A Cognitively Aware Dynamic Analysis Tool for Program Comprehension

Iyad Zayour
Alumni of the School of Information Technology and Engineering
University of Ottawa, Ottawa, Ottawa
iyad@alumni.uottawa.ca

Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University
Montréal, Québec, Canada
abdelw@ece.concordia.ca

Abstract

Software maintenance is perhaps one of the most difficult activities in software engineering. Reverse engineering tools aim at increasing its efficiency. However, these tools suffer from the low adoption problem. To be adoptable, a tool has to reduce the cognitive overload faced by software engineers when performing maintenance tasks. In this paper, we identify the cognitive difficulties encountered by software engineers during software maintenance based on an experiment we conducted in an industrial setting. Our results support the idea that comprehension during software maintenance tasks consists of a process of mapping between the static and the application domains via the dynamic domain. We present a prototype dynamic analysis tool, called DynaMapper, designed to support these domain mappings. A preliminary evaluation of the tool is presented to assess its effectiveness.

Keywords: dynamic analysis, software maintenance, cognitive models, reverse engineering tools

1. Introduction

It is estimated that 50% to 70% of software costs are spent on maintenance [7]. Maintaining a large software system, however, has been shown to be an inefficient process; software engineers must understand many parts of the system prior to undertaking the maintenance task at hand. The difficulties encountered by maintainers are partially attributable to the fact that changes made to the implementation of systems are usually not reflected in the design documentation. This can be due to various reasons including a lack of effective round-trip engineering tools, time-to-market constraints, the initial documentation being poorly designed, etc. As such, program comprehension is considered to be a key bottleneck of software maintenance [10].

Reverse engineering research aims to reduce the impact of this problem by investigating techniques and tools that can help extract high-level views of the system from low-level implementation details. Reverse engineering tools build on the knowledge obtained from studying how programmers understand programs.

There exist several program comprehension models that describe the cognitive difficulties encountered by programmers when understanding large programs (e.g., [1, 9, 12]). However, these cognitive models tend to describe the major internal cognitive activities in a generic way. In this paper, we rely on the knowledge provided by these models, and expand it by investigating in more detail the practical problems that can be addressed by a reverse engineering tool. We present the difficulties and associated cognitive overloads encountered during software maintenance and then we present our approach based on dynamic analysis that addresses these difficulties.

This paper is organized as follows: In Section 2, we present our approach of how we identified the difficulties in software maintenance. In Section 3, we describe a dynamic analysis tool, called DynaMapper, which addresses these difficulties, followed with related work. We conclude the paper in Section 5.

2. Cognitive Overloads

Identifying cognitive overloads is possible by observing the work practices of software engineers, by asking software engineers to identify them, or even by introspection [6]. Introspection consists of relying on the proper experience of the software engineers in doing software maintenance to detect cognitive difficulties that other software engineers face when performing maintenance tasks. In fact, the personal experience of the authors of this paper in doing software maintenance was highly valuable in determining the overloads identified in this paper.

To identify the cognitive overloads during maintenance, we worked with software engineers from a telecommunications company that maintains a large legacy software system, which was developed in 1982. It includes a real-time operating system. The system is written in a proprietary structured language and contains over 2 million lines of code. The company suffers from the high cost of maintaining the system.

We focused on small corrective maintenance tasks that are often assigned to newly hired software engineers, with little knowledge of the structure of the system. We observed their work practices while asking them to think aloud. We summarize the result of our observations in the following steps:

First, software engineers start by understanding the maintenance request by reading its description. The next activity consists of locating the code relevant to the problem, and mapping problem behaviour to the corresponding code. This involves locating a starting point in the code, which is typically a snippet of code that is part of the execution path of the current problem.

Once the starting point is located, they proceed with identifying the rest of the code responsible for the maintenance problem. For this purpose, they follow events in program behaviour and try to match them to code. Once the code has been located, the next step consists of understanding this code as executed. The code statements are mentally visualised as executed (symbolic execution) and mapped with the problem behaviour.

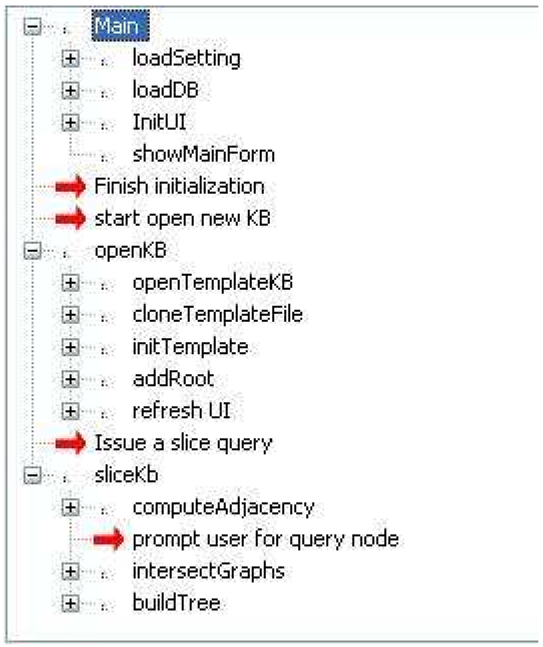
The maintenance activities involve a substantial mapping from program behaviour to source code and then mapping from source code to behaviour. In other words, it consists of a series of mapping activities between the static domain and the application domain. The static domain consists primarily of source code including comments and any additional documentation that describes the design and implementation of software. The information in this domain is always available, fixed, and explicit. The application domain is defined here as the functionality of the program from the user's perspective. In other words, it 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. This bi-directional mapping seems to be an activity that places a heavy load on the human cognitive resources. This is because this mapping involves the intermediate dynamic domain (that consists of run-time information) that is largely invisible and requires to be mentally constructed.

Accordingly, our primary goal is to reduce the cognitive cost of inter-domain mapping; hence, dynamic information has to be generated and presented in a efficient way. This information has to act as an explicit representation of the dynamic domain, thus reducing the cognitive effort that would otherwise be required to mentally construct it. Since dynamic data such as program traces can be very challenging to be managed and comprehended, let alone to be used for domain mapping, our representation of the dynamic domain has to support the inter-domain mapping in an efficient way.

3. Domain Mapping Using Traces

We embarked onto designing a prototype tool, referred to as DynaMapper, which supports the identified cognitive difficulties. The tool should sub-contract from the working memory 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 of a larger mathematical problem. Another example is using paper to store intermediate results of multiplication, instead of storing the results mentally.

The tool should also take over some cognitive load by explicitly representing the implicit processing constructs and operations that go on in the working memory using its processing power and the screen display (e.g., extracting and displaying the call tree on the screen).



```
Main
    loadSetting
    loadDB
    InitUI
    showMainForm
Finish initialization
start open new KB
openKB
    openTemplateKB
    cloneTemplateFile
    initTemplate
    addRoot
    refresh UI
Issue a slice query
sliceKb
    computeAdjacency
    prompt user for query node
    intersectGraphs
    buildTree
```

Figure 1. Snapshot of DynaMapper call hierarchy

3.1 DynaMapper Description

DynaMapper is a trace analysis tool that aims at creating a dynamic representation of data while facilitating domain mapping using program trace generation and processing. In addition to domain mapping, DynaMapper provides several other features that facilitate the comprehension of program behaviour, and visualisation of traces.

The input for DynaMapper is a trace file that contains the names and call levels for all the routines that are executed during a scenario. The choice of routines as the level of granularity of the trace was driven by our observation of software engineers. When they explore code in order to trace control flow, they look at routine calls more than at other programming statements. This may be because routine names tell a lot about execution, and in our subject system a high percentage of the statements are routine calls.

The first challenge in creating a useful dynamic representation is to deal with the size of traces that is usually very large. DynaMapper performs several processing phases on the input file to prepare it for visualisation. First, any redundancy created by calls to routines within loops or by recursion is detected, removed and replaced by its number of occurrences. Next, other kinds of redundancy are detected, such as routine sequences that occur repetitively but non-contiguously in several places in the trace. Moreover, routines that contribute little information to the trace, called utility routines, are identified and removed. For this purpose, we used utility detection techniques based on fan-in analysis [3]. Finally, the processed trace is visualised as a call hierarchy in a user interface (see Figure 1). The user can expand and contract particular sub-hierarchies, show or hide patterns, and restrict the entire display to particular levels of depth. A summary of trace reduction techniques can be found in [4].

3.2 Bookmarks

The novel aspect of DynaMapper compared to existing trace analysis tools is the ability to perform mapping between the application and static domains (via the dynamic domain). DynaMapper supports mapping using a special trace entry called a *bookmark*. A bookmark is a kind of trace annotation – a special node that can be inserted inside the trace to indicate the occurrence of an application domain visible event. 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, a software engineer will identify this node and thus identify what part of the trace occurs before, i.e., the one leading to the error message, and the part that comes after. Instead of dealing with the entire trace as one monolithic block, the user can deal with it as set of segments that proceed or succeed application level events like the error message that is visible from the application domain.

This way, a bookmark can be inserted in the trace to identify the relative position of such an event within the trace. 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. Figure 1 shows an example of a trace displayed as a call tree and annotated with bookmarks (having an arrow icon). For example, the first bookmark shown in Figure 1, labelled "Finish initialization", indicates that at this point of the program behaviour, an initialization phase occurred.

Bookmarks are created by instrumenting the code to produce distinguishable trace entry whenever certain code with application level visibility is identified (based on any clue available in the source code). This is like inserting print statements inside the code to track the proper time and order of occurrence of special events during application execution.

Bookmark Types:

The choice of events to instrument is open and depends on the type of applications. An obvious choice of application-visible events in code is the user interface and program output. One can choose to bookmark many or all user interface events such as screen display or button pressed or even logged events. We found that all program output (e.g., error message) that is generated during exceptional (erroneous) behaviour are very useful for maintenance tasks.

Also DynaMapper supports interactive bookmaking both during application execution and during trace or program exploration. During application execution, a target application can be instrumented so that it responds, while running, 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 special bookmark entry. This can be very useful during maintenance where the SE can bookmark the program behavior while reproducing problems, so for example, to mark the start of the malfunctioning in program behavior.

Code Bookmarks:

The granularity of trace bookmarking is determined by the size of code that runs between two interactive events (i.e., where a user interface is generated or an application wait for a user input so the user can press the hot key to enter a bookmark). 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. In minimum interactive systems, the size of trace between two bookmarks can be still significant.

Therefore, DynaMapper permits the user to choose a routine during trace or static program exploration and mark it as “application domain visible”. These routines can have names that directly reflect application domain concepts such as “dialNumber” or ‘postInvoice”. Once tagged as such, a special instrumentation is inserted so that this routine execution will produce bookmarks node during trace generation in addition to their normal routine trace entry.

Navigation and Visualization:

Traces even after compression can be very large. Just finding the visually distinguished nodes of bookmarks within the trace by manual exploration can be very inefficient. Therefore, DynaMapper offers several features to help making use of bookmarks such as:

- Searching for a bookmark by its text so if found its node is selected and made visible.
- The "view bookmarks only" operation that collapses all nodes in a way that ensure that all bookmarks nodes are made visible (see Figure 2). This offers a bookmark view of the call tree where the user can easily locate a certain bookmark and expand the call tree guided by the bookmarks.
- The slice operations permits removing all trace entries except those that are located between two or more bookmarks

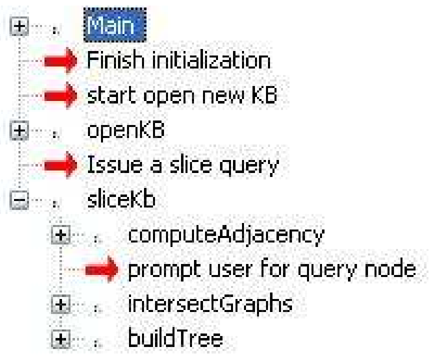


Figure 2. Same call tree as in Figure 1 but after “view bookmarks only” operation

3.3 Evaluation

In order to evaluate the usefulness of DynaMapper for mapping domains, we designed an experiment in which five software engineers of the telecommunications company were asked to a) identify the part of trace (displayed as call tree) corresponding to application visible events (mapping from application to static), b) describe what application events this trace is causing (mapping from static to application).

In the experiment, we first asked the software engineers to locate the code that needed to be maintained according to a specific maintenance request without using DynaMapper. This was not trivial given the size of the subject system. We asked the participants to use explicitly the bookmarks after we had demonstrated how they work. The participants inserted bookmarks before each interactive application event (when the application is waiting for a user input) that preceded the feature they had to locate. Using bookmarks 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 they inserted and only investigate the few call sub-trees after that bookmark.

Results have shown that the bookmark feature to be particularly useful in finding the code relevant to an interactive application visible events (e.g., UI event). Bookmarks were also useful to identify an ending point in the trace. That is, the trace segment relevant to maintenance request (program behaviour) could be identified and sliced reducing the space in which comprehension needs to take place.

However, while the application to dynamic domain mapping was effective, the opposite mapping was not as much useful. After locating the starting point, bookmarks were found to be less used. As our model of difficulties suggests, the software engineer’s effort shifts to the mapping from the static to application domain after the code is identified. This mapping takes place at a lower level of granularity where seldom interactive bookmarks were present to facilitate the mapping from trace to the application domain.

4. Related Work

DynaMapper can be considered to belong to the set of tools that apply dynamic analysis to aid in the behavioural understanding of programs. A survey of existing trace analysis tools is presented by Hamou-Lhadj et al. [4]. Most of these tools provide the dynamic information in terms of visualisation at the component level that can either be user-defined as in IsVis [5], showing modules and subsystems as in the “Run Time Landscape” [11], or at the physical source file level as in RunView [8]. None of these tools, however, is developed taking into account a comprehensive framework oriented towards understanding the cognitive overloads that occur when doing software maintenance. In addition, we are not aware of any tool that supports the concept of bookmarks for domain mappings as described in this paper.

A close research area to the work presented in this paper consists of feature location – Identifying the most relevant components that implement a given feature. There exist several feature location techniques (e.g., [2, 14]). These techniques, however, operate in after-the-fact fashion. In other words, a trace (or many traces) has to be generated by exercising the feature under study and then heuristic-based techniques are applied to identify the feature most relevant components. In this research, we propose that the use of bookmarks, which are conceptual tags inserted in the source code, will lead to a trace that is segmented in such a way that software engineers can easily map the behaviour embedded in a trace to the corresponding code.

Concept assignment is concerned with finding the correspondence between high-level domain concepts and code fragments. Concept assignment main task is of discovering individual human-oriented concepts and assigning them to their implementation-oriented counterparts in the subject system [10]. 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.

5. Conclusion and Future Work

In this paper, we developed a tool called DynaMapper that allows the mapping between static and program behaviour, based on identifying cognitive difficulties facing software maintainers.

The key feature of DynaMapper is the concept of bookmarks, which is an instrumental feature that segments a trace into behavioural parts that a user can understand. Bookmarks, however, as proposed in this paper may not produce a small enough segment especially for large and non-interactive software systems. We are investigating automatic identification of routines that have application domain visibility so that their presence in a trace would play the role of a bookmark.

Finally, we also need to conduct large-scale experiments involving a larger number of software engineers in order to better assess the applicability of bookmarks to reduce cognitive overloads through domain mappings.

6. References

[1]. Brooks R, “Toward a theory of the comprehension of computer programs”, *International Journal of Man-Machine studies 18(6),* pp. 542-554, 1983.

[2]. Greevy O., Ducasse S., and Girba T., "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis", *In Proc. of 21st International Conference on Software Maintenance*, pp. 347-356, 2005.

[3]. Hamou-Lhadj A. and Lethbridge T. C., "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the 14th IEEE International Conference on Program Comprehension,* pp. 181-190, 2006.

[4]. Hamou-Lhadj A. and Lethbridge T. C., “A Survey of Trace Exploration Tools and Techniques”, *In Proc. of the International Conference of the Centre for Advanced Studies,* IBM Press, pp. 42-54, 2004.

[5]. Jerding, D., Rugaber, S., "Using Visualisation for Architecture Localization and Extraction", *In Proc. of the 4th Working Conference on Reverse Engineering,* pp.267-84, 1997.

[6]. Lakhotia A, “Understanding Someone Else’s code: Analysis of Experience”, *Journal of Systems and Software,* vol. 23, pp.269-275, 1993.

[7]. Lientz B., Swanson E. B., and Tompkins G. E. “Characteristics of application software maintenance”, *Communications of the ACM, 21(6),* pp 466-471, 1978.

[8]. McCrickard, D. S., and Abowd, G. D., “Assessing The Impact of Changes at the Architectural Level: A Case Study on Graphical Debuggers”, *In Proc. of the International Conference on Software Maintenance*, pp. 59-69, 1996.

[9]. Pennington N., “Comprehension Strategies in Programming”, *In Proc. of the 2nd Workshop on Empirical Studies of Programmers,* pp. 100-113. 1987.

[10]. Rugaber, S., “Program Comprehension” TR-95, Georgia Institute of Technology, 1995.

[11]. Teteishi, A., "Filtering Run Time Artefacts Using Software Landscape", *M.Sc. Thesis, University of Waterloo*, 1994.

[12]. Von Mayrhauser A, Vans A. M., “Program comprehension during software maintenance and evolution”, *IEEE Computer, 28 (8),* pp.44-55, 1995.

[13]. Wilde N. and Scully M., "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice, 7(1)*, pp. 49 – 62, 1995.

[14]. Woods S. and Yang Q., “The program understanding problem: analysis and a heuristic approach”, *In Proc. of the 18th International Conference on Software Engineering,* pp.6-15, 1996.