reduce the CL in the tasks that are cognitively overloading:

1. Reduce the need for search aimed at locating delocalized code or increase the search efficiency.
2. Enhance slice comprehension by reducing the negative CL effects caused by the issues presented in the difficulties model such as delocalization, deep nesting, uncertainty and low meaningful encoding.
3. Enhance comprehension by supporting domain tracebility (mapping).
4. Afford exploration flexibility, at least to the level that is available to the SEs without the tool. For example, if they do not want to look at low level calls, they should not have to see them unless they ask for them.

## 3.4 Evidence and support

In this section we present support for our research assumptions from the literature and other reported experiences.

### 3.4.1 *Experiment and introspection*

The way we viewed software is not unique; several references in the literature support our view. We find the conclusions of Lakhotia [93], who experimented on SM, remarkably close to our conclusions. Although he used introspection, which is usually not well respected scientifically, for identifying the characteristics of SM during his experiments,

his conclusions are, in our opinion, much more valuable than many of the sophisticated pieces of empirical work that have been analysed using rigorous statistical techniques. This opinion is in line with our support for the ecological approach where one has to be actively seeking problems instead of depending on confirming or disconfirming hypotheses. Moreover, introspection is viable because it allows a person to be aware of his goals rather than staying at the level of externally observable techniques.

Lakhotia [93-b] designed an experiment in which he had to modify a program of medium size. He executed the experiment while introspecting about the nature, tasks and information requirements of SM and program comprehension.

### *3.4.2 Locating pieces of code*

In his description of his finding, Lakhotia mentions that the first step is, "to locate the places where a specific behaviour is implemented." To do so he scanned and searched the code looking for clues and says, "the symbols and the comments in the program were used to locate the code segments that were relevant to the change." He describes the convergence on related code by saying that, "the symbols acted as 'beacons' that helped us in 'homing in' on the relevant code." He also notes that he heavily used "grep" to search the code. He stresses the importance of the task of locating pieces of code that implement a piece of functionality noting that, "very little research has been done to support this activity."

He concludes that: "We don't always need to understand the design of the whole system to change it correctly" and the extent to which a program is understood depends on the "amount of functionality" of interest to the programmer.

Some of his other conclusions that support our assumptions include:

- The importance of the call tree exploration. He mentions that in exploring code, he "visited only what is called by or calls the routine."

- The severity of the gap in perception between the real SM problem and the reality. Before conducting this experiment, Lakhotia was trying to develop a tool to aid in SM. After the experiment, his perception of user needs has been changed dramatically.
- The lack of comprehensive analysis and classification in SM. In [93] Lakhotia notes that: "There is a need for classifying, at various levels of detail, the tasks performed by a maintenance programmer" and that while discussion for some of the tasks exists, they may be only found scattered in the literature.

### *3.4.3 Cognitive models and domains cross-referencing*

Despite the difference in details of the various cognitive models, there is a common theme that coincides with our theoretical framework of comprehension and explanation. We note that all models include, as part of comprehension, the mapping (matching) or cross referencing between code and high-level domain concepts through intermediate domains or models. This mapping can either originate from world domain to code (as in top down models) or from code to domain (in bottom up). The integrated model [von 95] considers top down and bottom up as different strategies that can be used in an opportunistic way depending on whether the code is familiar (top down) or unfamiliar (bottom up).

Pennington who produced a bottom up model [Pennington 87] says that two distinct but cross referenced mental representations of the program are constructed during program comprehension: a) Program models that highlight the procedural relation between program parts in the language of the program, and b) Domain model that highlight the functional relation between program parts in the language of domain objects.

She theorizes that effective comprehension requires the two mental models to be cross-referenced in a way that connects program parts to domain objects. Her analysis of experiments performed on 40 programmers suggests that the best comprehension is achieved when cross-referencing strategies are utilized. In such strategies, the comprehender tries to relate program text to domain function. When a 'trigger"

(equivalent to a beacon) in text is identified, the programmer forms a hypotheses about why the program wants to do something in world terms, and then try to verify the hypotheses in text.

Programmers who have been program-level comprehenders only, understand "what" the program does but not why.  While domain-level-only comprehenders have vague ideas about "why" but no idea about "what" happens in the program.

Brooks [83], whose model is top down, defines programming as a mapping from the problem domain into the programming domain through several intermediate domains. Hence, he defines comprehending to be the reconstruction of part or the entire mapping. He writes, "The task of understanding a program becomes one of constructing enough information about the modeling domains that the original programmer used to bridge between the problem and the executing program".

He describes the process of comprehension as a formulation of a hypothesis about the program functionality and then attempting to verify it in code. If the hypothesis is not detailed enough to match with code, sub hypotheses are created continuously until the bottom hypotheses can match against the code to be confirmed or disconfirmed.

In short, both theories regard comprehension as a mapping between code and application domain via intermediate abstraction and transformation. They disagree, however, on which is the source and which is the destination.

Finally, the three models of the integrated model of von Mayrhauser and Vans [von 93] loosely match our three-domain categorisation. The static domain may be identified with their program model, the dynamic domain with the situation model, and the application domain with the domain model.

### *3.4.4 Support for the difficulties model*

The fact that large systems exhibit different challenges and different ways to look at problems has been noted by von Mayrhauser and Vans [von 95]: "while a lot of important work exists, most of it centers around general understanding and small scale code" and, "theories regarding large-scale program comprehension for specialized maintenance tasks are in their infancy."

Kozaczynski [89] argues that the crucial difficulty in maintenance work is that change and modification requests coming from the user are most likely expressed in a very domain-specific language. Considerable difficulty is encountered given that the actual modification is to be carried out on the source code, and the mapping between domain and code may not be trivial.

Corbi [89] also identified the use of domain language as a major source of difficulties. He also notes that, "studying the dynamic behaviour of a program can be very useful and can dramatically improve understanding by revealing program characteristics which can not be assimilated by reading the source code alone".

## 3.5 Generalization for cognitively based assumptions

Many of the software related tasks or problems targeted by software engineering researchers have mental performance as the bottleneck. These include, among others, program comprehension and hyperspace browsing. So it is natural for someone working on these tasks and problems to try to develop a better understanding of the human cognitive system and how to enhance its performance. While the assumptions and techniques we provided are context specific, we hope that the generalization provided in this section will transform them into general cognitively based design and evaluation guidelines that can be consulted in any cognitively intensive application. Specifically, we try in this section to highlight some of the knowledge areas of psychology that may be illuminating for the tool designer who is designing tools to support mentally intensive applications.

An additional objective is to relate our cognitively related assumptions to mainstream psychology, thus enriching their value, since they become manifestations of well-accepted theories. We also try to show where similar techniques have been used or proved to be valid in different contexts and domains.

The following threads of literature were found to be useful: memory literature which is the heart of cognitive psychology; text comprehension literature which is a quite mature area aimed at pedagogic goals; as well as program comprehension, human machine interface, and data presentation research aimed at enhancing decision-making tasks.

### *3.5.1 Cognitive overload and its cost*

Some of our fundamental assumptions in this thesis concern cognitive resources; we assume, for example that high usage of these resources for some tasks will degrade higher order comprehension activities. In particular, we assume that the need to maintain a significant number of items of information in WM (such as search results and symbols) for a significant time and the simultaneous need (at the same time) to visualize call trees and control flow relations, will degrade performance.

These assumptions turn out to be highly generalizeable because they greatly coincide with a new model of WM. This model is also supported by experimental results, as we will show in the next section.

#### *3.5.1.1 A model for the WM*

The view of WM has been evolving continuously from its primitive view as a short-term memory buffer that is only able to hold 7+-2 items of information. One of the most credible models for WM is that suggested by Baddely [Baddely 74, 86], the foremost authority on WM, who views it to be more like a mental workbench.

Under Baddely's model, the WM is made out of a central executive and two slave subsystems: the articulatory rehearsal loop and the visuo-spatial sketchpad. The central executive is thought to be the primary workbench area of the system where mental work of all sorts is done. It initiates a variety of mental processes, such as decision making, retrieval of information from long-term memory, reasoning and language comprehension.

The articulatory rehearsal loop is a sound-based system that can hold and recycle small quantities of information; it corresponds to a short-term rehearsal buffer. The visuo-spatial sketchpad is a specialized slave system that holds visual or spatial codes for short periods of time.

More importantly for us, Baddely considers that the central executive can be thought of as a pool of mental resources available for any of several different tasks but which is limited in overall quantity. Each of the two slave systems also has a limited pool of resources. However, resources are shared in one direction, from the central executive down to either the articulatory rehearsal loop or the visuo-spatial sketchpad.

The central executive shares its resources with the slave systems when either one of the slave systems becomes overburdened (with an overly demanding task) and needs extra resources. However, when the central executive shares its resources, it often ends up having insufficient capacity to do its own work.

This theory has been tested empirically. Experiments have focused on proving that there are separate subsystems that can work independently when none is overloaded. This was done by having the central executive perform some mental task, then giving a second task to one of the slave systems. As the tasks become more and more demanding, interference effects showed up, usually as a slowing down of performance or as an increase in errors.

For example, in one of Baddely and Hitch's [Baddely 74] earliest experiments, experimental participants were asked to hold randomly chosen letters or digits in the short-term "buffer" (i.e., the articulatory rehearsal loop). The other activity was a concurrent

(simultaneous) language-based reasoning task. That is, while several items were being held in the short-term buffer, participants also had to do a mental reasoning procedure (performed in the central executive). Results showed that as the number of rehearsed digits reached the capacity of the rehearsal buffer, the reasoning tasks began to be affected and performance degraded significantly.

##### *3.5.1.2 Relation to our work*

The similarity to our assumptions can be easily observed. When too many intermediate search results need to be maintained in the WM (rehearsal loop), some of the central executive resources are used (shared) causing the overall comprehension and other high-order processes to degrade in performance. The same applies for the need to visualize a call tree, when such a call tree is large enough, it will drain some of the central executive resources and thus also degrade the overall performance.

### 3.5.2 The implicit-explicit and cognitive distance

Increasing the explicit where there are extensive implicit operations was a cornerstone in our approach. In the requirement section, we required that the tool should reduce CL caused by the implicit dynamic domain by constructing an explicit representation of that domain.

##### *3.5.2.1 The implicit explicit dichotomy*

The explicit aspects of a task are its input information (raw problem representation); the implicit aspects are what is not tangibly presented and thus needs to be constructed mentally to perform the task. In the Shepard experiment, the object picture was shown only as it existed at the beginning and the end of the transformation – the explicit points, while the whole rotation path is implicit and has to be constructed or inferred mentally, within the WM.

A close analysis of this concept reveals a similarity to the classical psychology literature with the well-known "mental rotation" experiment of Shepard [Ashcraft 98 p.120].

#### *3.5.2.2 Mental rotation experiment*

In his experiment Shepard gave his subjects two pictures of objects; in some cases the second picture was a rotated view of the first object and in other cases it was a similar, but different object. The subjects were asked to determine if the two pictures represent the same object. Different subjects were given pictures with increased rotation and the time spent by subjects to find the answer was recorded. The results showed that the time spent on finding the answer is linearly related to the degree of rotation.

To illustrate the relation between a comprehension model and the explicit/implicit dichotomy, we look at Kintsch's model for text comprehension [Kintsch 98]. Kintsch's theory considers that a main sub-activity of text comprehension involves the mental construction of a macrostructure – a kind of outline of the text being comprehended. Under our assumptions, the explicit construction and presentation of the macrostructure should reduce the effort needed, and therefore increase comprehension. This is exactly what was demonstrated in an experiment by Beyer, as mentioned in [Kintsch 98 p. 309], who reported that comprehension was better when the macrostructure of the text was explicitly presented using headings and subheadings than when macrostructure was implicit.

Kintsch [Kintsch 98 p. 303] also describes an experiment where two pieces of text were presented: in the first piece, sentences were explicitly linked by prepositions, and in the other piece sentences were not. The explicit text was found to be easier to comprehend. Aschraft [99 p.298] also concludes that in text comprehension, the more indirect the reference, the slower and more difficult it is to comprehend.

Slovic [72] uses the notion of concreteness to illustrate the importance of the explicit representation of input data: "Concreteness represents the general notion that a judge decision maker tends to use only the information that is explicitly displayed and will use it only in the form in which it is displayed. Information that has to be stored in memory, inferred from the explicit display, or transformed tends to be discounted or ignored. "

Thuring [95], for example, noted that in hyper-document browsing, "readers need to keep track of their moves, which results in a considerable memory load". As a remedy, he suggests to use a graphical map in a navigation tool. The map would show all nodes visited and their relationships; this way the map would be an explicit counterpart of the needed memory load. In his opinion, this would help free memory from the burden of maintaining the map.

### *3.5.3 Gap in perception between user and tool designer*

The first theoretical assumption in this thesis was that reverse engineering (RE) tools' low adoption rate is largely due to a gap in perception of software maintenance (SM) problems between tool designers and the tool users – the SEs. We also suggested that there should be a characterization of SM's real difficulties and problem using a cognitive analysis of the nature of these problems. Finally, we suggested that requirements for tool design should stem from these requirements.

#### *3.5.3.1 Cognitive fit*

The notion of cognitive fit developed by Vessey [91] constitutes an excellent generalization for our assumption. According to Vessey, cognitive fit exists when the problem solving aids (tools, techniques, or problem representations) support the task strategies (methods or processes) required to perform that task. Cognitive fit causes the complexity in the task environment to be effectively reduced.

Vessey draws on the problem-solving literature and the notion of cognitive effort to illustrate cognitive fit. Vessey's model for problem solving views it as an outcome of the relationship between problem representation and problem solving tasks. The mental representation is the way the problem solver represents the problem in human working memory. It is formulated using the characteristics of both the problem representation and the tasks.

Vessey conjectures that adoption of problem solving aids results from fit, which is the degree of match between the processes that one uses to act on the representation and the processes and strategies that act on the solution. Greater fit implies less cognitive effort in translating between mental representation of the problem and solution strategies.

Vessey's cognitive fit experiments were focused on data representation. He empirically showed, for example, that tabular presentation leads to improved decision making performance for certain tasks and lower performance for others. The opposite applies to graphical representations: tasks in which tables fared badly tended to result in better performance when presented graphically, and vice versa.

Cognitive fit had a ripple effect in the MIS literature. Based on cognitive fit, the notion of *task technology fit* (TTF) [Dishaw 98] became a model to predict adoption. TTF is defined as, "the degree to which a technology assists an individual in performing his or her tasks". In the context of a tool, TTF can indicate the correspondence between task requirements and the functionality of the tool. Dishaw and Strong [Dishaw 98] conjecture that a higher degree of "fit" between task and technology leads to increases in both utilisation and performance. On the other hand, Goodhue [95] conjectures that a higher degree of "fit" leads to expectations of positive consequences of use. As such, TTF becomes aligned with other social psychology theories that emphasise expectation as the driver for behaviour.

In revisiting the correspondence between these theories and our work, we note that, in the cognitive fit theory, the cognitive effort created by low fit can be considered as an instance of a more general notion of cognitive load as we used it in the thesis. TTF is a direct generalization of our requirement to have accurate characterization of users' tasks and to match these tasks with the tools' capabilities using cognitive load as the main criterion to increase the match.

### *3.5.4 Problem solving*

Perhaps the most general umbrella in psychology that covers our work is the area of *problem solving*. This is described as the study of individuals confronted with a difficult, time consuming task where the solution is not immediately obvious and the person is not certain what to do next [Ashcraft 98 p. 383]. Almost every textbook on psychology has a chapter about problem solving. SM and computer programming in general are considered to be a set of problem solving tasks by experts in domain specific knowledge [Brooks 83]. The problem solving literature suggests many principles to enhance the efficiency of this process. One of the most important principles is to support the WM. Obviously this support was central in our thesis, we discuss other principles below.

### *3.5.5 Automated processes*

In addition to supporting WM, the other major principle suggested in psychological literature to enhance problem-solving capabilities, is to automate some component of the problem solving solution by increasing the role of automated processes [Ashcraft 99 p.412]. This should not suggest that these two principles are totally separate; rather that automation can also be sought as a way to relieve scarce cognitive resources, notably the WM.

In general, tasks and activities both at the macro and micro level, belong to two different camps: a) automated and unconscious, and b) intentional, serial and conscious. Automated processes are highly learned and utilized, such that they do not need a conscious effort to perform and thus they are not competing for scarce cognitive resources such as memory and attention. In fact, automacity is a continuous value, the more a process is practiced, the more it gains automacity and the less it consumes resources.

Only those tasks that are conscious are relevant to optimization. Completely automated tasks do not drain any significant resources from the scarce cognitive resources and thus several automated tasks can be executed in parallel [Ashcraft 98 p. 412]. Problem solving is considered an intentional conscious activity; however, it can utilize many automated

processes (micro activities). For example, writing programming code is an intentional task that utilizes other automated processes such as keyboard typing. Typing, in this sense, does not compete with programming for the limited cognitive resources.

The utilisation of this principle in design involves looking at the particular micro activities that result from a design choice and aim at increasing the automated ones and decreasing the non-automated (intentional) ones as expected to be found with the potential users. Lloyd [99] describes this process as mapping or fitting sub-tasks to prior knowledge and skills in order to use the automated capacities.

#### *3.5.5.1 Other encounters*

Kotovsky et al [85] tested adult subjects on various versions of the Tower of Hanoi problem; their results showed that a heavy WM load was a serious impediment to successful problem solving. As a solution they suggested to automate the rules that govern the moves in the problem, doing this frees WM resources to be used for higher subgoals.

Lloyd [99] uses the automation notion with regards to data format. Based on comparative empirical studies, he notes that, "certain formats for presenting information are more automated than others (horizontal and vertical lines are better than curved lines and usage of space, and punctuation is better when at the end of line)". He proceeds to claim, "human perception is wired to detect certain symbols more readily than others (e.g. grid structure as a good example)".

In fact, Lloyd extends the role of automation when he suggests avoiding using automated processes in different ways from those that have been learned. He calls this, "unlearning highly learned processes". For example, using the scroll bar in a UI, which is automated for view scrolling, for a totally different purpose results in increased cognitive cost.

### *3.5.6 Increasing recognition vs. recall*

After supporting the WM and automating processes, a third important principle from the problem solving literature to enhance performance is the one related to the difference between the cognitive cost of recognition and recall. The trade-off between recognition and recall has a close proximity to the previous trade-off between automated and deliberate processes.

Recognition is largely done at the level of the perceptual system; therefore it is less cognitively demanding than recall, which involves memory retrieval. In fact, the cognitive system is fundamentally parallel in its recognition phase and fundamentally serial in its action phase [Card 83 p. 42]. Thus the cognitive system can be aware of many things but can't do more than one deliberate thing at a time.

A clear manifestation of this notion is in the successful use of menus in graphical user interfaces. Menus are successful because invoking a command via a menu involves recognition of available commands; without a menu, extra efforts are required to remember a textual command (to recall it).

In our RE tool described in the next chapter, we extensively used this notion, although for different reasons. For example, putting related software symbols (such as the routines of a call tree) on screen instead of forcing the user to remember them is an example of how we increased recognition on the expense of recall.

### *3.5.7 Facilitating meaningful encoding*

One of the causes of slice comprehension problems that we identified in our work was "low meaningful encoding". Meaningful encoding, which is sometimes called *recoding*, is a fundamental concept in cognitive psychology since it is the primary way by which the WM alleviates its limitation [Ashcraft 98 p. 102]. Recoding is the general process by which WM can overcome its capacity limitation by chunking separate items into groups. The success of chunking depends on the ability to relate the new information with learned

information (i.e. the familiarity of the new information) [Kintsch 98 p.330]. Thus, meaningful encoding or recoding occurs when a learned high-order meaning can be used to represent several items of information that need to be maintained. For example, the three digits “613” can be chunked into one item of information if there is the meaning that “613” is the area code of Ottawa.

As an example of the role of meaningful encoding in comprehension, Branford and Johnson [Kintsch 98 p.287] experimented with the role of titles in text comprehension. They observe that text was more easily understood when given a title. The title allowed subjects to use their knowledge and to disambiguate the otherwise obscure text. Without the title, the text was more difficult to understand since subjects could not interpret the situation as a familiar one; therefore, subjects could use their knowledge in the title condition but not in the no-title condition. The title helped activate the proper context in the long-term memory and thus facilitate the meaningful encoding.

### *3.5.8 Generalization for code delocalization and deep nesting*

In our difficulties model, we identified the delocalization of code and the deeply nested relations between delocalized pieces of code as causes for cognitive overloads. Related pieces of code that are delocalized or too nested may exhaust the memory resources while they are mentally brought together in order to be assimilated. This notion, as we show in the next section, is fundamental to text comprehension.

The successive acquisition of different pieces of information in order to form one larger meaningful structure is one of the basic activities in text comprehension [Thuring 95]. Many references in the literature address the necessity of forming a meaningful mental representation out of multiple items of information that need to be kept in WM or perceived together.

Turner [96] studied the input cycle during text comprehension – how many propositions to read before pausing and encoding the input. He noted that as more propositions are input

(read), the number of propositions in working memory increases. Decisions to suspend input come from two sources: a) if all propositions form a “connected chain“ i.e. a connected unit of information so that its gist can be extracted, or b) when the capacity of the WM is approached or reached.

Turner argues that a reader has some sort of awareness of how occupied his or her WM is: “Psychologically, this corresponds to the introspection that one’s working memory is approaching capacity, and the increasing sense that input will soon have to be suspended.”

Thuring [95] applied the same concept on hyper-document comprehension. He describes this meaningful unit formation as the basic process of comprehension in psycholinguistics (called “given new strategy”). He writes “in attempting to understand the content of a new node, readers try to extract its information and relate it to the content of other nodes they have visited.” He highlights the importance of being able to perceive all related information at the same time by saying, “when readers can see the given information of the previous nodes together with the new information of the current node they can detect semantic relations between both sources more easily.” Note that this last reference also supports the principle of increasing recognition versus recall, since “seeing” is a form of recognition.

### *3.5.9 The gap effects*

In describing what makes delocalized code particularly CL demanding, we mentioned before that the nature of locating another piece of code (the new information), that involves searching and evaluating search results, increases the degradation of the cognitive resources. Search creates a gap between the new and given information, this gap is of time and complexity.

Cognitive psychology confirms the conjecture of the time gap by indicating that information can be only maintained in WM for about 20 seconds or rehearsal (a demanding activity) has to be used [Ashcraft 98 p. 105]. Given that typically the search

and evaluation cycle takes more than 20 seconds, rehearsal can be of significant effect as we discussed when we generalized from the Baddely model of WM.

Cognitive psychology also confirms the conjecture of the complexity gap by indicating that if certain information is stored in WM and another task is initiated, the stored information becomes very vulnerable due to interference with the new information created by the new task [Ashcraft 98 p. 105].

Thuring calls this complexity gap *cognitive overhead* and defines it as, “the additional effort and concentration necessary to maintain several tasks or trails at one time” [Thuring 99]. He indicates that the primary way cognitive overhead affects cognitive performance is by interference and competition for the limited capacity of cognitive resources between different tasks. This is because each task needs to create and maintain its own memory context.

In the context of hyper-document navigation, Thuring looked at reducing cognitive overhead as a way to increase comprehension because, in his words: “every effort additional to reading reduces the mental resources available for comprehension”. He was primarily concerned with overhead caused by orientation, navigation, and user interface adjustment.

### *3.5.10 Conclusion*

The above generalizations are attempts to show that our use of psychology in this thesis is not a specific isolated case but rather a part of a larger well-founded picture. The ideas presented are opportunities to show how the domain of psychology can be of practical use for computer science and software engineering.

## 3.6 Generalization of the theoretical lessons

Finally, to prove the generalizability of our theoretical findings and conclusions, we apply some of these findings to identify difficulties in the object-oriented paradigm based on case studies of different authors.

### *3.6.1 Object oriented code*

The system that we targeted in our case study, in addition to the other systems used in DynaSee evaluation, were all written in structured (procedural) programming languages. An interesting question to answer would be how much of our findings in the case study apply to object oriented (OO) programming languages.

We base our analysis on a paper [Wilde 93] entitled “Maintaining Object-Oriented Software” by Wilde et al. The authors, in addition to their own experience in OO development, studied two large OO systems at Bellcore and interviewed their developers to come up with a set of observations and conclusions about sources of difficulties in maintaining OO code.

The authors begin by arguing “that a major goal of object orientation has always been to make change easier.” In their opinion, this can be achieved because, “objects in the program match objects in the real world more closely, so real world changes should be easier to map to program modifications.”

This clearly makes sense within our theoretical framework about domains and domain mapping. In our terminology, the equivalent of what they said is: since domain objects are realised directly in corresponding source code, mapping between static and application domain is easier because it is more direct and explicit.

They mention additional maintenance advantages of OO over procedural languages such as encapsulation that localizes the effects of change, and inheritance that facilitates code

reuse. On the other hand, they note “object oriented techniques may not make programs easier to understand.”

In explaining why there are such additional sources of difficulties, they used many arguments that fit within our own theoretical explanation for the difficulties in our work. Some of these sources not only still apply in OO code but also are exacerbated by object orientation, as we show in the following sections.

#### *3.6.1.1 Task view*

In trying to characterize how OO maintainers comprehend a program and what are their information requirements, Wilde et al. present a task view that is largely compatible with our model presented in this thesis.

Wilde et. al observe that comprehension is partial and that the maintainers try to “understand program behaviour well enough to modify it safely”. Following control flow in source code is found to be a central activity. In particular, tracing calling relationships was identified to have a major role in comprehension, “typically, a maintainer first tries to find where the fragment is called from to get clues about its purpose from its context of use.”

They also observe that when a maintainer needs to understand a fragment of code, he must locate other subroutines called by this fragment and examine them to see what they do and to trace the flow of data values passed to them. Tracing control flow in OO code is slightly different from procedural code since it involves tracing messages that tend to be more deeply nested; also the calling relationships are complicated by polymorphism . This tracing and study of code allows the maintainer gradually to gain a mental picture of the fragment's purpose and design.

Therefore we can conclude that OO program understanding also involves slicing in a similar form to procedural program understanding. In both cases, relevant code has to be identified by following control flow and sometimes data flow.

#### *3.6.1.2 Delocalization and atomic unit*

One of the major differences between OO and procedural code relates to the number and size of routines in typical OO systems. The authors argue that the nature of object orientation causes a, “profusion of relatively small program parts with many potentially complex relationships”. The large number and small size of methods (routines) in typical OO programs increases the number of relationships that a maintainer must understand.

As we discussed above, like in procedural programs, to understand the system behaviour the programmer must trace chains of methods related to the investigated problem. In performing this tracing, the problems identified in procedural languages are not only present but are also exacerbated by the small size and large number of methods.

More methods mean that a functionally cohesive chunk of information (an atomic meaningful unit) is made of more methods and their relations. This is similar to the deep nesting problem in our difficulties model, except that instead of nesting of routines, there is a “lengthening of the chain” of methods and relationships to follow.

Also for this same reason, delocalization increases: “a central problem in program understanding is the reconstruction of delocalized plan” and, “program plans are often dispersed through several non contiguous program segments.” In fact, the increase in delocalization is not only due to the size of routines (methods) but also due to many inherent properties of OO languages.

In procedural languages designed using functional decomposition, program functions are often localized in routines. In contrast, OO languages distribute program function across several classes and the functionality is achieved through the interactions among these classes. The authors note that, in addition to inheritance, “Polymorphism and dynamic binding create more opportunity for delocalization.”

#### *3.6.1.3 Uncertainty caused by inheritance, dynamic binding and polymorphism*

The uncertainty when following control flow in procedural code is exacerbated in OO code. In procedural code, uncertainty is mainly caused by the absence of dynamic information such as the value of variables that decide where the control flow would branch.

In OO code, in addition to these reasons, dynamic binding and polymorphism increase uncertainty because when a message is sent to an object it may be impossible, from the source code alone, to determine which of several methods will be executed and, "maintainers reading the code must consider all the possibilities".

Each time a message is sent, a programmer may have to examine several levels to determine how it could be handled by the receiver. Wilde et al determined that in their studied systems, "libraries frequently contain 10 or more methods that implement a given message". They conclude that while dynamic binding and polymorphism give flexibility for the evolution of software, they also make the program harder to understand.

#### *3.6.1.4 The dynamic domain*

It seems that the reasons that we have used to argue about the need for explicit representations for the dynamic domain in our targeted system are even stronger in the case of OO systems. There are additional sources of uncertainty that cannot be resolved until run time. There is also more delocalization as well as larger and more fragmented atomic meaningful units.

More generally speaking, object-oriented source code tells us less about the program's behaviour – the behavioural information is more implicit. That is, the cognitive distance is even larger since the procedural code parallels better (more explicitly) the function of a program and its dynamic behaviour. In OO code, although there is better mapping from application to static domain, the mapping from static to dynamic domain is less obvious and the mental effort in reconstructing the dynamic domain seems to be larger than in procedural code.

The authors of the paper that we used for analyzing OO code [Wilde 93] conclude that, "a difficult problem in understanding OO code is detecting and deciphering these interacting groups of classes. To do so, one must trace many possible sequences." They suggest using dynamic analysis tools to alleviate this comprehension problem; such tools might include animation techniques that animate the message passing between objects and classes.

### *3.6.2 Conclusion*

Most sources of difficulties identified in our thesis were also manifested and sometimes exacerbated in a different paradigm of programming (objected oriented) than the paradigm that we worked on (procedural). The same theoretical framework can be applied to OO systems where many of the existing findings can be reused and others need to be refined. A fruitful avenue for future research could be to revisit the findings from an OO point of view. The suitability of the call tree (that we will be using for procedural code) for OO code should also be investigated. Our conjecture is that the call tree would still be highly useful, based on our limited experience where we generated the call tree for some code written in Java. We also think that the call tree will be more useful if the classes to which the methods belong are shown in addition to the method name.

In fact, we believe that many of techniques that we defined or used could be even further generalizeable to all analysis of cognitive tasks in software in general. The implicit/explicit balance notion helps characterizing situations in programming techniques where cognitive overloads could occur. Take for example recursion, in which a few explicit statements create a large invisible distance to visualize, making comprehension harder. Another example is where a system is highly distributed and where process interaction and message passing can be highly implicit in the code. Again, too little is said in the code about what is invisibly taking place. Note that in our targeted LLS system, processes and message traces were indispensable for SM although they were of very low usability since they were text based.

# Chapter 4 Overview of DynaSee

In this chapter we present DynaSee, the reverse engineering (RE) tool that is designed to satisfy the requirements presented in the case study in the previous chapter. The satisfaction of these requirements implies having to solve some of the typical problems of dynamic program analysis related to making usable presentations for large volumes of dynamic data.

## 4.1 General functionality

DynaSee is a dynamic analysis tool aimed at facilitating software maintenance and program comprehension by bridging the gaps among the static, dynamic and domain views of software. It is a part of a larger program comprehension tool set named TkSee, developed by the KBRE group at the University of Ottawa.

DynaSee accepts, as input, a trace file that contains the names and call-levels (nesting level) for all the routines that are executed during a scenario. After processing the trace, DynaSee presents it as a call tree as shown in Figure 1. DynaSee also provides several features to facilitate comprehension and visualisation of traces in addition to domain mapping aids.

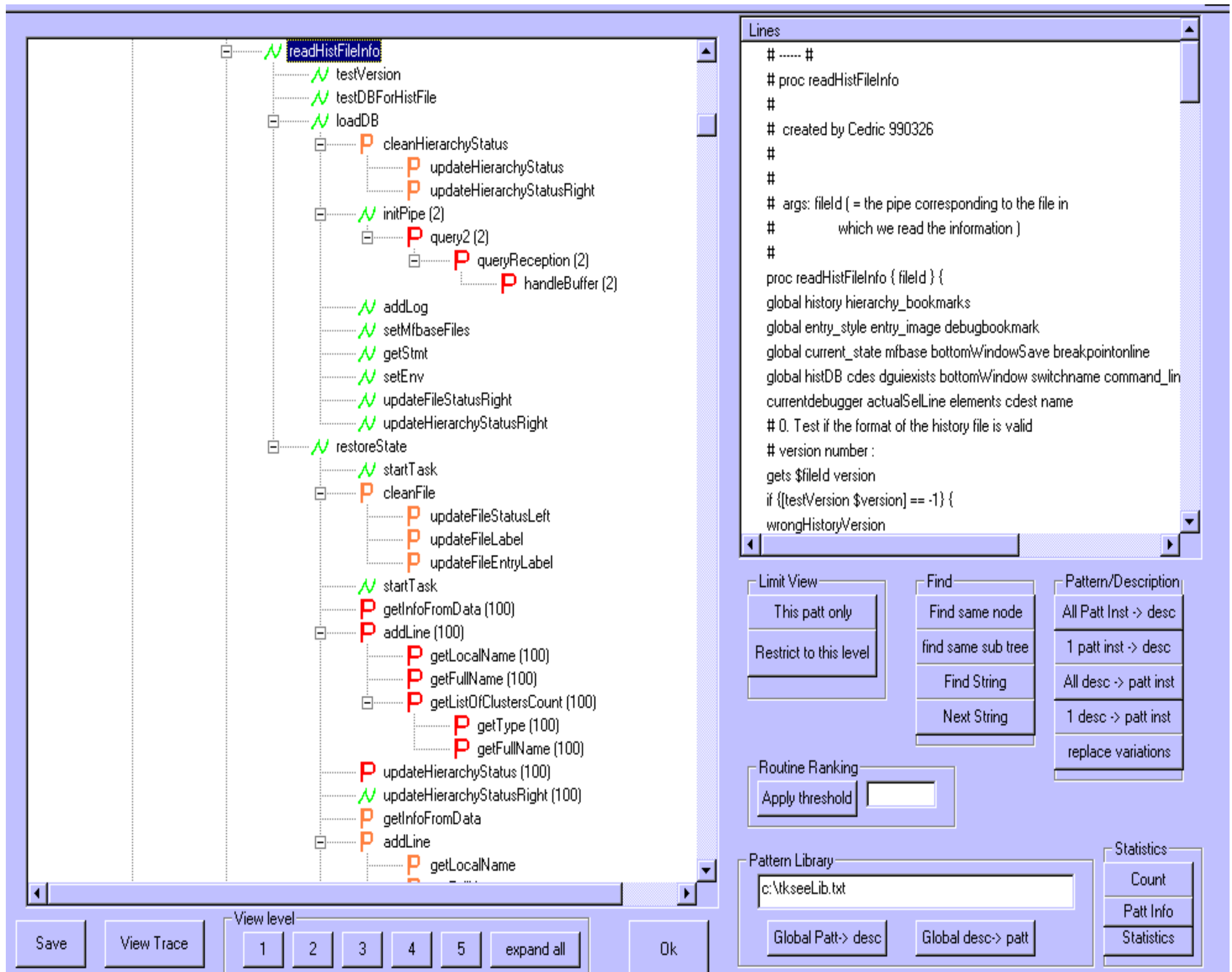


**Figure 5: A screen dump of DynaSee where the call tree is on the left and code window on the right**

## 4.2 Instrumentation

Traces can be generated in different ways; one of the most common is instrumentation. In our approach, the source code is instrumented by parsing the code and automatically inserting for each routine several print statements (probes) at the beginning of a routine and at each of its possible return points. Instrumented code is then compiled. The resulting application is run for several sessions or scenarios. When executed, each print statement (probe) outputs the name of the host routine and an additional character that signals if the probe is at the beginning or a return point of the routine. The probes are directed to a trace

file that will be used in building the call tree relation between the routines. Each trace file will represent a session and thus can be analysed separately.

## 4.3 The unit of instrumentation

Not all instrumentations are done at the routine level. The granularity of what is instrumented can vary from high level, such as a module, to the lowest level or the *basic block*. The choice of routine as the unit of instrumentation is supported by the fact that routines are the fundamental methods of abstraction in programming. A routine name is a compact description of the important functionality that its contained statements perform. This abstraction is created by the original software author who would typically have a clear idea about his intention.

Unlike many abstractions found in software documentation such as design diagrams or program comments, routine names are reliable and do not become obsolete easily as they are a compiled part of the source code. Routines can also be arranged in a semantically rich hierarchy of abstraction – the call tree – as we will show later.

Moreover, the choice of routines as the unit of tracing was also influenced by our task view presented in the previous chapter where we noted that SEs explore code in order to trace control flow and that they look at routine calls more than at other programming constructs.

## 4.4 DynaSee features

In this section we describe each of the various feature of DynaSee. Figure 6 below shows the various phases applied to a trace from program instrumentation described above to the call tree in the DynaSee features. Features can either be for preparing the trace for visualization (repetition removal, pattern detection, and routine ranking) or visualization features that are applied on the call tree itself.

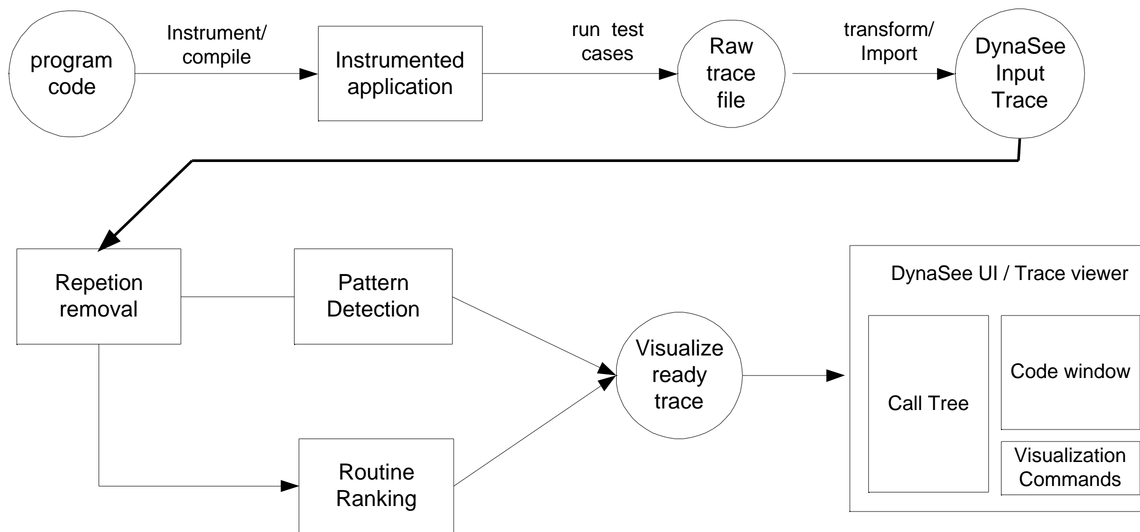


**Figure 6: Processing phases in DynaSee where a box corresponds to processing and a circle to a data in a particular format**

### *4.4.1 Repetition removal*

To improve the comprehension of the typically very large traces, we want to reduce the amount of data to be looked at during visualisation while preserving the information content of the trace. The primary way to achieve this is by locating redundancies and compressing them. An important source of redundancy is generated by the routines called recursively or inside loops. As the trace records each routine's execution, it will contain a trace line for each routine inside a loop at each iteration of the loop.

The first processing phase of DynaSee is to apply an algorithm that detects any contiguous repetitions of routine sequences created by calls to routines within loops or by recursion. Repetitive sequences are replaced with a single sequence with the number of removed duplicates concatenated to the routine name and enclosed by brackets (see Figure 8). For example, ABBBBC (each letter corresponds to a routine name) will be replaced by AB[4]C, and XABABABABY will be replaced with XA[4..][..B4]Y. Note that the bracket opens at an entry and closes at another to differentiate from representing the compressed form of  XAAAABBBBY that should show as XA[4]B[4]Y.

### *4.4.2 Patterns*

The order of routine calls in a program is not random; rather, several pieces of functionality tend to be executed repetitively [Jerding 97]. Each of these pieces is realised from the execution of a determined set of routines in a specific order. These sets may occur in exactly the same order or as different variations. For example, the "Save database" or “Initialise preference" functionality may be executed several times during a program execution; each execution generates the same (or similar) trace of routines.

We mean by a pattern in a trace any sequence of routine entries that occurs more than once non-contiguously (if two sequences are contiguous they would be simply considered as redundancy as discussed in the last section). Note that in this context, the word “pattern” is related to the general English definition of a pattern. In computer science, the word pattern, as used in the phrase “design patterns” has a meaning different from what we are discussing in this thesis.

We adapted an algorithm by Tseng [98] and used it to locate the maximal sequence of entries in a trace that occurs more than once. When found, these sequences are tagged so they would be visually distinguishable during trace visualisation.

#### *4.4.2.1 Visualization*

Patterns identified in the processing phase are visually identified within the trace viewer by assigning a special “p” icon to each node that belongs to a pattern. Different colours are assigned to different patterns to distinguish them. Also, a unique pattern number can be concatenated to the text of each node of each occurrence of that pattern.

Patterns at the syntactic level add little information to facilitate comprehending the trace besides indicating that a pattern sequence occurred more than once. However, it is this indication that gives a hint to the SE that the sequence may be a good candidate for abstraction. This is because if a sequence of routines recurs in several places, then it is likely to correspond to a common functionality or to a high-level concept. When the SE identifies this underlying concept during visualization, he can replace the pattern sequence

with an entered meaningful description. This problem shows similarity to the problem of identifying reusable components in source code using dominance tree analysis [Cimitile 95].

The entered description and the sequence that constitutes the pattern replaced by the description are saved permanently in an external file. This saving allows one to create pattern libraries that accumulate patterns, thus allowing the automatic substitution of patterns by descriptions when a library is applied to a trace.

So far, only the part of patterns that belongs to one subtree can be abstracted (replaced by a description) since this significantly simplifies the implementation. Very rarely we encounter cases where more than one subtree needs to be abstracted as one pattern.

#### *4.4.2.2 Related operations*

DynaSee offers the following operations on patterns:

**All pattern instances to descriptions:** The user selects the root node of a subtree of a pattern. All instances (occurrences) of the pattern subtree would be removed and replaced by a user-entered description.

**One instance to description:** Only the selected pattern instance is replaced with a description.

**All descriptions to pattern:** All occurrences of a selected description node will be replaced with the pattern that corresponds to that description; the nodes of the patterns will be restored exactly as they were prior to being replaced by a description.

**One description to pattern:** A selected description node will be replaced with the pattern that corresponds to that description.

**Global patterns to descriptions:** Replace all patterns in the tree with their descriptions; matches are based on the current pattern library file.

**Global descriptions to patterns:** Replace all descriptions in the tree with their corresponding patterns; matches are based on the current pattern library file.

**Show this pattern only**: After selecting a pattern root node, the entire hierarchy will be collapsed enough to keep the instances of that pattern visible.

#### *4.4.2.3 Pattern Variations*

The notion of pattern variations evolved from experiences with patterns. Many patterns having the same root nodes, with similar but not exactly the same subtrees (variations), were frequently present in the traces we studied. We consider the set of similar subtrees as a set of variations of the same pattern. In other words, if two subtrees are the same, they are instances of the same pattern, but if two subtrees are very similar, they can be considered as two variations of the pattern where each variation can have one or more instances.

Since variations have slightly different syntax, when one variation is replaced as a pattern, the other variation does not get replaced. The user has to manually replace each variation and assign it almost the same description with other variations although most of variations do have the same functionality with respect to the user level of interest. For example, “browseHierarchyList” is a routine that refreshes the hierarchy window in TkSee. “browseHierarchyList”, when called, executes a little differently depending on the content of the hierarchy windows. However, asking the programmers who used it, they say that they call it within their code they only care about refreshing the window; the routines called by it have no importance for them.

To address such cases, we developed the pattern variations feature that automatically replaces all variation subtrees, S1..Sn with description D1..Dn, and saves the sequences and description in an external file. This is similar to batch patterns replacement by descriptions except that the tool will automatically locate different variations and assign different descriptions for each variation. The different descriptions are formed by the root node name (e.g. the browseHierarchyList) or by a user-entered description concatenated with a string and a serial number to indicate the variation number. Thus each variation will be treated exactly as one pattern from now on.

The identification of what makes a good candidate for variation replacements depends on the user's discretion. Certain visual cues, however, are good indications that a pattern has many variations. The visual display created by the fact that nodes belonging to a pattern have special "P" icons and the fact that the colour of this icon changes between successive patterns gives powerful hints about the existence of similarity with other subtrees. For example, a subtree that is made mostly of P icons in changing colours with few non-P icons would suggest that this subtree resemble another pattern with the non-P icons as the differences.

DynaSee offers the following operations on pattern variations:

**Replace variations:** The variations of a subtree will be replaced by a description made out of the name and a string indicating the variation number.

**Pattern info:** gives information about the number of instances and variations of a subtree. A regular pattern would have one variation and many instances. While a pattern that has variations may have one or many instance for each variation.

### *4.4.3 Routine ranking*

A major assumption in the DynaSee design (the use of call tree level and the expand/collapse features) is that, for comprehending traces, not all routines are equally important and relevant to the SE's needs. There is a hierarchy of importance that loosely mirrors the call hierarchy itself where the routines at high level of the call tree are closer to application concepts and those at bottom are implementation concepts.

Observations from interacting with the SEs indicate that it is desirable to look first at a small number of "important" routines that are typically present at high level of the call hierarchy in order to comprehend the general functionality of the trace. In some cases, the SEs need to examine the lower level details where the less important routine shows up.

The selective level display and collapse expand features of DynaSee's call tree (see section 4.4.4) are aimed to satisfy this requirement. However, the call tree was found to be

crowded with routines that have low importance or relevance even at high levels of the call tree. In this feature, our goal is to allow filtering out the undesirable-to-see routines to reduce the overhead in exploring the trace.

#### *4.4.3.1 The notion of utilities*

We conjecture that most of the undesirable-to-see routines have a "utility" role. Typically, a utility routine does not have an application-related functionality, rather it performs some generic operation that is used by other routines in different areas of the program.

Removing such routines would not break the integrity of the trace because the routine trace anyway is not a complete description of the execution, as it does not describe the control flow inside the routines. As such, these routines may be considered as an extension to the programming language and thus be considered as lines of code instead of as units of abstraction. In fact, the line between what is a routine and what is a language feature is not always crisp. A new version of a programming language could provide new functions that match those which have been created by programmers in utility libraries.

Utility routines described above represent the least important routines and thus can be removed first. At the other end of the importance spectrum there is the event handling routines that are the first to execute as a response to an application event. They typically appear at the top of the call tree, parenting large subtrees.

#### *4.4.3.2 Weight computation*

The routine ranking feature is designed to heuristically suggest what routines are needed to see first and what are needed least or not needed at all. This is done by computing for each node in the call tree an importance weight (W). W is computed as a function of proximity to the typical attributes that characterize utility routines (thus scoring low on W) or to event handling routines (thus scoring high on W). This weight W will be the means to automatically allow the SE to accordingly control what is displayed.

We compute for each routine in the trace a weight W between 10-100 where the least important routines would have a value 100. So during trace visualization, the user can choose a value and cause only the routines that have weight below that value to be displayed. The W is thus an approximation of the order in which the user wants to see routines.

```
For each node in N the call tree
        W(N) =  (FanIn / maxFanIn) *(occurrence / maxOccurence) *10
        If distanceFromBottom(N) <  10
                W(N) = 100 – distance(N)*10 + W(N)
        Else
                W(N) =10
         End if
End For
```

**Figure 7: The general algorithm to compute W**

The W value is computed in two phases: first, the weight W is computed as a value between 1 and 10 using a formula that combines the number of occurrence of a routine in the trace and its fan-in – the number of different routines that call the current routine. This formula is an attempt to capture the assumption that the increases in occurrence and fan-in correspond to an increase in the likelihood that the routine is a utility. The relative weight of each of these two factors can be adjusted using a coefficient if any additional investigation showed the usefulness of doing so.

In the second phase, the trace is partitioned into 10 different groups according to distance from the bottom. This captures the leaf layering of the tree: the user can apply ranking on one leaf-layer at a time to successively removes leaf layers. The actual leaf nodes are given W = 100, while their direct parents get a value of 90 and so on until the tenth layer is reached (if ever) where all the rest of the nodes are given W= 10.

This incorporation of distance from bottom in the computation of W addresses the fact that routines at the bottom of the call tree are less important as they tend to be generic in nature. Also, those routines that are far from bottom are important because they caused a large nesting of calls to occur. More importantly, the layering ensures that no node could

have a lower W value than any of its children in case that removing the low W parents may cause the removal of other higher value children.

The user can set a numeric value to represent the utility weight threshold (W) of the nodes of the call tree that he wants to be visible. That is, when the threshold is equal to 100, all nodes are visible, but when the threshold is decreased to, say, 87 all the nodes that have W of more than 87 will be removed.

#### *4.4.4 Call tree*

Traces that track routine invocations are overwhelming for human comprehension if presented as one long linear list of entries. The call tree, as displayed in the trace viewer of DynaSee, organises the trace in a hierarchy that exhibits many features that facilitate the browsing and comprehension of traces. The user can expand and contract particular sub-hierarchies or restrict the entire display to a particular level of depth. We use an expanding/collapsing tree control for the user interface similar to that of Windows Explorer.

Expanding/collapsing capabilities are powerful features for dealing with large volumes of data, especially when data are hierarchical in nature. Call relations form an aggregation hierarchy of functionality, as the called routines perform part of their parents' functionality. If the high-level routine is not relevant, none of its children would be, since calling a routine is equivalent to sub-contracting some of the functionality that the parent is supposed to do. For example, a routine that initialises the GUI named Init_gui, may call init_menu, init_midWindow and so on. If an SE is trying to locate a problem that is not related to the GUI, he won't need to look under Init_gui, whose subtree can remain collapsed.

As we discussed in the routines ranking section, the call level in the tree also mirrors a hierarchy of relevance. The high level routines (minimum nesting level) in the call tree reflect more high-level and application domain concepts, while low level calls are more

generic and implementation oriented. The ability to look at only the high level routines and drill down into the lower level only when necessary is particularly useful in establishing the mapping with the application domain and greatly mirrors the actual code exploration behaviour of SEs as our task view suggests.

**Figure 8: DynaSee call tree, the P nodes belong to patterns while the N node do not. Note the numbers in brackets: they represents compressed repetitions**

#### *4.4.4.1 Call tree operations*

Additional operations for tree visualization are:

**Limit to this level:** Select a node and the entire tree will be collapsed, so only the level of the selected node, and higher levels, are shown

**Level 1,2,3,4,5:** Collapse the tree to a particular level

**Expand all**: fully expand the tree

**Find same subtree :** Find a similar subtree to the subtree whose root is the selected node; when found, the new root will be made visible and selected.

**Find same node:** Find a similar node to the selected one, when found; the new found node will be made visible and selected.

**Find string:** Find a user-entered string within the text of the nodes of the call tree.

**Next string:** Repeat the previous string search (continue)

**Save:** Save the displayed tree as a file to be loaded later in the same layout as saved

#### *4.4.4.2 Call tree visual pattern*

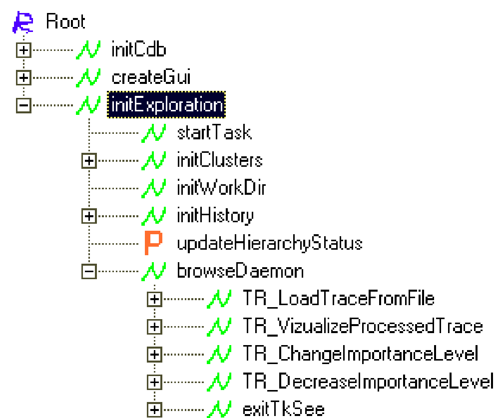


**Figure 9: Call tree collapsed to form a visual pattern**

We experimented with the traces of several systems (see Chapter 5) and concluded that the call trees are far from being random; rather they share common properties that can be exploited to facilitate their comprehension. One of these properties is about the nature of program execution that leads to visual patterns as shown in Figure 9. The figure shows a collapsed call tree of a trace generated from the trace of the TkSee target system. Most interactive program execution can be separated in three parts: the initialization routines,

the event catching loop routines, and the event handling routines. In Figure 9, the "browseDaemon", as its name suggests, is the routine that catches users' events. All routines before it are for application initialization (note that most of them begin with "init"). Each of the routines after it is an event handling routine that corresponds to user initiated events (clicks of buttons or selecting of menu items). Another observation about these routines is that their names closely resemble the labels of the buttons or menu options that initiated them. For example, from the menu "trace", the option "change importance level" has a corresponding event handling routine named "TR_ChangeImportanceLevel". This pattern, and also the fact that all traces have the same initialization, event-catching loop, and exit code (exitTksee), make locating the trace corresponding to an event easier.

#### *4.4.5 Bookmarks*

DynaSee supports a special trace entry called a *bookmark.* A bookmark plays the role of trace annotation that can be inserted inside the trace to indicate an application-domain-visible event. Examples of bookmarks include user-entered descriptions, user interface messages, and logged events. These bookmarks act as cross-reference points, a way to tell where an application event corresponds in the trace. For example, if an error message is inserted as a bookmark in the trace, an SE will identify this node and thus identify what part of the trace occurs before and leading to the error message and what comes after. Note that the bookmark insertion mechanism inside the trace is application dependent; typically it would be part of the instrumentation. SEs using the trace can also add bookmarks manually as they attempt to understand it.

In addition to supporting the display of bookmarks as visually distinguished nodes, DynaSee offers the "view bookmarks only" operation that collapses all nodes enough to show all bookmarks nodes. This will offer a bookmark view of the call tree where the user can easily locate a certain bookmark and expand the tree guided by the bookmarks.

#### *4.4.6 Code window and TkSee*

To achieve traceability between the dynamic and static domain, DynaSee uses a code window such that when the user double clicks on a routine node, the source code of that routine will be displayed in this window (the right window in
Figure 1).

An additional feature of the code window is that the user can select a routine (node in the call tree) and click on the "Show calling lines" menu option; the code window will show the code of the parent (caller) of the selected routine. In the parent's code, the line(s) that may have caused the call to the selected routine will be highlighted.

Moreover, we are working on closely integrating DynaSee with the static analysis part of TkSee. The integrated product of TkSee and DynaSee will allow the user to select a trace routine then perform static queries such as examining the definition of variables, types and routines. Eventually, the user will have full access to all the other features of TkSee, so he or she can do such things as look at the source code for a routine, study the data it uses or study its history of maintenance.

#### *4.4.7 Summary of features*

The features of DynaSee are attempts to increase the comprehension of the program by making the trace play the role of a comprehensible representation of the dynamic domain in a way that, in turn, facilitates domain mapping. The main functions to achieve this are a) to perform compression of the trace (repetition removal, and pattern detection), b) insertion of explicit links (bookmarks and pattern descriptions) and c) to permit selective views of nodes (call tree and routine ranking).

The call tree with its subtree expand/collapse and selective call levels display permits the user to avoid dealing with a high percentage of non-relevant nodes. Repetition removal was indispensable for the navigation of the call tree as it reduces the size of traces by many folds without reducing its information content.

Patterns, in addition to their role in compressing the trace, facilitate the traceability between the dynamic domain and the application domain. As pattern descriptions mostly correspond to application concepts, assigning a description to a sequence of trace routines corresponds to an explicit mapping from dynamic to application domain.

Bookmarks facilitate the traceability between the application domain and the dynamic one by representing explicit cross-reference points. Also it reduces the need to search for a starting point in the code.

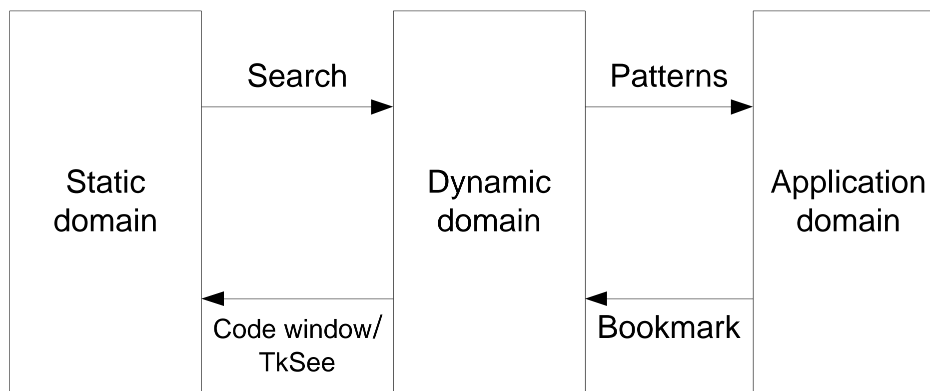


**Figure 10: the support for domain mapping in DynaSee.**

Figure 10 depicts a diagram of how the different features of DynaSee support all directions of mapping between domains. Note that “Search” corresponds to allowing the user to search for a routine name inside the call tree.

Routine ranking reduces significantly the size of the trace and enhances the selective view by removing the noise caused by frequently occurring routines that have little value for comprehension (the utility routines).

Many of these features intersect. Repetition removal is the most certain thing to do first as it is the most necessary and does not interfere negatively with the other features. Depending on the trace, the target system and the nature of the user’s task, different

combinations of features may be used (Repetition removal would be the common dominator).

Patterns are more desirable when the accumulated effect is evident. Assigning descriptions to patterns is a manual task but the accumulation of saved patterns (a pattern here means the description and the corresponding routine sequence) from many users over a period of time would increase their coverage of traces and therefore make the trace a "functional tree" composed of pattern descriptions.

Pattern variations are appropriate when different variations of the same pattern occur frequently. Bookmarks are very useful when the trace gets very long; they partition the trace into smaller search spaces. Routine ranking is particularly useful when there are many "utility" routines that clutter the trace.

# Chapter 5 Evaluation

The first part of this chapter concerns the success of the solution techniques that we developed to provide a usable dynamic domain representation. We independently evaluated each feature by gathering trace files of different systems, and performing data analysis on these files.

The second part of the chapter focuses on the tool as a whole and its success in satisfying its high level goals and requirements. Among other things, we perform observational experiments with the SEs to evaluate their performance and their perceptions.

## 5.1 DynaSee features evaluation

### *5.1.1 Trace test set*

We evaluated DynaSee by examining two test systems, and then for each system we generated several trace files. We applied DynaSee features on these trace files to perform data analysis in order to evaluate the features.

The first test system is TkSee, developed in the KBRE group to which the author belongs. TkSee is a software exploration tool that is written in tcl/tk and consists of about 50 files averaging about 50K lines of code. Three trace files have been generated of small, medium and large size that were called smallLog, medLog, and largeLog respectively. The choice of TkSee is for logistic reasons; its developers are always available so they can answer crucial questions and give insights about the performance of DynaSee.

The second system is the large legacy system (LLS) that we targeted in the case study. For this system we also collected three trace files representing different scenarios. We also classified them as small, medium and large as the size of the trace has a significant effect on most of the presented results. We call these traces MsmallLog, MmedLog, and MlargeLog respectively.

### *5.1.2 Repetition removal*

The first feature to evaluate is the repetition removal feature that removes redundancy created by loops and recursion. Table 1 below shows the results of applying repetition removal on the trace files. The “Initial lines in trace” represents the original number of entries in the trace file. “% of distinct lines repeated” represents the percentage of unique routines that occurred repeated at least once and thus their repetitions were removed. “% distinct routine compressed” is the percentage of distinct trace lines repeated where compression applied. “Count after” is the size of the trace after the repetition removal operation. “% overall compression” is the percentage of compression achieved.

We used the “% distinct lines compressed” criteria to evaluate effectiveness of this feature because trying to use the percentage of overall compression may be misleading. A loop that contains inside it a routine call may iterate depending on external parameter N that can be the number of lines in an input file or other data sources. But since input files are external to the program, such files may contain one or a million lines and thus one or million entries in the trace may be compressed giving totally different percentages of compression. In largeLog, for example, we noted that about half of the lines compressed were dependent on external parameters such as the size of input file or the number of items in a list.

| Trace File | Initial lines in trace | % distinct lines compressed | % distinct routines compressed | Count after compression | % overall compression |
|---|---|---|---|---|---|
| SmallLog | 3100 | 3 | 2 | 495 | 84 |
| MedLog | 9741 | 3 | 1 | 1229 | 87 |
| LargeLog | 12960 | 3 | 1 | 1712 | 87 |
| | | | | | |
| MsmallLog | 214 | 13 | 6 | 186 | 13 |
| MmedLog | 651 | 6 | 2 | 558 | 14 |
| MlargeLog | 2009 | 8 | 4 | 1723 | 14 |

**Table 1: Analysis of compression achievable by compressing repetitions (including recursion)**

As Table 1 shows, traces may be compressed significantly. Removing repetition is found to be indispensable feature especially for large enough N (the external number controlling the number of loop iteration). However, in rare cases, minor differences in repeated call sequences (that is caused by loops) at low levels caused large amounts of redundancy to remain. This happens when the routines called inside a loop differs from one iteration to another. Fortunately, another feature of DynaSee, pattern variations, alleviates this problem significantly as discussed below.

The large differences in compression percentage between TkSee and the LLS test systems are due to the nature of the processing tasks between each system and the size of the samples taken. TkSee mainly operates iteratively on each node of a list, so repetitions are frequent. The LLS is a call processing system that does not have multiple items to iterate on. Moreover, while TkSee represents several complete events and their handling, the LLS trace represents only a fraction of the handling of an event due to the complexity of its events. We could not obtain a larger sample for the LLS due to practical limitations such as the size of its buffer that holds the trace and because processing in it is distributed among different processes.

However, it is important to note that the percentage of distinct lines and routines compressed in the LLS is larger than for TkSee. This suggests that what made the compression percentage that high in TkSee is a large external factor. The LLS did benefit significantly from this feature as up to 13% of its lines were exposed to compression.

### *5.1.3 Patterns*

Patterns are a way to suggest what sequences of routines can form cohesive functional units so that they can be abstracted by a single description, thus achieving some compression as the description node would replace many routine nodes.

In order to evaluate the compression contribution of patterns, we manually identified the patterns that had the properties that make them meaningful and cohesive to be abstracted by an entered description, and then we replaced them by their appropriate descriptions. Table 2 below shows the number of different patterns identified and replaced and the degree of reduction in the number of nodes in the trace caused by the replacements. "Initial lines in trace" is the size of the trace that is outputted from the repetition removal operation. A pattern is the unique sequence that occurs more than once non-contiguously in a trace. A pattern instance is an occurrence of a pattern.

| Trace File | Initial lines in trace | Total # of patterns found | Total # of instances replaced | Lines after pattern replacement | Average # of instance per pattern | Average length of patterns | % compression |
|---|---|---|---|---|---|---|---|
| SmallLog | 495 | 6 | 32 | 409 | 5 | 4.6 | 17 |
| MedLog | 1229 | 9 | 107 | 921 | 12 | 7.5 | 25 |
| LargeLog | 1712 | 17 | 319 | 1204 | 19 | 6.7 | 30 |
| | | | | | | | |
| MsmallLog | 186 | 2 | 13 | 174 | 7 | 3.5 | 6 |
| MmedLog | 558 | 5 | 13 | 496 | 3 | 7.5 | 11 |
| MlargeLog | 1723 | 20 | 80 | 1316 | 4 | 7.5 | 24 |

**Table 2: Effects of detecting patterns and compressing traces based on these patterns.**

Note that the percent of compression in both systems increases with the size of the system. This can be explained by the fact that the larger the trace, the more likely that more instances of a pattern will be detected. Remember that a sequence has to occur more than once in a trace to be considered to be a pattern.

#### *5.1.3.1 The accumulation factor*

Given that pattern replacement is a manual and exhaustive task, one of the promises of this feature is that patterns will recur across traces so that when saving patterns and

descriptions in a permanent pattern library, these libraries will cover a significant percentage in any new trace.

| Trace file | # of accumulated patterns | # of accumulated instances | % of accumulated patterns | % of accumulated instances |
|---|---|---|---|---|
| SmallLog | 0 | 0 | | |
| MedLog | 6 | 92 | 67 | 86 |
| LargeLog | 8 | 133 | 47 | 42 |
| | | | | |
| MsmallLog | 0 | 0 | | |
| MmedLog | 1 | 3 | 20 | 23 |
| MlargeLog | 4 | 24 | 20 | 30 |

**Table 3: Accumulation factor of patterns where accumulation is the number of pattern of instances identified in the previous smaller trace that matched in the current one**

Table 3 helps in analyzing this pattern accumulation factor. As the table shows, accumulation can be useful: a significant percentage of instances and patterns detected in a trace are matched from a previous pattern library of a smaller trace. In Table 4, we reversed the order of accumulation, instead of beginning with small traces, we began with large traces to see of how many matched patterns in a small trace can be found in a larger one.

| Trace file | Pattern library of: | # of patterns matched | # instances matched |
|---|---|---|---|
| SmallLog | largeLog | 12 | 42 |
| MedLog | largeLog | 11 | 46 |

**Table 4: Accumulation beginning with pattern library of the larger trace**

In SmallLog 12 patterns were matched as compared to 6 patterns when no pattern library was applied. Applying an accumulated pattern library may identify many patterns that

were not detected in the trace itself without applying the library given that they have only one instance (sequence of routines).

The accumulation applied on the LLS system did yield little or no accumulation factor. That is in line with our explanation that since each trace belongs to a different functional part of the system, commonality is minimal and little can be accumulated. Pattern library accumulation is expected to be related to the size of the system since patterns have to cover most of the areas of the system in order for accumulation to be significant.

The tables above should not give the impression that the only role of patterns is to compress traces. Another aspect of the patterns, that is difficult to evaluate by data analysis, is the degree of contribution of patterns in enhancing comprehension, not by compression, but from incorporating more knowledge through assigning meaningful descriptions to parts of the trace. Moreover, descriptions can also enhance comprehension when thought of as explicit links between the dynamic and application domains. We will evaluate these hypotheses later in the chapter.

### *5.1.4 Pattern Variations*

In Table 5 below we analyze the effect of a replacement of one pattern's variations for each trace. The chosen pattern variation is chosen to be the worst-case pattern variation in term of created redundancy and thus will be a best case for compression when replaced. Note that "#of variations" is the number of different variations of the worst-case pattern and "total # instances" is the number of instances across all variations for this pattern. "Root node" is the name of the root of the subtree of the pattern. "% of overall compression "is the percentage of reduction in the trace from the replacements of all instances of the pattern variations.

| Trace file | Initial lines in trace | # of variations | Total # of instances | Lines after variation replacement | % overall compression | Root Node |
|---|---|---|---|---|---|---|
| SmallLog | 495 | 5 | 5 | 277 | 44 | browseHierarchyList |
| MedLog | 1229 | 10 | 22 | 498 | 59 | browseHierarchyList |
| LargeLog | 1712 | 14 | 25 | 897 | 48 | browseHierarchyList |
| | | | | | | |
| MsmallLog | 186 | 6 | 8 | 176 | 5 | translate_swid_to_ss4_sw |
| MmedLog | 558 | 4 | 8 | 455 | 18 | cp_msgmon_dispatch_with |
| MlargeLog | 1723 | 8 | 21 | 1478 | 14 | cp_msgmon_dispatch_with |

**Table 5: The effect of one pattern variation replacement in each system, the pattern variation is chosen to be the most pervasive**

Clearly from Table 5 the pattern variation whose root node is "browseHierarchyList" in the TkSee traces was a remarkable worst-case scenario as it accounted for about 50% of the displayed trace. It was surprising for us and for the SE who work on this system to know how pervasive this subtree was.

| Trace file | Initial lines in trace | # of pattern variations | Total instances | # of line removed | % overall compression |
|---|---|---|---|---|---|
| LargeLog | 1712 | 25 | 42 | 983 | 57 |
| | | | | | |
| MmedLog | 558 | 11 | 24 | 156 | 28 |
| MlargeLog | 1723 | 13 | 21 | 340 | 20 |

**Table 6: The effect of the 3 most pervasive variations**

In Table 6, we further show how pervasive pattern variations are. For each of the traces presented, we show the result of removing the three most pervasive (worst case) pattern variations in that trace. Our analysis, however, did not show that it is always possible to

find frequent pervasive pattern variations such that their replacements would cause a significant compression rate. That is why we omitted some of the smaller traces in Table 6 since they did not have 3 pervasive pattern variations.

On the other hand, many pattern variations can be found to account for a high percentage of the trace. In addition to the “browseHierarchyList” example presented above, for a trace of the LLS (not presented in the tables) that has a count of 1316 entries, performing 9 pattern variations reduced its size to 751, a reduction of about 40%.

Obviously the pattern variations are a very effective compression technique beyond our initial expectations. Replacing pattern variations should not incur any loss of information since any replaced variation whose details may become needed can be simply restored (reversing the replacement).

Our assumption about variations is that their syntactical differences are irrelevant to the interest of typical SEs. The root routine is only called to perform a simple service regardless of what routines (children) are called to perform this service. Our consultations with the software experts on how much this assumption holds were encouraging: all variations mentioned in our data analysis were considered by the SEs to be compatible with our assumption.

The remarkable success of this feature may stand alone as a contribution in a separate thread of research that relates to compressing dynamic data. Particularly, we note the conceptual similarity between these variations and the concept of utility routines: they both perform generic services making them both non critical to SM tasks. However, given the theme of this thesis we leave further development and capitalizing on this technique to future research.

### 5.1.5 *Routine ranking*

In order to evaluate the routine ranking feature and its ability to elide utility routines, we began by decreasing the threshold level of importance (W) by one point at a time and in the same time tracking the routines that are removed after the call tree was refreshed to reflect the new threshold.

Table 7 shows the routines removed at each new threshold value as applied on largeLog. Note that for largeLog, removing only 5 routines reduced the trace by 31%. More importantly, the experts on the subject system ranked all of these 5 routines as low level and generic; their removal did not reduce the information value of the trace but significantly eased the browsing of the call tree.

| Routines removed | Threshold value | # of nodes removed | % of nodes removed |
|---|---|---|---|
| updateHierarchyStatus | 98 | 165 | 10 |
| startTask | 95 | 107 | 6 |
| UpdateHierarchyStatusRight, getInfoFromData | 91 | 186 | 11 |
| getTypeFromData | 90 | 82 | 5 |
| | **Total** | 531 | 31 |
| | **Average** | 135 | 8 |

**Table 7: the effect on routine removal after ranking applied on largeLog**

In Table 7, however, we only reduced the threshold to 90. Reducing it below 90 means that all leaf nodes will be removed regardless of their importance ranking. When we did that, a large number of routines were removed, not all of them were ranked as utilities by the experts.

| Routines removed | Threshold value | # of nodes removed | % of nodes removed |
|---|---|---|---|
| Get_ptr_ss4_record | 96 | 115 | 7 |
| translate_swid_to_ss4_sw<br>check_cos | 93 | 93 | 5 |
| get_ptr_brdcst_record | 91 | 37 | 2 |
| dpnss_party,<br>check_call_setup_state,<br>user_class_of_service,<br>get_ptr_acd2_ss4_to_ager | 90 | 63 | 4 |
| | Total | 308 | 18 |
| | Average | 77 | 4 |

**Table 8: the effect on routine removal after ranking applied on MlargeLog**

Table 8 shows the effect of the gradual decrease of the threshold on a LLS trace file. Table 9 gives the total compression achieved when the W threshold is set to 90 on all traces.

| Trace File | Initial lines in trace | Count with threshold 90 | Number of distinct routine removed | Compression % |
|---|---|---|---|---|
| SmallLog | 495 | 144 | 6 | 29 |
| MedLog | 1229 | 397 | 6 | 32 |
| LargeLog | 1712 | 531 | 5 | 31 |
| | | | | |
| MsmallLog | 186 | 40 | 5 | 22 |
| MmedLog | 558 | 80 | 5 | 14 |
| MlargeLog | 1723 | 301 | 8 | 17 |

**Table 9: the effect on routine ranking with threshold equal 90 on all the trace files**

Removing routines beyond the 90 threshold was found however to be very useful in showing a skeleton of the entire trace. That is, it clearly showed the major branches of execution/functionality as diagonal lines (see Figure 11) giving a good quick overview of execution events. The diagonal shape of the branch of the call tree can be explained by the fact that when leaves were removed, only the routines falling in a deeply nested series of calls remained. This coincides with our assumptions that important routine are the root of large subtrees which means that they will be the last to be removed in a removal order based mainly on distance from the bottom.

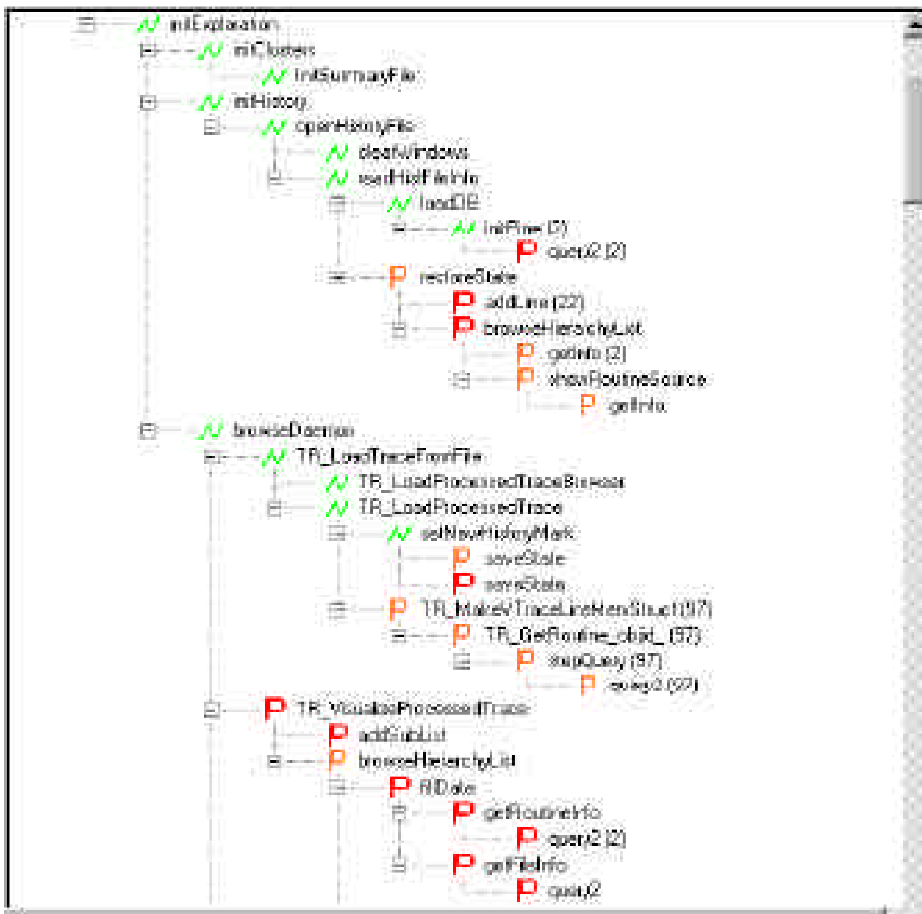


**Figure 11: the diagonal shape of the call tree branches after W is set to 71**

### *5.1.6 Combining features*

So far in this chapter, we evaluated each feature independently (except for repetition removal that was applied before all other features). The combination of features and the order in which they are combined should affect each feature's results in addition to the overall results (e.g. the total compression percentage). For example, pattern variations will remove many of the utility routines thus reducing the achievements of routine ranking. On the other hand, removing utility routines will improve pattern matching as some of the removed routines may have caused some subtrees not to match.

Table 10 shows one possible permutation of applying features in succession from left to right. Each column shows the remaining number of lines after applying the feature mentioned in the column heading.

| Trace File | Initial Count | Repetition Removal | Patterns replacement | Routine ranking at 90 | Pattern variation 1 | Pattern variation 3 | Count of D node |
|---|---|---|---|---|---|---|---|
| TkSee largeLog | 12960 | 1712 | 1204 | 903 | 576 | 464 | 142 |
| MlargeLog | 2009 | 1723 | 1316 | 1033 | 937 | 734 | 73 |

**Table 10: count after applying the operation in succession**

"Pattern variation 1" is the replacement of one pattern variations per trace file while "Pattern variation 3" is the replacement of three pattern variations. The "count of D nodes" is the number of nodes that are either pattern description nodes that are entered by the users during patterns replacement, or generated descriptions by the pattern variations replacement feature.

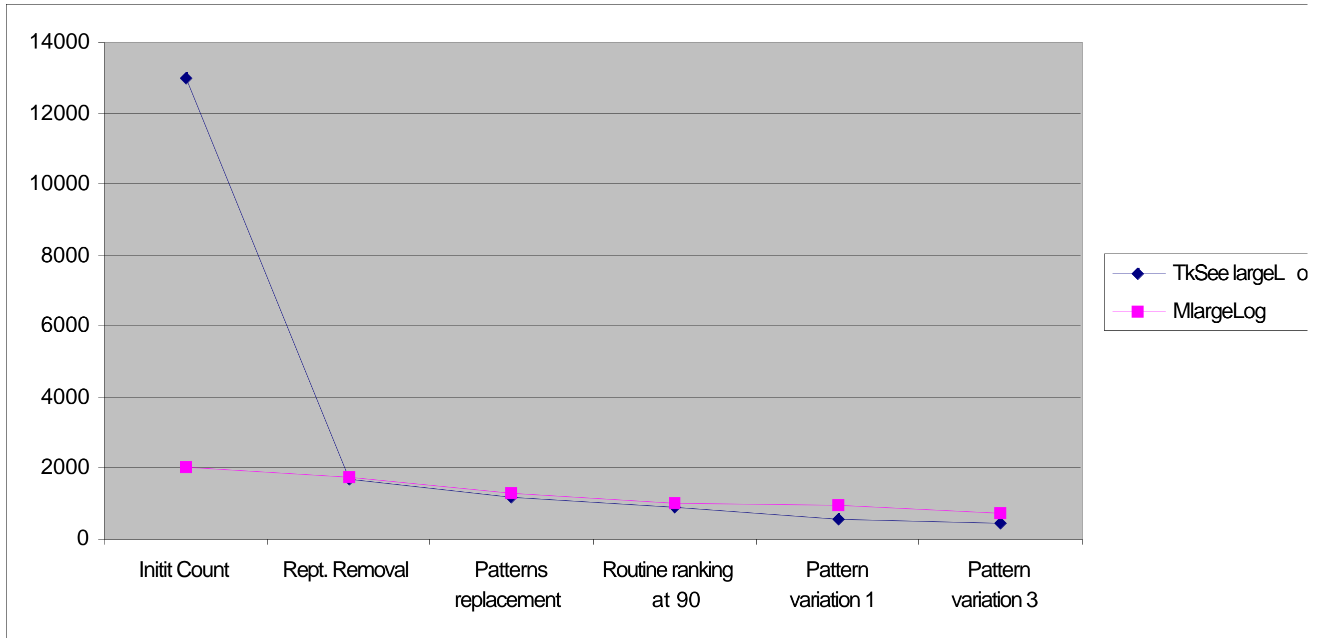


**Figure 12: Graph showing the compression rate after applying each feature**

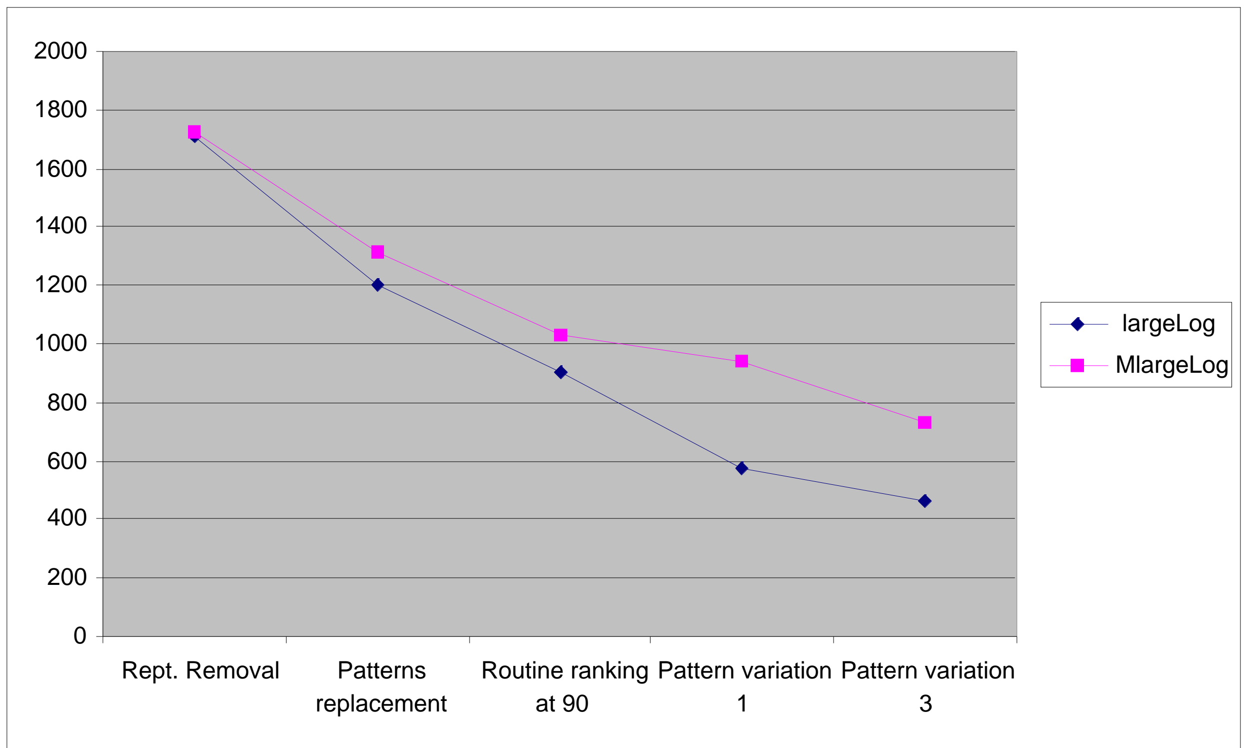


**Figure 13: The same graph without repetition removal that biases the graph**

| Trace File | Initial Count | Repetition Removal | Pattern | Routine ranking at 90 | Pattern variation 1 | Pattern variation 3 | Combined reduction | % of D Node |
|---|---|---|---|---|---|---|---|---|
| LargeLog | 12960 | 87 | 30 | 25 | 36 | 19 | 98.9 | 31 |
| MlargeLog | 2009 | 14 | 24 | 22 | 9 | 22 | 63.5 | 10 |

**Table 11: Reduction percentage of applying the operation in succession**

Note that 10% of the overall nodes are D nodes that make the call tree heavily populated with nodes that describe known high-level concepts and thus make the tree close to a functional hierarchy [Chen 95].

Table 11 shows a counterpart for Table 10. Figure 12 shows the same data in graphic form, while Figure 13 ignores the repetition removal to better depict the contribution of the other features.

### *5.1.7 Conclusion*

The concern of this first part of the evaluation has been to determine the success of the solution techniques that we provided; in other words, have we been able to develop a usable dynamic domain representation. The goal is to prove the feasibility of reducing traces corresponding to realistic situations into a manageable form that can serve SM purposes. Obviously the various features reduced the size dramatically as Table 11 shows. In addition, a significant percentage of the remaining nodes were high-level abstractions (D nodes). This figure of 98% shows that, even though it may be exceptional, such dramatic cases do exist. When these are encountered, without our techniques, SEs would develop a very bad impression about the usability of traces or call trees. In our opinion, what is negative for a user is not the size of the trace but its degree of redundancy.

In general, the depth of the data analysis and its statistical validity is not a concern for us as we are not presenting our features as “the solution” but rather as a proof of what can be done for call trees and how successful it can be. In fact, as we mentioned in several places, the possibility for improvements and for the development of additional techniques were

abundant, but the focus of this work is not to develop techniques, we only want the techniques to play a role in the overall approach of the thesis.

## 5.2 Revisiting requirements

The second part of the chapter focuses on the tool as a whole with regard to its intended goals and requirements. In this section, we begin by revisiting the requirements that we generated at the end of the case study to verify how much we were successful in realizing them. Each requirement is expressed in terms of a question that we try to answer.

### *5.2.1 Does DynaSee create new difficulties or overhead for interpreting the dynamic view?*

To answer this question, a good approach is to compare DynaSee against tools for exploring code statically, since DynaSee is supposed to replace the need to follow the call tree (control flow in general) in the static code. The first source of additional overhead is the redundancies created by the dynamic program interactions. Most of DynaSee features attempt to reduce this overhead to an acceptable degree (the call tree becomes useable). The data analysis and experiments that we provided in the previous section show that, in DynaSee, traces are compressed significantly, so that most redundancy is removed.

The second source of additional overhead would be that a static explorer would not follow the whole depth of the call relations and only investigate the next level of call when needed. The expand/collapse features ensure that no additional level has to be seen if not wanted. The routine ranking goes a further step to minimize the number routines that have to be seen within an expanded subtree by removing the undesired utility routines.

In other words, DynaSee brings exploration close to the state where the SE does not need to look at any more software routines while exploring code using DynaSee than while performing static code exploration. Even if it is not very close, nevertheless, the gain from a dynamic view of code compared to having only a static view far outweighs the additional overhead.

### *5.2.2 Does DynaSee Reduce the CL in search?*

As discussed earlier, search has been identified as cognitively taxing for the comprehension of code; therefore reducing the need for search or increasing its efficiency was a requirement. Comprehending a slice using the call tree dramatically reduces the need to search for the "next delocalized pieces" of code in the control flow as there is no need to get such next pieces given that in the tree, code fragments of the slice are made contiguous and are no more than a mouse click away.

Even when the SE is looking for other pieces that do not immediately follow in the flow sequence, he or she need not search the whole system but only need consider the trace that accurately represents a slice as it was really executed. Exploring and scanning the call tree and its directly related code becomes feasible, whereas if one needs to deal with the whole system, only searching is feasible.

Moreover, the search for a starting point in code is also alleviated by the call tree as the tree corresponds to the control flow, something that a start point is trying to catch. Moreover, the bookmark feature reduces the need for a starting point as bookmarks represent points that are very close to the most relevant code.

### *5.2.3 Does DynaSee reduce the negative CL effects caused by the issues presented in the difficulties?*

As for delocalization, as argued in the previous section, the main delocalization in slice comprehension and the gap caused by search are gone all together when using a call tree.

Deep nesting is significantly alleviated. As we mentioned, the problem of the nesting at its origin is that the atomic meaningful unit that needs to be formed from the nested relations is often larger than WM capacity. DynaSee reduces this problem by reducing the need for memory resources in general. More specifically, making routine names (the components

of the atomic unit) visually available means that they no longer need to be retained in the WM but instead are available thru the perceptual system.

Uncertainty about the actual flow of execution within code is removed because the call tree only displays the actual execution path thus giving total certainty.

In DynaSee, meaningful encoding is facilitated in particular by using pattern descriptions. A pattern description is derived from existing knowledge, so by assigning the description to groups of routines, the relation between the new and the existing information is made explicit.

### *5.2.4 Does DynaSee support domain traceability?*

Under our theoretical framework, the gist of comprehension is a mapping between domains. The call tree is by itself a mapping facilitator as a representation of the dynamic domain. Moreover, the full inter-domain mapping was directly supported by several DynaSee features as depicted in Figure 10 in chapter 4.

## 5.3 Holistic evaluation

As a continuation of the evaluation of DynaSee with regards to its high level goals, we try in this section to evaluate the usefulness of the tool as a whole. In the holistic evaluation approach, the system is tested without decomposing it into component parts. In this section, we are concerned with the tool as a whole – how it scores on its main high-level objectives regarding facilitating slicing and slice comprehension.

As a relaxed form of observational experiments for evaluating the high level objectives, we created artificial tasks that simulate realistic requirements during SM based on our task view of SM. We asked a set of SEs to use DynaSee to perform the tasks. The pseudo-experiments are observational, since controlled experiments are hard to perform in this domain [Storey 97]. Results will also be observational and analytic, focusing on

perception, and quantitative where possible. Later in this section we discuss the problem associated with methods of evaluation and the nature of the results.

### *5.3.1 Task 1:*

The goal of Task 1 is to test the comprehension of a trace: is the trace represented as a DynaSee call tree comprehensible enough to be a description of the dynamic domain that can be interpreted by humans? Specifically, we want to test if the trace can allow for the synchronization between trace routines (dynamic domain) and program behaviour (application domain).

#### *5.3.1.1 Setting*

Two to three traces (call trees) were presented to the participants (five SEs who have worked on the target system to which the traces belong). The participants did not know the scenarios and events that created the traces but were familiar with the application in general, and other high level concepts.

The participants were given a short tutorial and demonstration about DynaSee and its features. Then they were asked to tell what events are occurring in the application domain (events that are visible to the user) from browsing the trace using DynaSee; they were allowed to use all of its features. Examples of application domain events include, events triggered by the user (e.g. button presses), the display of output to the user.

#### *5.3.1.2 Results*

The participants were able to identify from the trace on average of 4 out of the 5 application level events. This was accomplished mostly without looking at code or expanding beyond the tree's higher levels (level 2-4). While browsing, participants frequently used the tree navigation features such as limiting to a call level or collapsing and expanding the tree. They did not, however, volunteer to use the routine ranking but they welcomed it when it was suggested to them.

#### *5.3.1.3 Observations*

Routine names at the top 4 levels of the tree were informative enough; i.e. their names match or connect well to domain concepts. The limited cases where routine names did not connect, and the participants needed to drill down to lower call levels or to look at code statements to find the connection (mapping) included situations where:

1. Routines implemented general functionality and thus had vague names.
2. There were non-common areas of the system where the code was unfamiliar.
3. Program behaviour was unexpected by the participants or contradicted their hypothesis of how it should be.
4. Program behaviour became particularly relevant to the problem investigated and thus lower level details became more relevant.

### *5.3.2 Task 2:*

The goal of Task 2 is to test the ability of the call tree in helping the SE to locate the relevant code (the slice) for a certain functionality. In other words, can the call tree represent a sufficient slice for maintenance tasks? The role that the call tree should play and that we are testing is to facilitate mapping from the application domain to the dynamic domain and consequently the static domain. This is where the problem of finding a starting point occurs.

#### *5.3.2.1 Setting*

Two to three traces (call trees) were presented to the participants. The general functionality that the call tree represents was described to them. They were asked to find in the call tree the sequence of routines where a certain piece of functionality (program event) is implemented. They were asked questions such as: identify the call tree nodes where the syntax of the input file is checked, pattern painting occurs, or the ID of the caller is forwarded etc.

*5.3.2.2 Results*

The success in locating the desired functionality depended on its conceptual level. An implementation level functionality was harder to find than a higher-level functionality that is more related to the application concepts.

Higher-level functionality identification had similar success to what is reported for Task 1. There were also the same situations in which users needed to drill down (look beyond level 4 of the call tree). The only differences compared to Task 1 were that exploration was less systematic and more opportunistic.

Implementation level concepts were less easy to find since as users began looking at more specific details, there was more need to look at lower level routines, at routine comments and even at code comments and code lines. At this level of detail, less matching occurs between routine name and functionality concepts (it becomes more difficult to guess that a routine implements a functionality from its naming only).

*5.3.2.3 Observation*

An intriguing observation is that some medium level users (in terms of application expertise) still wanted to search the call tree for routine names despite the fact that the call tree represents a small space to explore. Maybe they are wired to search given their habitual use of this technique. But in some cases search inside the call tree was very efficient particularly when they know, from previous experience, the exact name of a routine that performs a piece of functionality. This was not alarming for us given that we are not targeting experts who usually find search to be a convenient and efficient mechanism given their accumulated knowledge. Those participants used search intuitively whenever they expected that exploring was too lengthy and search might yield faster results. But for a novice, search would be of less help, given that he faces a shortage of cues on which to base the search.

Finally, it seems that the success in the assigned tasks depends predominately on the quality of routine formation and naming. The more a routine name accurately describes its

functionality faithfully and unequivocally, the more the call tree satisfies its objectives as a means to describe the relevant code and the dynamic domain. Routines at their origin are means to abstract the functionality of their statements in order to have functional cohesive decomposition of the program. If this role is maintained and respected, then the call tree would be an excellent and reliable hierarchy of abstraction to manage the complexity of programs.

## 5.4 Semi-holistic evaluation

In this section, we want to evaluate how the utilization of certain features of DynaSee would improve its overall usefulness. We repeated Task 2 in two different variations that we will call Task 3 and Task 4 where in each variation we explicitly asked the participants to utilize some of DynaSee's specific features, and then observed the contribution of each feature.

### *5.4.1 Task 3*

The goal of Task 3 is to evaluate the role and usefulness of bookmarks in reducing some of the inefficiency of Task 2. That is, given that bookmarks are designed to be aids for mapping from application domain to dynamic domain, we want to test how much they can help in locating the call tree nodes that correspond to a piece of functionality. Note that this experiment was only executed on the TkSee system, as it wasn't possible to instrument the LLS.

#### *5.4.1.1 Setting*

As we mentioned in Chapter 4, bookmarks are produced by the same mechanism (instrumentation) that produces the trace at the application level. DynaSee only reads traces and does not produce traces or bookmarks. In order to generate bookmarks inside the trace, we modified the TkSee subject system so that it responds to pressing a hot key (F2) by opening a dialog box where the user can enter a description. This description will be inserted inside the trace as a bookmark entry (with a special padding to distinguish it

from routine trace entries). When the trace is displayed as a call tree, the bookmark nodes will have their special icons that are easily distinguishable from other routine nodes.

We asked the participants explicitly to use the bookmarks after we had demonstrated how it works. The participants inserted bookmarks before each interactive application event (when the application is waiting for a user input) that preceded the functionality in question (functionality that was hard to locate in Task 2). We then asked the participants to locate the events (functionality) that they failed to locate in Task 2.

#### *5.4.1.2 Results*

Using bookmarks with Task 2 greatly facilitate locating the code for an event. The user can collapse the tree to show only the bookmarks, and then locate a bookmark that he inserted and only investigate the few subtrees after that bookmarks.

#### *5.4.1.3 Observation*

Bookmarks are powerful in mapping between application and dynamic domain but only to a certain level of granularity. In our case, bookmarks can only cross-reference between interactive events. That is, a bookmark can be inserted only when the system is a waiting to accept a new event and not while it is processing the event handling. This granularity is acceptable in our target system (TkSee) but may not be in other larger and less interactive systems. Yet, other mechanisms for inserting bookmarks can be used by more selective instrumentation for the points in code that have application-domain significance such as on errors or exception handling points. This way, a bookmark can be inserted in the trace to identify the relative position of such an event within the trace.

### *5.4.2 Task 4*

The goal of Task 4 is to evaluate the usefulness of patterns in increasing the comprehension of traces. We want to evaluate if patterns reduce the effort required to explore the call tree and consequently facilitate slice comprehension.

#### <u>*5.4.2.1 Setting*</u>

With the aid of developers for each system we used, we began by replacing all meaningful patterns by descriptions entered by the developers. After replacing several patterns with descriptions, we repeated Task 1 focusing on the areas that caused difficulties of comprehension.

#### <u>*5.4.2.2 Results*</u>

With pattern descriptions, user performance improved in many occasions. Many generic routines that confused the participants during Task 1 were replaced with more meaningful descriptions that describe the exact functionality of the specific occurrence of that routine within its corresponding subtree. Note that a generic routine can be the root of different subtrees and thus perform different functionality.

#### <u>*5.4.2.3 Observations*</u>

Contrary to our initial assumptions, pattern descriptions at lower level of the call tree were more valuable than those at higher level for two reasons. First, as we mentioned before, there is less match between routine names and functionality concepts at these levels since they tend to be more generic, and therefore it is harder to identify their exact role. A user embedded description will facilitate this identification. Second, lower level patterns occur more frequently; one pattern replacement will replace many instances. This means more compression and more cases were the benefit of the patterns would occur.

### *5.4.3 Conclusion*

The value of the above observational experiments does not come from "confirming" or "proving the success" of DynaSee since such confirmation needs a more formal setting. The goal is rather to "discover" what is working and what is not, consequently what should be the thing that should be focused on in future iterations, given the learned lessons. In summary, the experimental results were consistent with our assumptions about the usefulness of the tool; however, they stimulated many opportunities for the next version of DynaSee.

## 5.5 User perception

One of the fundamental assumptions in this thesis is that, excluding environmental factors, what matters for adoption and success in general is the user's perception about the usefulness of the tool. An important question to answer would then be: is the user's perception about the tool's usefulness positive enough so that he or she would be likely to adopt it?

Given our choice of the context to study, we had the unique opportunity to have our users to be at the same time software designers. In fact, some of them may be more experienced than us in software development, making debriefing a viable method to capture their perceptions. In addition to debriefing about their perception of usefulness, we also asked the SEs about their impressions, suggestions and what they liked or disliked about DynaSee.

### *5.5.1 Debriefing results*

Some of the SE's that we talked to were well aware of the issues of tool adoption. Their recommendations gave us useful evaluation knowledge.

An entry-level user (4 weeks in the company) said that DynaSee is exactly what he wants to comprehend the system that he is trying to maintain. He said that he asked his manager for any diagram that shows the control flow of the system. For others, DynaSee was quickly considered as a useful "debugger" that can give a panoramic view of the execution sequence. Actual debuggers (like gdb) are line based and are very time consuming and disorienting as they cause frequent jumps among delocalized parts of the code. One user said that it will be of great help to him if we can make DynaSee to allow the online construction of the call tree by stepping through routines and then allowing the user to branch to line-based debugging when he gets closer to the problem. He noted that actually he does exactly that with a line debugger as he steps over routines (skips the step-wise execution of routine statements) at higher call levels and drill down to lower-level routines, then to statements. His comments were a striking observation as they imply that

programmers, including the author, do construct implicit call trees much more frequently and in more instances than we originally thought. This will be clear when we realize that stepwise debugging is actually equivalent to an exploration of a call tree where a "step into" is equivalent to expanding a node and "step over" is the collapsing of a branch. Of course the difference is that in stepping one can only see the current node and only move forward, which explain why the word "panoramic" has been used by a software engineer to describe DynaSee.

Other comments suggested that DynaSee opens a window on an otherwise invisible knowledge domain. The SEs reported signs of enthusiasm and surprise at how the call tree actually looked like, even for the parts of code that they wrote themselves.

### *5.5.2 Concerns*

Users, however, had concerns that DynaSee is only part of their requirements, especially when it comes to exposing (making explicit) the dynamic domain. The telecommunication company users wanted, at the high level, to have explicit presentation of process level interaction. At a low level, they asked for the ability to follow control flow at the level of individual statements, as well as the ability to track the values of variables.

The telecommunication company users were generally more aware of the importance of the trace. In fact, an internal company study about the role of traces that surveyed the maintainers indicated that most of the surveyed users perceived traces in their raw format to be useful in principle but practically unusable due to their sheer size. Notably, many of the users suggested restricting the trace to the fourth call level (the trace can often go up to 13 call level in the LLS) as a way to reduce its size.

### *5.5.3 Discussion about the methods of evaluation*

Designing a holistic evaluation in the form presented above was not a straightforward decision to take. We were aiming at producing a more empirical and quantitatively based evaluation, but we ended up with an analytic one that we found to be more suitable. We

found that it is important to report on this issue, as it is relevant to a major theme about the proper balancing between empirical certainty and problem solving achievement.

In the first chapter we described our approach as "result oriented", seeking practical achievement rather than scientific certainty. We want to close the loop from the problem to the solution in a realistic setting. This requirement implied the need to have an "economy" of effort: given the limited resources of time and energy of the researcher, what is the best allocation of research effort with respect to the practical goals. The rationale is not to spend too much on one particular task so the research would not "sink" in the task sub-problem and resources are consumed enough to minimize the ability to eventually reach a practical solution. Also needed is an economy of complexity: humans are limited in their ability to digest complexity and amount of information as proven in decision science [Umanath 94]. The goal of this economy is not to complicate the research so that the complexity will mask the simple conceptual gains in solving big problems that often float on the surface.

The main vehicle to manage the complexity and effort is to reduce the formalism and to manage the abstraction to the level that serves the practical goals and avoids any costly details that do not constitute an optimal investment in the particular problem solving effort.

What makes holistic evaluation costly in efforts and complexity is that it tries to mostly capture human factors as opposed to the data analysis that dealt with trace statistics only. In the holistic evaluation we want to know how the humans (users) perceive the usefulness of the tool, what they liked or dislike (of features) and how they are willing to behave (adopt the tool or not). Dealing with human factors in quantitative manner is of course challenging. In the next section we discuss some of the encountered challenges:

#### *5.5.3.1 Questionnaire*

In one attempt we tried to develop a questionnaire for participants to capture the factors that would affect their likelihood of adoption such as the perceived usefulness and ease of

use. As an example of the complications of human factors we noted that politeness or the tendency to avoid upsetting the author made the answers for some of the questionnaire questions to be too positive to be reliable. Another example of the difficulties is when answering the question "would you use DynaSee in your SM work practice (adopt it)?"; the answer was almost always "it depends". The "it depends" covers a large number of interdependent conditions, "if it is totally integrated in my tool set and I get enough situations that DynaSee helps me enough to overcome the additional overhead of collecting traces, learning the tool, then running it etc…". The major shortcoming of questionnaires seems to be that no discrete set of questions could capture all the factors that affect a required answer.

More valuable analytic results were obtained from continuing a free-form conversation on this question with the participants. From the conversations, we understood that adoption would take place if there exists a perception that the benefit of the tool will outweigh its cost. More importantly, the conversation helped identify what are the causes that are increasing the cost of using the tool. Such results have obviously more influence on the design of the tool or any further tool than any statistical results. A direct support for our conclusion comes from Carroll [89], one of the father of ecological approach, who criticized human factors psychology that analyses real tasks because it "resolved this concern into simple and isolated quantitative performance measures (task time and error rate). Such analysis rarely provide any articulate direction in the design of artefacts".

#### *5.5.3.2 Task comparison*

Another possibility that we tried and found to be more expensive than economically feasible is the direct comparison between doing tasks with the tool and without it. This was expensive for the following reasons:

1. **There is a task/artefact relationship** [Carroll 89]: This means that the artefact used (the new tool) may dramatically change the nature of the task in a way that makes the comparison very difficult or invalid (the new tool changes the ecosystem into a new one),

only the goal of the task remains the same. But when the common goal is as high level as program or slice comprehension the comparison becomes even more difficult.

For task 1 (tell what events are occurring in the application domain from looking at the call tree) there is no directly comparable activity. DynaSee here is not enhancing an existing activity, but actually adding a new activity that was unaffordable, although very important for SM in general. The goal, which is to comprehend a slice, is done totally differently without DynaSee. If we really want to compare, the closest we can get to such a task without DynaSee is to look at the trace in its raw format or to use line based debuggers; these are out of the question for large enough execution (on large systems). In the targeted LLS, the trace generation feature was rarely used even though it was available with the system since its inception, for the same reason as noted above.

**2. It is hard to measure familiarity**: User familiarity with the code is crucial in showing the utility of DynaSee over existing methods. For example, as familiarity increases, so does the user's ability to find keywords (cues) that get him what he wants through keyword search. Measuring or predicting familiarity is hard because familiarity is different for each user for different parts of code (especially in LLSs). Even if we can do that, we still cannot do the comparison because when a user performs the first task, his familiarity improves for the code, and repeating the same task on the same code would make the comparison invalid.

Of course, the science of empiricism and statistics may provide solutions to these situations; yet, these solutions require more complex settings, more sophisticated techniques and many more participants – all making the effort and complexity beyond our capability (we only had access to about 10 software engineers working on the LLS for example, and we were only occasionally able to get them to help us). Our approach in this thesis was to sample a limited number of SEs in a limited context and to work deeply with them; involving a statistically significant number of users would be contradicting to our assumption about how big problems are identified and solved.

### *5.5.4 Conclusion*

In general, the use of statistics to capture human factors and behaviour, such as in structured equation models (SEM), is still very controversial and highly challenged [Chinn 95]. We think that we can bring more practical value by staying at the conceptual level especially since the current practice and understanding of the current "tool development" community still ignores many of the fundamental and trivial principles of good development practice that can be as simple as the need to investigate and characterize user problems in their realistic setting [Shneiderman 88].

In this type of research, quantitative methods are not always required, and sometimes should be avoided since they may not constitute the optimal investment of effort and complexity when the focus is not pure science but rather problem solving, as in our approach. In fact, as we will discuss in the last chapter, it is not that clear what is more scientific or what is more valuable a scientific contribution as the definition of what is science is continuously evolving. The recent trends have been of moving beyond empiricism toward conceptual and problem solving issues [Laudan 96].

# Chapter 6 Contribution, Conclusion and Future Work

## 6.1 Executive summary

SM is costly mainly because of program comprehension. Reverse engineering (RE) tools are supposed to help but have not experienced a satisfying level of success, mainly because of their low adoption rate. We think that this is because tool designers do not understand real users' (SEs') tasks and problems. Consequently, users do not perceive usefulness from the tool since without identification of real problems, no useful solutions can be provided. To understand their tasks, we have to use proactive methods of conversation, observation, and introspection. To understand difficulties we have to think in terms of cognitive load (CL) and overloads. We argue that the problems of SM mainly correspond to cognitive overloads. Using some cognitive psychology knowledge, the overloads can be analyzed and the elementary cognitive sources of difficulties can be identified. The overloads can then be targeted by requirements for RE tools that should alleviate them.

We applied this approach on a case study. We sampled a context made from a set of non-expert engineers doing maintenance on an LLS. Our proactive methods showed that the major difficulties are related to slicing – the identification of the code relevant to the maintenance request. The analysis of the cognitive sources of difficulties showed that the use of search in this process with the nature and size of the system incurs WM overload due to a) delocalization of code, b) the deep nesting of call relations causing the meaningful atomic unit to be too large, c) the uncertainty in following the control flow, and d) the low meaningful encoding. We theorize that slicing, and maybe all code comprehension, is a process of mapping and synchronization between the code (static domain) and the program behaviour (application domain) that has to go through the dynamic ordering of executing code (dynamic domain). Most the cognitive difficulties are due to the invisibility and implicit nature of the dynamic domain, thus causing the user to rely on his memory to mentally construct this domain thus overloading his memory. According to the distributed cognitive model, we propose that this load has to be moved

from the head of the user to the tool, and thus the tool should build an explicit representation of the dynamic domain.

DynaSee is an attempt to construct an explicit presentation of the dynamic domain, a non-trivial task with many design decisions and problems to solve. DynaSee reads routine traces and processes them for visualization as a call tree. Several characteristics of routines made them the ideal trace unit for dynamic data including that they parallel the actual granularity of code exploration by SEs that we observed. However, like all dynamic data, they tend to be very large in size and incomprehensible in their raw form. To alleviate this problem, we combined classical solutions such as expanding and collapsing the call tree and removing of redundancy caused by loops, with novel solutions such as pattern abstraction, pattern variations, selective viewing after routine ranking, and other visualization operations. Data we used to evaluate our work showed that our techniques were dramatic not only in reducing the size of the trace but also by presenting the minimally-needed amount of information to the software engineer. Other features like bookmarks and pattern descriptions directly facilitate of domain synchronization.

DynaSee evaluation with the users showed that their perception and experience was very positive. They could perform the tasks that we assigned to them with great ease. However, they demanded that DynaSee become an integrated part of a larger tool or tool set that covers more information needs such as process and line based tracing as well as data flow.

Building on the cognitive findings in our experience, we looked in the literature for examples of how to mesh our findings with more general cognitive principles, and for opportunities where similar finding have been applied in different domains. The goal is to construct a scientifically grounded theoretical framework for the design and evaluation of software tools. For example, we showed how some of the learned theoretical findings could be applied to analyse the problems of object oriented languages.

## 6.2 Contributions

The contributions of the thesis span different areas and different levels of abstraction. At a high level, we contributed to the knowledge about how to perform analysis and research. At a low level, our contributions include such things as different compression techniques for dynamic data.

### *6.2.1 Research paradigm*

We contributed in the development of a relatively novel research paradigm that constitutes an important improvement over the dominant paradigm which stems from a restrictive philosophical view of science – *positivism*. We have taken an "anti-positivist" approach in our research (the ecological approach) criticizing much of SM research as being too positivist (see section 2.4.6 for example).

The current worldview of science is largely dominated by the positivism school of philosophy that is considered passé by many people because it is extreme and too restricting [Laudan 96 p. 3]. Positivism still greatly influences the current practice of science particularly in psychology and other human-factors related domains including some of the SM work.

Positivism acclaims experience as the sole source of human knowledge. The emphasis is on theories and hypotheses and how to test and validate hypotheses empirically; claims that have no empirical backing are treated as meaningless.

To appreciate our contribution at this level, we next give an overview over recent trends and opinions of the major philosophers in the philosophy of science illustrating the shortcomings of positivism [Dyer 01]:

#### *6.2.1.1 Recent trends in the philosophy of science*

Popper argued that the positivist emphasis on verifiability only encourages confirmation of theories rather than genuine discovery. He considered that science should be a process of

finding the best answer to a problem rather than the sombre application of logic, as the logical positivists suggest.

According to Kuhn most scientists are conservative – seeking to apply existing methods and theories to new problems rather than seeking to develop new and better theories. Scientists are locked into a paradigm – a common framework for understanding and tackling problems – that constrains their achievement. Kuhn also sees the aim of science as puzzle-solving and stresses the importance of the role of a paradigm on what and how problems will be solved.

Feyerabend emphasizes that, in principle, "all forms of theories are worthwhile". For Feyerabend, logical positivism was too cautious since science is far from being unified by method and can use any of many methods for different types of problems. Feyerabend disputes that any one set of rules or system can be taken as the only or best universal system of scientific enquiry, arguing that such an approach would inhibit scientific progress, no method should be ruled out if it works.

Laudan [96 p.78] provides a problem-solving view of science. To Laudan the objective of science lies in solving problems: "The aim of science is to secure theories with a high problem-solving effectiveness." He argues that science progresses "just in case successive theories solve more problems than their predecessors".

Laudan maintains that acceptance and rejection are too restrictive to represent the range of cognitive attitude taken by scientists toward theories. He notes that the continuum of attitude between acceptance and rejection can be seen to be a function of the problem solving progress of theories.

Laudan addressed another example of how positivism oversimplified the view of science when he notes that not all theories are of one type; rather, "there is a range of levels of generality of scientific theories, from laws at the one end to broad conceptual frameworks

at the other. Principles of testing, comparison, and evaluation of theories seem to vary significantly from level to level".

In the conclusion, he suggests that there is no fundamental difference in kind between scientific and other forms of intellectual inquiry, "all seek to make sense of the world and of our experience".

#### *6.2.1.2 Conclusion about the research paradigm contribution*

We argued  (along with a camp of scholars), that the work that is solely based on rigorous hypotheses proving (positivism) is limited in its ability to generate practical value for design-based sciences. Scientific research should not be constrained by any philosophical view or paradigm but rather has to be open to any methods that bring problem-solving efficacy as it is the ultimate goal of science. We paralleled this conviction when we focused on not only on effectiveness but also on the efficiency of research effort with respect to problem solving. This was mainly achieved by carrying out a breadth-first approach where a problem is investigated to the "necessary" depth relative to its contribution toward the solution of the higher order problem. In the competing depth-first approach, a researcher would spend most of his time at one level of the investigation (e.g. a single problem) without being able to close all the levels necessary to close the practical problem (the high order one).

In few words, we summarize our contribution at his level as the fostering of a research paradigm that is free from the residue of positivism. In our opinion, any successful application of an approach beyond the influence of this philosophy should help to prove the utility of such an alternative and to illustrate its concepts, thus fostering more adoption of it. Note, however, that this contribution is independent from the value of the results reached, it concerns the methodology regardless of how successfully we performed in this particular work.

### 6.2.2 Interface with psychology

Psychology is a science with over 100 years of accumulated knowledge for understanding and predicting human behaviour and performance; it is a rich and well-established science. It is very natural and rewarding to be able to utilize its principles in cognitively intensive software engineering activities that are tightly related to human cognitive performance e.g. program comprehension. Yet a large gap exists nowadays between psychology as an academic science and software engineering.

In our attempt to involve psychology in our approach we found that this view makes our attempt and perhaps previous attempts of using psychology for practical computer science applications to be a difficult and challenging task, mainly because the knowledge of psychology is not structured in a way that supports applying it on a highly practical domain like computer science. The negative effect of positivism made psychology a science that focuses on isolated facts, which makes it hard to benefit from its knowledge base for software application.

#### 6.2.2.1 The structure if academic psychology

Many scholars complain that academic psychology is still a science of laboratory hypotheses, evidence and mini-theories rather than comprehensive theories. The lack of comprehensive theories implies that someone who attempts to use psychology for practical purposes will come across many knowledge gaps – needed areas that naturally belong to psychology but are not covered by any of its literature.

Card [83 p.11] complains that psychology is overly concerned with hypotheses testing – to find out which one of two ideas is right, creating many difficulties in using its knowledge for practical applications.

Dyer and McGhee [01] note that, “there are many different models of the scientific endeavour apart from those that most psychologists use. The evolution of ideas about how to do science indicates that some psychological research is still stuck in an earlier

framework which has been supplanted by more recent and arguably more challenging and liberating models of how to carry out empirical investigations."

Kintsch [98 p.2], a prominent figure in the text comprehension theories, criticizes the concentration of psychology on narrow empirical results and the lack of comprehensive theories. He argues that during the last 100 years, scientists have mostly left global theorization about human cognition to philosophers. He continues to argue that these years allowed us to acquire solid information, but that this information is limited in practical value. This data, according to Kintsch, needs comprehensive theories because even good data are not totally satisfying if they are not tied together within some theoretical framework.

Carroll [89] who is one of the fathers of the ecological study of programmers, criticizes academic psychology saying that it has favoured the study of narrow and artificial tasks not because they are illuminating or useful in real life, but because they are tractable to study in laboratories. In his opinion, this kind of work ends up instantiating theories so narrow that they can be only considered theories of laboratory puzzle solutions. In commenting about attempts to use psychology for HCI, Carroll [89] argues that, "basic psychology theories and methods are radically unsuited to the scale and complexity of HCI design problems".

#### *6.2.2.2 Current state of interface with psychology*

Yet as our experience shows, there is still a lot of room for making use of psychology in software, provided that appropriate education is provided in both domains. More efforts are needed to bring together these two domains and to find ways to deal with knowledge gaps in this area.

During our literature review and reading, we did not encounter any deep utilization of mainstream psychology in computer science. Only psychological terminology or methodologies were used in a non-formal way, such as in cognitive models of program comprehension [von 95]. The work of Lloyd [99] on diagram clarity was a prominent

exception; he referenced literature that goes back as far as 70 years and applied it to diagram design choices.

#### *6.2.2.3 Our approach for the interface between design and science*

Our approach to interface with psychology is to have psychology to shadow the tool development cycle (i.e. to have it in the background and to be aware of its knowledge) so that to seize any opportunity where a link can be established between tools development and psychology. That is, we began from actual problems and tried to obtain psychological explanations, and from these explanations we tried to influence the proposed solution. In the opposite direction, exposure to psychology and to other domains where psychology affected design, illuminated many areas within the development cycle that were otherwise obscured without this knowledge.

In other words, we do not propose any methodological interface mechanism, but rather an opportunistic approach where a tool designer with the appropriate educational background in psychology and its application would identify areas of intersection between the two domains. That being said, we don't argue against the possibility of a more methodological interface that may emerge after maturity in the relationship between the two domains. Some of the work of the thesis can be considered as a contribution to the educational background that a tool designer should have.

The gain from this active attempt to interface and mesh with psychology is more than the illumination that psychology provides for some design decisions. The gain spans the basic premises of scientific practices. Science grows by the additive accumulation of efforts, where new information uses existing information and builds upon it to push to edge of scientific knowledge. The ability to mesh with psychology allows using its cumulative facts to reduce the effort of new work and to increase the certainty of design decisions and the generalizability of any finding or observation.

To illustrate how the interface was performed and gain achieved give as an example using many concepts from mainstream cognitive psychology about the nature of the human

cognitive system and its limitations, particularly regarding working memory (WM). This increased our confidence in our conjecture about the existence of bottlenecks and limitations of mental activities. Based on this, we suggested a cognitively aware approach that places cognitive considerations as a high priority in cognitively intensive computer applications.

Perhaps the most important generalization from the cognitively aware approach concerns cognitive load (CL). This acts as a cross reference between psychology and design in computer science. We argued that the success of a reverse engineering tool in terms of adoption is more dependent on how successful the tool is in reducing CL as opposed to simply reducing the raw time required to perform tasks. As such, CL becomes the vehicle to interface concepts from psychology and tool development since it has visibility in both domains.

#### *6.2.2.4 Conclusion*

Given the existence of a large gap between basic academic psychology and computer science, our contribution in this area stems from the approach presented to solve the difficult relationship between academic psychology and practical design applications such as the design of computer artefacts (tools to aid humans in their cognitive tasks) and the demonstration of how this interface can work.

Our work can be the foundation for more elaborate collaboration between the two disciplines. The ability to generalize beyond the particular case presented in this work has been shown in the previous chapters. We think that our approach and the theoretical lessons that deal with cognitively intensive applications in general and supporting SM specifically, can lead to solid psychologically grounded theories and methodologies for design and evaluation of computer systems

### *6.2.3 DynaSee*

DynaSee constitutes a considerable contribution as a dynamic analysis tool even when taken in isolation from the approach and other theoretical work of the thesis. In general, the need to have an insight into the dynamic domain is non arguable [Wilde 92, 95].

#### <u>*6.2.3.1 The gap of levels*</u>

Existing dynamic analysis tools are either too low level or too high-level; the call tree of DynaSee fills the gap between the levels. Low level dynamic analysis tools such as line-based debuggers are useful only for finely localized problems. They are exhausting for any major code explorations. On the other hand, high level dynamic analysis tools usually show the dynamic relations between architecture level components [McCrickard 96, Teteishi 94, Pal 97] as such they are of little help for code exploration. As we noted before in the task view, complete high-level system understanding is rarely needed or practical for SM on large legacy systems (LLS) because comprehension is always partial and is closely related to the particular problem being investigated. Moreover, the actual needs of SM are closely related to code, so any tool that works on a level that is distant from code will be of limited usefulness for SM. The call tree of DynaSee affords both the proximity to code and the ability to work on high-level abstraction.

#### <u>*6.2.3.2 Making traces usable*</u>

Despite the fact that the idea of call trees being crucial structure for program comprehension is old and widely accepted, there is a conviction in the software engineering community that, because of their large size, they are not useable [Korel 98, Jerding 97-b]. We did not encounter any work that goes beyond this prejudice into thoroughly trying to alleviate this non-usability.

DynaSee proved that this is not true. With some creativity and effort, a call tree can become highly useable, the size problem of data can be made manageable, and therefore call trees can have an instrumental role in SM. It is this revisiting of what looked obviously needed, yet was abandoned, that is the essence of our contribution.

The compression and visualization techniques used to make call trees useable were not all totally new. The novelty is in the concentrated effort to bring all these techniques together and to apply them in this context. Next, we revisit the DynaSee techniques discussing their novelties:

**Repetition Removal:** This technique has been used in flat trace compression [Larus 93]. Similar algorithms were used to construct directed acyclic graphs [Jerding 97-b]. However it has been maintained that even when applied on call trees, the resulting trees would be still unusable [Korel 98]. While we tend to agree that this is true if that was the *only* technique used to deal with the size of call trees, we showed that additional simple features such as bookmarks reduce the size of the call tree that needs to be investigated to one subtree (succeeding the bookmark). Simple eliding techniques can actually elide everything else. In fact given that the tree is hierarchically explored, the user can expand a node at each level to see what he wants and forget about all the rest.

**Patterns:** Pattern detection is also a well-investigated technique inside and outside trace processing literature. However, in the context of call trees it has a totally new form and meaning, given their hierarchical and semantically rich nature. In other words, instead of looking at the trace at the mere occurrence of trace entries, the call tree entries (routines) manifest a significance in their location and meaning. Abstracting such patterns becomes highly integrated with their meaning, since the user is able to see their functional role within the execution represented by the overall call tree and thus abstracts them into meaningful descriptions.

The pattern variations feature is only feasible because of the intimacy of patterns to their semantic (functional) role so to allow the user to identify what could be a variation not from the syntactic or topological properties of the pattern entries but from their perceived functional role within the overall execution. We did not come across a similar feature in the literature, at least as far as SM and program comprehension are concerned.

**Routine ranking:** This is probably the most novel technique when it comes to our

context. Its distinction comes from its ability to discriminate the relevance of a routine from several of its properties, some of them unrelated to its position in the tree which most of the other techniques depend on. This technique is very promising, yet very research hungry: there is much more room for improvement and future research.

**Call tree:** The call tree user interface is the same one that is used in the Windows file explorer. On the negative side, reusing such a widely known control creates the impression of déjà vu that a user would have from looking at the tool. This may negatively affect his perception about the usefulness of the tool since it implies less novelty and sophistication.

On the positive side, this reuse of packaged sophisticated controls allowed us to benefit from a highly elegant and sophisticated interface that was otherwise very hard to implement. Various expanding /collapsing operations and selective level displays were affordable to satisfy the users needs. The pros of this control do not come only from its sophistication and ease of programming but also from its widespread and frequent use; remember from the previous chapter that frequent use makes a task more automatic thus of lighter cognitive load. It is important to note that this control was not available when others were trying to construct browsing tools, hence limiting their success.

### *6.2.4 A final note on contribution*

We focused on the big picture of the whole cycle of development: the theory, the methodology, the case study, the tool and the evaluation. Taken in isolation none of these look particularly deep.

Literature contains a lot of work on software maintenance task analysis and program comprehension characterization, case studies and on reverse engineering tools but very rarely there is any work that spanned all these separate threads in a highly theoretically grounded and integrated way as we claim we are presenting in this thesis.

## 6.3 Conclusion & Future work

The final conclusion that we draw from our experience is that we are operating in an area that has a severe shortage of multidisciplinary theories. RE tool development is being treated as mere software development without regards to the necessary theoretical foundations (e.g. cognitive and psychological background). More work is needed both at the theory level and at the level of techniques that we provided through DynaSee.

### *6.3.1 About the conceptual issues*

One of the main things that this thesis should prove is that the development of reverse engineering tools is a different breed of software development that has its own priorities. The different priorities of tool development versus general software development comes from:

1. The voluntary nature of tool adoption that make the user perception of the tool a priority
2. The cognitively intensive nature of SM and program comprehension that makes cognitive factors a priority.
3. The difficulties in gathering requirements given that SM is a cognitive and creative process (problem solving) that makes characterizing SM by proactive methods a priority.

All of these factors need theoretical backups, often from different disciplines than computer science. In this thesis we reviewed aspects of several different disciplines retrieving relevant ideas and trying to fit these ideas in one practical approach. Therefore, regardless of how successful or generic is our approach, its value stems from being an early multidisciplinary attempt to bridge the gaps in the theoretical aspects of tools development. Moreover, our approach demonstrates how the multidisciplinary theorization can have a profound influence on actual development effort and decisions.

One thing to pay attention to about theorization is that it is an accumulative and iterative process; we don't provide the truth. Our work is more of hypotheses generation than

validation; many of the ideas are prone to criticism and maybe to invalidation. However, if we want to think beyond positivism the mere endeavouring to suggest new ideas and new relations between existing ones is a valid kind of science provided that sufficient logical support is used. We provided a start so that other researchers may have something to verify, refine, upgrade or build upon.

### *6.3.2 Areas that need more effort*

To be more specific, we point out to the following areas as having a shortage of theorization and therefore are in need of additional research effort:

#### *6.3.2.1 A model and theoretical framework for the development of tool*

Similar to the evolution from viewing programming as the science of producing software to the existing software engineering where programming is only a phase within a larger process that includes analysis, design and testing phases, a larger view of tool development is required where the production of a tool is only a phase within a more general process. In this thesis we provided one such model that involves phases such as task abstraction, difficulty modeling, and cognitive analysis. Yet, as we said, what we provided is only a start for further research to generate more reliable and validated processes.

#### *6.3.2.2 Difficulties in program comprehension*

Although there is much work on cognitive models for program comprehension, there is still a theoretical gap in bridging the findings of these models into design decisions for the development of tools that support program comprehension. This gap is evident in the separation between these models and any available RE tools as far as we know. In fact, the gap is more general as we discussed above between the science of psychology and computer science in general.

The thesis contains many starting points to build upon. The three-domain theory that explains the difficulties by the implicitness of the dynamic domain that we used

extensively is an example of such gap bridging. It transformed the work of many cognitive models such as those of Brooks and Pennington into direct DynaSee features for increasing the mapping between domains. Future work can generalize, validate (or invalidate) and refine our work particularly in the areas described above (section 6.3.2).

### *6.3.3 About the tool*

As we said, this work is not about a tool; rather it is about new paradigms for the development of tools that support cognitively intensive applications. The case study and the tool were only to support these foundations of which generalization and contributions can be deduced. That, however, does not mean that the success of our tools to meet its objectives is not a goal.

#### *6.3.3.1 Adoption*

DynaSee adoption was not tracked and studied. In evaluating the success of DynaSee we stopped at the success of the processing objectives (e.g. compression rate), direct usefulness (e.g. locate an event in the call tree) and at user perception.

A study of the success of adoption requires the ability to control all the variables that affect the large-scale adoption, as it is enough to have one negative variable (that can be unrelated to the tool) for the whole adoption to fail. Controlling all the variables is difficult as a consequence of the difficult relation between academia and industry; the latter has a mindset that can be unsupportive for the research. An academic researcher in an industrial setting is an outsider without the proper authority to push or even to track its adoption in an effective way, as we found out. Another valuable piece of future work would be to continue from here, given that there is positive perception of usefulness, investigating what is needed to obtain full-scale adoption of our ideas in an industrial setting.

For any future researcher, our view is that most impediments for the full adoption of DynaSee are external to it and are of a logistic nature. The most significant impediment is the overhead in utilizing DynaSee particularly in the collection of traces. The nature of the

LLS made collecting the trace particularly difficult as the system runs on a switch and not on a general-purpose computer where DynaSee runs.

Second, there is the overhead for software engineers in learning and incorporating the tool in their daily work practice. Many users did not want to deal with an additional new tool; they say that they are more likely to adopt it if it is an integrated part of their development environment such as the IDE of many recent programming languages

#### *6.3.3.2 Data gathering*

Traces and dynamic data in general are hard to gather and their gathering may cause disturbance and perturbation. Instrumentation is not trivial; it needs language-parsing capabilities for inserting probes that can cause subtle errors if parsing is not impeccable. A better solution should be researched; a good alternative could be using compiler and debuggers technologies that can poll the run time system to get dynamic data such as the call stack.

As we mentioned in the evaluation, users consider the mere use of an additional tool as a significant overhead, so additional overhead in the preparation (e.g. gathering trace) to use the tool is even more negative. There is a need to find ways to reduce the overhead in bringing the dynamic data to DynaSee and to reduce the difficulty of generating these data in the first place.

#### *6.3.3.3 Additional dynamic coverage*

While DynaSee covers a significant gap in the requirements for software maintenance tools, software maintenance is a highly creative and versatile process that has a large number of needs including needs for dynamic information that are not provided by DynaSee. In Chapter 5, we mentioned explicit requirements by the users to add additional features for the dynamic presentation of as process and message level traces. Additional coverage of these dynamic aspects is likely to be fruitful and should not be difficult to integrate with DynaSee.

In fact, some of the features for additional dynamic coverage are currently being investigated (by another group in KBRE) such as the presentation of trace at a statement level. With this feature, a user can click on a routine to see its source code with actually executed statements highlighted. Dynamic slicing algorithms are considered in order to achieve this statement level tracing and additional features provided by dynamic slicing are to be exploited.

#### *6.3.3.4 Patterns*

Discussions with users indicate that the most promising feature of DynaSee that can play a different role than what is envisioned to DynaSee is trace patterns. The usage of patterns for dynamic clustering is being investigated to be added to TkSee in order to augment in order to augment its static clustering capabilities.

Patterns can also have a pedagogic role. We also think that with an accumulation of patterns description, the call tree can be transformed into a dynamically generated functional tree of the program. Such a view will be of significant help to newcomers to get an overview of a particular system.

#### *6.3.3.5 TkSee*

Finally, DynaSee is being ported to TkSee so it becomes an additional feature of it, sharing the same user interface. TkSee encompasses a set of integrated tools such as a static analyser and a line-based debugger and more features are being added to cover more information requirement within the same environment. Note that the porting to TkSee that runs under UNIX caused many compromises in a way that reduced significantly the elegance and usability of the UI. We are still trying to balance the increased usability vs. the increase integrity.

# References


Ashcraft 99 Ashcraft M., *Fundamentals of cognition*, Addison Wesley Longman, (1999)

Baddely 86 Baddely A. D., *Working Memory*, Oxford: Clarendon Press (1986)

Baddely 74 Baddely A. D., Hitch G. J. Working memory. In G. H. Bower (Ed.), *The psychology of learning and motivation: Advances in research and theory,* vol.8 (1974)

Barnard 93 Barnard P.J., May J., "Cognitive Modeling for User Requirements", in Byerley, P.F., Barnard, P.J., & May, J. (eds) *Computers, Communication and Usability: Design issues, research and methods for integrated services,* Elsevier: Amsterdam (1993), pp.101-145.

Britannica 98 Britannica encyclopaedia, CD edition, (1998)

Brooks 83 Brooks R, "Toward a theory of the comprehension of computer programs", *International Journal of Man-Machine studies,* vol.18, No.6, (1983)

Busse 98 Busse D.K. and Johnson C.W. "Using a Cognitive Theoretical Framework to Support Accident Analysis" , in *Human Error, Safety, and Systems Development*, Seattle (1998)

Card 83 Card Stuart, Moran Thomas, Newell Allen, *The Psychology of Human Computer Interaction*, Erlbaum Assoc., Hillsdale NJ, (1983)

Carroll 89 Carroll J M, Campbell, R L. "Artifacts as psychological theories: The case of human computer interaction", *Behavior & Information Technology*. Vol. 8, no.4, Jul-Aug (1989)

Chen 90 Chen S, Heisler KG, Tsai WT, Chen X, Leung E. "A model for assembly program maintenance," *Journal of Software Maintenance*, vol.2, no.1, March (1990)

Chikofsky 90 Chikofsky I, Cross JH, "Reverse Engineering and Design Recovery: A Taxonomy" *IEEE Software*, vol.7, no.1, Jan. (1990)

Chinn 95 Chinn Wynne, Todd Peter, "On the use, Usefulness, and Ease of use of structural equation modeling in MIS research: A note of caution" *MIS Quarterly*, vol.19, no.2, (1995)

Cimitile 95 Cimitile A, Vissaggio G, "Software salvaging and the call dominance tree", *Journal of Systems & Software*, Vol. 28, no. 2, (1995)

Compeau 95 Compeau D, Higgins, C., "Computer Self Efficacy: Development of a Measure and Initial Test", *MIS Quarterly*, June (1995)

Corbi 90 Corbi TA, "Program Understanding : challenges for the (1990s." *IBM Systems Journal*, vol.28, no.2, (1989)

Davis 89 Davis D., "Perceived Usefulness, Perceived Ease Of Use And User Acceptance Of Information Technology," *MIS Quarterly*, Sept. (1989)

Dishaw 98 Dishaw M. and Strong D. "Supporting Software Maintenance With Software Engineering Tools: A Computed Task-Technology Fit Analysis", *The Journal of System and Software* vol. 44 (1998)

Ducasse 99 Ducasse M., "Opium: An extendable trace analyser for Prolog", *The journal of Logic programming,* vol. 39, (1999)

| | |
|---|---|
| Dyer 01 | Dyer H., McGhee P., ”Psychology and Science”, *http://www.sar.bolton.ac.uk/ltl/lecture2/intro.htm* |
| Elnozahy 99 | Elnozahy EN, "Address Trace Compression Through Loop Detection and Reduction", *ACM. Performance* Evaluation Review, vol.27, no.1, June (1999) |
| Erdos 98 | Erdos K., Sneed .M., “Partial comprehension of complex programs (enough to perform maintenance),” *Proc. 6th International Workshop on Program comprehension* (1998) |
| Fjelstad 83 | Fjelstad R.K., Hamlen W.T., “Application Program maintenance study: report to our respondents”, *Tutorial on software maintenance*, April (1983) |
| Flor 91 | Flor Nick, & Hutchins Edwin, Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance. *Empirical Studies of Programmers: Fourth Workshop* (1991). |
| Goodhue 95 | Goodhue D. and Thompson, R., “Task-Technology Fit and Individual Performance”, *MIS Quarterly,* vol.19 (1995) |
| Jcheck | http://www.compuware.com/products/numega/dps/java/jc.htm |
| Jerding 97 | Jerding, D. Rugaber, S., "Using Visualisation for Architecture Localization and Extraction." *Proc. 4’th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, Oct. (1997) |
| Jerding 97 b | Jerding D, Stasko J and Ball T, “Visualising interactions in program executions”, *Proceedings of the (1997 International Conference on Software Engineering*, ICSE 97. ACM. (1997) |
| Jinsight | http://www.research.ibm.com/jinsight/ |
| Jorgensen 95 | Jorgensen M., “An empirical study of Software Maintenance Tasks”, *Journal of Software Maintenance*, vol.7, Jan.-Feb. (1995) |
| Joyce 96 | Joyce M., “Shifting science”, http://www.arts.unimelb.edu.au/amu/ucr/student/1996/m.joyce/fpeople.htm |
| Kintsch 98 | Kintsch W., *Comprehension: a Paradigm for Cognition*, Cambridge university press, (1998) |
| Korel 97 | Korel B., Rilling J., "Dynamic program Slicing in Understanding of program Execution", *Proceedings Fifth International Workshop on Program Comprehension,.* (1997) |
| Korel 98 | Korel, B., Rilling, J. “Program slicing in understanding of legacy system”, ", *Proceedings Fifth International Workshop on Program Comprehension* (1998) |
| Kotovsky 85 | Kotovsky K., Hayes, J.R., Simon, H.A “Why are some problem hard? Evidence from Tower of Hanoi”*, cognitive psychology*, 17, 248-294 (1985) |
| Kozaczynski 89 | Kozaczynski W., Ning JQ. “SRE: a knowledge-based environment for large-scale software re-engineering activities.” *Proceedings. 11th International Conference on Software Engineering,* Washington, DC (1998) |
| Kunz 94 | Kunz T., "Reverse engineering distributed application: an event based abstraction tool." *International Journal of Software Engineering and Knowledge Engineering,* vol. 4, no. 3, September (1994) |

Kunz 97 Kunz T and Seuren M, "Fast detection of communication patterns in distributed executions", *CASCON 97*.

Lakhotia 93 Lakhotia A., “Understanding Someone Else’s code: Analysis of experience ”, *Journal of systems and software,* vol. 26, (1993)

Lakhotia 93-tr Lakhotia A., “Analysis of experiences with modifying computer programs”, *University of Southwestern Louisiana, technical report TR–93-5-6*, (1993)

Lakhotia 94 Lakhotia A., “What Is Appropriate Abstraction For Understanding And Reengineering A Software System”, `*Reverse Engineering Newsletter*, Sept. (1994)

Larus 93 Larus JR, "Efficient program tracing", *IEEE Computer*, vol.26, no.5, May (1993)

Laudan 96 Laudan L., “*Beyond positivism and relativism*” Westview press, (1996)

Lethbridge 2000 Lethbridge T. C., “Integrated Personal Work Management in the TkSee Software Exploration Tool “, *Workshop on Constructing Software Engineering Tools*, ICSE (2000)

Lethbridge 96 Lethbridge T.C. and Singer J., "Strategies for Studying Maintenance", *Workshop on Empirical Studies of Maintenance*, Monterey, November (1996)

Lethbridge 97 Lethbridge T. C., and Anquetil N., “Architecture of a Source Code Exploration Tool: A Software Engineering Case Study”, *TR-97-07 University of Ottawa* (1997)

Lethbridge 98 Lethbridge T. C., and Singer J. , "From Work Patterns to Requirements", *International Journal of Human-Computer Studies*, December (1998).

Lientz 78 Lientz B., et. al. “Characteristics of application software maintenance”, *Comm. Of the ACM*, June (1978)

Lloyd 99 Lloyd K.B., Jankowski D.J., “A cognitive information processing and information theory approach to diagram clarity: A synthesis and experimental investigation” *Journal of Systems and Software*, vol. 45 (1999)

Marchionini 88 Marchionini G., and Shneiderman B., "Finding Facts and Browsing Knowledge in Hypertext Systems," *IEEE Computer* vol. 21, (1988)

McCrickard 96 McCrickard D.S. and Abowd G.D., ”Assessing The Impact Of Changes At The Architectural Level: A Case Study On Graphical Debuggers,” *Proceedings. International Conference on Software Maintenance* (1996)

Pal 98 Pal C., "A Technique for Illustrating Dynamic Component Level Interactions Within a Software Architecture", *CASCON 98,* (1998)

Pennington 87 Pennington N., “Comprehension strategies in Programming” in Olson, G.M. , Sheppard, S., et al. (Eds) *Empirical studies of programmers: Second workshop Human/computer interaction, Vol. 7*. Norwood, NJ, Ablex Publishing (1987*)*

Pfleeger 98 Pfleeger S.H., *Software engineering: theory and practice* Prentice Hall, New Jersey, NJ pp.416 (1998)

Rugaber 95 Rugaber S., “Program comprehension”, *TR-95 Georgia Institute of Technology*, (1995)

Shneiderman Shneiderman B. and Carroll J., “Ecological studies of professional

88 programmers”, *Communication of the ACM*, Nov. (1988)

Singer 97 Singer J., Lethbridge T. C., Vinson N. and Anquetil N., "An Examination of Software Engineering Work Practices", *CASCON '97*, Toronto, October, (1997)

Singer 98 Singer J., and Lethbridge T.C., "Studying Work Practices to Assist Tool Design in Software Engineering", *International Workshop on Program Comprehension*, (1998)

Slovic 72 Slovic P., “From Shakespeare to Simon: Speculation and some evidence – About man’s ability to process information” *Oregon Research Bulletin* vol.12 (1972)

Soloway 88 Soloway E., Pinto J., Letovsky S., Littman D., Lampert R., "Designing documentation to compensate for delocalized plans" , *Comm. of the ACM*, Nov. (1988)

Storey 97 Storey. M.A., Wong K., Muller H.A., “How do program understanding tools affect how programmers understand programs?” *Proceedings of the Fourth Working Conference on Reverse Engineering,* (1997)

Storey 99 Storey M.A. et. al., “Cognitive design elements to support the construction of a mental model during software exploration”, *Journal of Systems & Software* , vol.44, no.3, Jan. (1999)

Swanson 89 Swanson E.B., Beath C.M, *Maintaining information systems in organizations*, Wiley, New York (1989)

Teteishi 94 Teteishi A., "Filtering Run Time Artefacts Using Software Landscape", *M.Sc. thesis, university of Waterloo*, (1994)

Tilley 96 Tilley S. R., Paul S. and Smith D. B., “Towards a Framework for Program Comprehension”, *4th Workshop on Program Comprehension*, (1996)

Tseng 98 Tseng Y., "Multilingual keyword extraction for term suggestion.", *Proceedings of 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, Melbourne, Vic., Australia. Aug. (1998)

Turner 96 Turner A. et. al "A predication semantics model of text comprehension and recall” in *Models of understanding text*, Britton B. et al (Ed). Mahwah, NJ: Lawrence Erlbaum Ass, pp. 33-71 (1996)

Umanath 94 Umanath N.S., Vessey I. “Multiattribute data presentation and human judgment: a cognitive fit perspective”, *Decision Sciences*, vol.25, no.5-6, Sept.-Dec. (1994)

Vessey 91 Vessey I., “Cognitive Fit: A Theory-Based Analysis Of The Graphs Versus Tables Literature” *Decision Sciences* vol.22, (1991)

Von 93 Von Mayrhauser A. , Vans A.M., “From program comprehension to tool requirements for an industrial setting”, *Proceedings IEEE Second Workshop on Program Comprehension*, (1993)

Von 95 Von Mayrhauser A. , Vans A.M. ,“Program comprehension during software maintenance and evolution” *IEEE computer* , vol.28, no.8, Aug. (1995)

Von 96 Von Mayrhauser A., Vans, A.M., “On the role of program understanding

in re-engineering tasks*" Proceedings IEEE Aerospace Applications Conference*. vol. 2, (1996)

Von 96 b ?? Von Mayrhauser A., Vans A.M., "Hypothesis-Driven Understanding Process During Corrective Maintenance of Large Scale Software" *Proceedings. Fourth Workshop on Program Comprehension*. (1996)

Von 98 Von Mayrhauser A., Vans A.M., "Program Understanding Behavior During Adaptation of Large Scale Software*", Proceedings. 6th International Workshop on Program Comprehension* (1998)

Wilde 98 Wilde N. and Casey C., "Reverse engineering of software threads: A design recovery technique for large multi-Process systems", *Journal of Systems & Software*, vol.43, no.1, Oct. (1998)

Wilde 92 Wilde N., Gomez J., Gust T., Strasburg D., "Locating User Functionality in Old Code", Proc., *IEEE Conf. on Software Maintenance*, Orlando, November (1992)

Wilde 93 Wilde N., Matthews P., Huitt R., "Maintaining Object-Oriented Software", *IEEE Software*, Jan. (1993)

Wilde 95 Wilde N. and Scully M., "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, Vol. 7, No. 1, January (1995)

Welie 01 van Welie M., "*Task-based User Interface Design*" , Ph.D. Thesis (2001) http://www.cs.vu.nl/~martijn/gta/docs/Welie-PhD-thesis.pdf

Wilson 01 Wilson M., "Theory and Practice from Cognitive Science, in C. Stephanidis (ed.)" in *User Interfaces for All: Concepts, Methods, and Tools.* 137-164. London: Lawrence Erlbaum Associates. 2001.

Woods 96 Woods S. and Yang Q., "The program understanding problem: analysis and a heuristic approach", *Proceedings of the 18th International Conference on Software Engineering*. (1996)

UACSD The University of Alberta's Cognitive Science Dictionary http://web.psych.ualberta.ca/Emike/Pearl_Street/Dictionary/dictionary.html

Yan 98 Yan J.C. and Schmidt M.A., "Constructing space-time views from fixed size trace files- getting the best of both worlds," *Parallel Computing: Fundamentals, Applications and New Directions. Advances in Parallel Computing*. Vol.12. Elsevier. (1998)