Andy Zaidman, Abdelwahab Hamou-Lhadj, Orla Greevy, David Röthlisberger
(*editors*)

PCODA'08
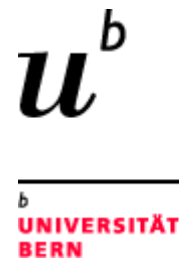
4th International Workshop on

Program Comprehension through Dynamic Analysis

co-located with the 15th International Working
Conference on Reverse Engineering (WCRE'08)

October 16th, 2008 – Antwerp, Belgium

TUDelft
Delft University of Technology

UNIVERSITÉ **Concordia** UNIVERSITY

u^b
b
**UNIVERSITÄT
BERN**

Contents

On Execution Traces

Pattern or Feature Detection

Tools

Applications of Dynamic Analysis

Program Chairs

Orla Greevy
Software Engineering gmbh
Switzerland
greevy@sw-eng.ch

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

David Röthlisberger
Software Composition Group
Institut fur Informatik und angewandte Mathematik
University of Bern
Switzerland
roethlis@iam.unibe.ch

Andy Zaidman
Software Engineering Research Group
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

Program Committee

Edna Braun
University of Ottawa, Canada
Constantinos Constantinides
Concordia University, Canada
Serge Demeyer
University of Antwerp, Belgium
Wim De Pauw
IBM, USA
Philippe Dugerdil
Haute école de gestion de Genève, Switzerland
Adrian Kuhn
University of Bern, Switzerland
Adrian Lienhard
University of Bern, Switzerland
Leon Moonen
Simula Research Laboratory, Norway

Visualizing an Execution Trace as a Compact Sequence Diagram Using Dominance Algorithms

Yui Watanabe, Takashi Ishio, Yoshiro Ito, Katsuro Inoue

Osaka University

1-3 Machikaneyama, Toyonaka, Osaka, 560-8531, Japan

{wtnb-y, ishio, yito, inoue}@ist.osaka-u.ac.jp

Abstract

Visualizing an execution trace of an object-oriented system as sequence diagrams is effective to understand the behavior of the system. However, sequence diagrams extracted from an execution trace are too large for developers to inspect since a trace involves a large number of objects and method calls. To support developers to understand extracted sequence diagrams, it is necessary to remove the less important details of the diagrams. In this paper, we apply a dominance algorithm to a dynamic call graph among objects in order to detect and remove local objects contributing to internal behavior of dominator objects. The case study shows our approach automatically removed about 40 percent of the objects from execution traces on average.

1. Introduction

Visualizing an execution trace of an object-oriented system as sequence diagrams is effective to understand the behavior of the system since understanding dynamic behavior of an object-oriented system is more difficult than understanding its structure [1, 19]. A sequence diagram extracted from an execution trace visualizes actual collaborations of objects that provide a larger unit of program comprehension than classes [14]. Extracted diagrams also enable developers to compare actual behavior of a program with its design.

Although several tools supported such UML-based visualization [4, 7, 15], a sequence diagram extracted from an execution trace may be too large for developers to inspect since a trace involves a large number of objects and method calls. A simple approach to reducing the size of a sequence diagram is a filter to exclude objects and method calls using their package, class and method names. Such name-based filtering approach is effective to remove well-known library such as JDK classes from sequence diagrams. How-ever, to filter out objects and method calls in an application, developers have to know important packages, classes and methods before understanding the system. In addition, the approach does not work when a particular set of objects is more important than other instances of the same class. For example, a web application using a database may create a large number of objects representing records in a database but use only few of them to construct an output for users.

In this paper, we propose to apply dominance algorithms to instance-level filtering. While objects shared by several features are important to understand the relationship among features [10], *local objects* contributing to only internal behavior of their *dominator* objects are less important. We apply dominance algorithms to detect and remove local objects in execution traces. In our approach, we first translate an execution trace to a dynamic call graph whose vertices and edges representing objects and method calls in execution traces. Then, we compute dominance relation among objects. A dominator object and objects dominated by the dominator form a cluster such that objects out of the cluster access only the dominator object. We regard objects in a cluster except for the dominator as local objects. A sequence diagram excluding local objects is still precise; the diagram includes all interactions among dominator objects shown in the diagram.

We have implemented our approach with an iterative dominance algorithm [2] and our sequence diagram extraction tool named Amida [7]. We conducted a case study on four implementations of a web application, and found that 40% of objects are categorized into local objects on average. Although we need further case studies on software in different domains, our approach is promising to provide a compact sequence diagram extracted from an execution trace to developers .

The rest of this paper is organized as follows. Section 2 explains the background of this research. Section 3 describes our approach to filtering local objects from sequence diagrams. Section 4 shows the result of a case study. Section 5 describes the summary and future work.

2. Background

Visualization of dynamic behavior of object-oriented programs is effective for program understanding and debugging. A popular approach is UML-based visualization [1]. For example, JIVE supports object interaction diagram and sequence diagram [4]. Shimba [15] and Amida [7] also support sequence diagram. To draw a compact diagram, Shimba can replace objects in the same package as a package object. Amida detects loops and recursive calls in a trace [16]. Several data compression approaches to detecting repeated method call sequences are investigated by Reiss [13].

Since UML-based visualization often outputs a large diagram, several new viewers and summarization approaches are proposed. Pauw proposed a simple left-to-right layout of a call tree that works well with zoom-in/out functionality [11]. Cornelissen proposed Circular Bundle View that visualizes an execution trace as a compact circular view [3]. These approaches are suitable to investigate an overview of a trace but not for developers to investigate the actual behavior of a program.

To summarize an execution trace before visualization, Hamou-Lhadj proposed a utilityhood function to identify utility methods [6]. Their utilityhood definition is based on a simple idea: many methods depend on utility methods while utility methods depend on few other methods. This approach simply excludes utility-like method calls from traces. Therefore, a summarized sequence diagram may miss method call events connecting objects; such a diagram would be a good overview of a trace but it does not support developers who would like to investigate the precise behavior.

Phase detection divides an execution trace into phases that are corresponding to functional units in the trace [12, 18]. Reiss uses statistical information of method calls [12]. Our approach monitors a working set of objects [18]. Although these approaches can divide a large sequence diagram into several smaller pieces, resultant diagrams may be still large for developers to investigate.

In this paper, we apply dominance algorithms to identify local objects contributing to internal behavior of a particular object. Our approach is based on a dynamic call graph whose vertices and edges represent objects and method calls in an execution trace, differently from utilityhood function based on static fan-in and fan-out of each method [6]. Dominance algorithm is already used for visualizing and navigating a program dependence graph [5]. We hypothesized that dominance algorithm would be effective for a dynamic call graph since many temporary objects are created to achieve a task in a system and such objects are locally used and destroyed after the task [9, 17].

Our approach is an instance-level filtering approach ex-cluding objects that are likely not important from an execution trace. A simple name-based filtering approach does not distinguish instances; it simply removes all instances of the class from a trace. Shimba implements another approach that replaces all instances with a single actor representing a class [15]. These class-based approaches are not applicable when an instance of a class is more important than other instances of the same class. For example, a web application using a database may create a large number of objects representing records in a database but use only few of them to construct an output for users. On the other hand, JIVE allows developers to hide member objects that are stored in fields of another object [4]. This approach is also instance-level but developers have to manually specify fields containing internal objects. Our approach automatically detects local objects from an execution trace.

3. Visualization of Dominator Objects

We apply dominance algorithms to detect and remove local objects from an execution trace in order to visualize the execution trace as a compact sequence diagram. Our approach comprises three steps. First, we construct a dynamic call graph from an execution trace. Next, we compute a dominance tree of the dynamic call graph. We regard objects dominated by a dominator as local objects contributing internal behavior of the dominator since only the dominator object interacts with dominated objects. Finally, we exclude local objects from an execution trace and visualize the resultant trace as a sequence diagram.

3.1. Dynamic Call Graph Construction

In the first step, we construct a dynamic call graph from an execution trace. Vertices and edges of a dynamic call graph represent objects and method calls in an execution trace, respectively. It should be noted that our approach is described based on Java language but applicable to other object-oriented languages.

An execution trace in this paper is a sequence of method call events. A method call event records at least a caller object c_{from} and a callee object c_{to}. To construct a dynamic call graph, first we prepare an empty graph G. For each method call from c_{from} to c_{to}, a directed edge from c_{from} to c_{to} is added to G.

We have a rule to deal with static methods such as `main` that are not belonging to any instance but a class. We translate each static method call into an individual vertex; for example, if a static method `Arrays.sort` is called twice in an execution trace, the resultant call graph contains two vertices $v_{Arrays.sort[1]}$ and $v_{Arrays.sort[2]}$. We distinguish these method calls since many static methods in utility classes such as `Arrays` and `Math` are independently
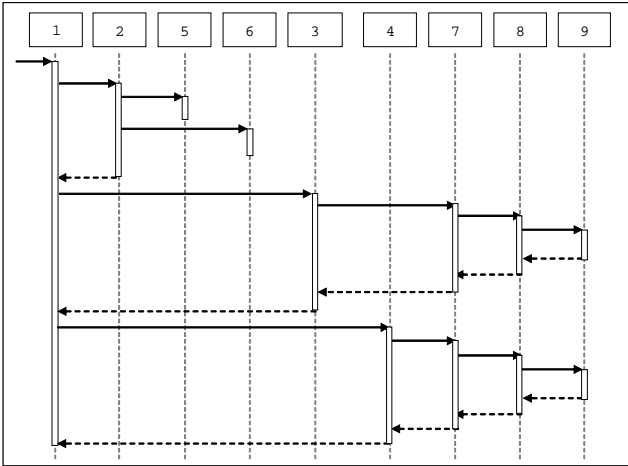
Figure 1. An example trace shown as a sequence diagram

Figure 2. A dynamic call graph of a trace in Figure 1 and its dominance tree

Figure 3. A reduced sequence diagram excluding local objects from Figure 1

called from many call sites in general.

The above rule for static methods enables us to obtain a dynamic call graph G with a root vertex corresponding to the entry point of a program, i.e., `main` method. An execution trace shown in Figure 1 is translated to a dynamic call graph shown in the left hand side of Figure 2.

3.2. Dominance Tree Construction

We apply a dominance algorithm to a dynamic call graph in order to compute *dominance relation* among objects in an execution trace. Dominance relation is a relation between two nodes in a directed graph G that has a single root node r. A vertex v *dominates* another vertex w in G if and only if every path from r to w contains v. Vertex v is the immediate dominator of w if v dominates w and every other dominator of w dominates v [8]. Dominance relation in a graph forms a dominance tree; the direct ancestor of node n in a dominance tree is the immediate dominator of n.

We can compute a dominance tree of a dynamic call graph since a dynamic call graph always has the single root `main` as we described in Section 3.1. In implementation, we have used iterative but fast dominance algorithm [2]. The right hand side of Figure 2 is an example of a dominance tree that is computed from the left call graph.

3.3. Visualizing Sequence Diagram

A dominance tree for objects involved in an execution trace indicates locality of interaction. Interaction among a dominator object and its descendant objects is invisible from other objects in the execution trace. Therefore, we regard descendant objects as local objects of their immediate

dominator object. A sequence diagram excluding local objects still involves all method calls among non-local objects in the diagram.

To visualize a trace excluding local objects as a sequence diagram, we classify objects into clusters. For each dominator d, we create a cluster $c(d)$ involving all objects dominated by d. The resultant clusters satisfy the following characteristics:

- Each cluster $c(d)$ has a single dominator object d.

- A method call from the outside of $c(d)$ always calls the dominator object d.

These characteristics enable us to visualize only dominator objects of clusters in a sequence diagram and hide their internal behavior. The hierarchy of a dominance tree

indicates the hierarchy of clusters; it allows developers to interactively visualize and inspect the detail of interesting clusters. Figure 3 visualizes a sequence diagram involving only the root object and its immediate descendant objects in the dominance tree in Figure 2.

4. Case Study

We have implemented our approach as a tool and conducted a case study to evaluate our approach. We have analyzed four implementations of an enterprise web application developed by four groups of developers in a training course. Four groups referred to the same specification and design documents but they implement the details of the system in different ways. We have prepared a use-case scenario that executes all features of the system according to the specification, and executed the scenario on four systems.

To obtain execution traces, we used an implementation of JVMTI, or Amida Profiler [7]. When recording execution traces, we filtered JDK standard classes out because of performance limitation. A call-back from JDK is regarded as an indirect call. In other words, when object o_1 called some JDK object and o_2 received a call-back, we recorded an indirect method call from o_1 to o_2.

After execution traces are obtained, we applied the tool to each of traces. Using Amida Viewer, we visualized a trace including only objects that are immediately dominated by `main` as a sequence diagram. We have compared the resultant diagrams with diagrams directly extracted from traces without our approach.

Table 1 and Table 2 shows the number of objects and method calls in each of execution traces before and after applying our approach. Since the target systems are implemented in Model-View-Controller architecture, we have categorized objects in traces into four categories: Model, View, Controller and Other. `Total` row shows the total number of objects in a trace. `Model` shows the number of objects whose classes represent data model and business logic. `View` includes only JSP objects. `Controller` includes Action, Servlet and RequestProcessor objects. `Other` includes other utility objects and static objects. It should be noted that we have regarded a static (class) object as a single object in Table 1, although each static method call is translated to an individual vertex when applying a dominance algorithm. In Table 1, the column `Before` and `After` respectively indicate the number of objects in a trace and the number of objects directly dominated by `main`. In Table 2, the columns indicate the number of method calls shown in sequence diagrams before and after our approach removes local objects. In both tables, `Ratio` is computed as follows.

$$\text{Ratio} = \frac{\text{After}}{\text{Before}} \times 100(\%)$$

Table 1. The number of objects in execution traces

System	Type	Before	After	Ratio(%)
A	Total	286	169	59.1
	Model	259	145	56.0
	View	9	9	100.0
	Controller	11	11	100.0
	Other	7	4	57.1
B	Total	300	176	58.7
	Model	273	152	55.7
	View	9	9	100.0
	Controller	11	11	100.0
	Other	7	4	57.1
C	Total	312	183	58.7
	Model	285	159	55.8
	View	9	9	100.0
	Controller	11	11	100.0
	Other	7	4	57.1
D	Total	354	4	1.1
	Model	326	0	0.0
	View	9	0	0.0
	Controller	11	0	0.0
	Other	8	4	50.0

Table 2. The number of method calls in traces

System	Before	After	Ratio(%)
A	3371	2390	70.9
B	3797	2716	71.5
C	3862	2646	68.5
D	4506	133	3.0

While about 60% of objects are directly dominated by `main` in System A, B and C, System D involves only few objects dominated by `main`. This is because System D uses a kind of Façade object representing a system itself. The system object is similar to `main` method in other systems and immediately dominates about 60% of other objects.

For other three systems, we have investigated objects involved in sequence diagrams and local objects filtered out by our approach. Our approach did not remove `View` and `Controller` objects such as Action and JSP since these objects interact with one another. These objects are important to understand the behavior of systems since they implement user interface. On the other hand, our approach excluded many `Model` objects from sequence diagrams. The resultant sequence diagrams involve "Data" objects containing database records since these objects are short-lived but shared by business logic and user interface. Our approach filtered out data access objects named "DAO" that

construct "Data" objects from database since such data access objects are locally used by Action objects. Objects for searching database records are also excluded from sequence diagrams since they are only used in a search function.

In the case study, local objects excluded by our approach are the less important implementation details of the target systems, while the resultant diagrams retained important objects such as user interface and business logic. Although we need further case studies on software in different domains, our filtering approach is promising to exclude local objects from a trace and provide a compact sequence diagram to developers.

5. Conclusion

We have applied dominance algorithms to identify local objects contributing to only internal behavior of their dominator objects. Excluding local objects from execution traces simplifies sequence diagrams extracted from the traces. In the case study, we found only 60% objects are directly dominated by `main` method and the other 40% of objects are local objects. We have confirmed that local objects excluded by our approach are the less important implementation details of the target systems. We implemented the algorithm to filter local objects from a sequence diagram in Amida Viewer. The resultant sequence diagrams retain all interactions among non-local objects; therefore, the diagrams are suitable for developers to investigate actual behavior of programs.

In future work, we have to conduct further case studies on various software. We would like to evaluate how our approach collaborates with other filtering and visualization approaches. We are also interested in how architecture and design of software influence the effectiveness of our approach.

Acknowledgements

References

[1] L. C. Briand, Y. Labiche, and J. Leduc. Towards the reverse engineering of UML sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[2] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`, 2001.

[3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 49–58, 2007.

[4] J. K. Czyz and B. Jayaraman. Declarative and visual debugging in eclipse. In *Eclipse Technology Exchange*, 2007.

[5] R. Falke, R. Klein, R. Koschke, and J. Quante. The dominance tree in visualizing software dependencies. In *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis*, page 24, 2005.

[6] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 181–190, 2006.

[7] T. Ishio, Y. Watanabe, and K. Inoue. AMIDA: a sequence diagram extraction toolkit supporting automatic phase detection. In *Companion Volume of the Int'l Conference on Software Engineering*, pages 969–970, 2008.

[8] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[9] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[10] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings of the Int'l Conference on Program Comprehension*, pages 59–68, 2007.

[11] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, 2001.

[12] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the Int'l Workshop on Dynamic Analysis*, pages 1–6, 2005.

[13] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the Int'l Conference on Software Engineering*, pages 221–230, 2001.

[14] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of the Int'l Conference on Software Maintenance*, pages 34–43, 2002.

[15] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering java software systems. *Software Practice and Experience*, 31:371–394, 2001.

[16] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of java program. In *Proceedings of the Int'l Workshop on Principles of Software Evolution*, pages 148–151, 2005.

[17] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[18] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the Int'l Workshop on Dynamic Analysis*, pages 8–14, 2008.

[19] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, 1992.

Using a Sequence Alignment Algorithm to Identify Specific and Common Code from Execution Traces

Marcelo de A. Maia, Victor Sobreira, Klérisson R. Paixão, Sandra A. de Amo, Ilmério R. Silva

Computer Science Department
Federal University of Uberlândia
Uberlândia, MG, Brazil

{marcmaia,deamo,ilmerio}@facom.ufu.br, {victor.sobreira,klerissonpaixao}@gmail.com

Abstract

Software product lines are an important strategy to improve software reuse. However, the migration of a single product to a product line is a challenging task, even when considering only the reengineering task of the source code, not mentioning other management challenges. The reengineering challenges are partially due to effort of identifying common code of similar features, even when we know those features in advance. This work proposes the alignment of execution traces in order to discover similar code of similar features, facilitating the reengineering task. We present the architecture of our approach and preliminary results that shows a promising direction.

1 Introduction

Changes are inherent to software systems [4]. Every successful software goes through continuous evolution either to support new user expectations, hardware changes or operational changes. However, providing software evolution easily, quickly and correctly is still a major challenge for software engineers because applications are increasingly complex. This complexity is consequence of more sophisticated non-functional requirements. Most maintenance tasks are originated from new user requests, that is, perfective maintenance tasks [3, 4]. One of the major problems in software maintenance is related to program comprehension. The effort of comprehending of what will be modified is estimated in 40% to 60% of the whole effort of the maintenance phase[1]. This situation is aggravated when software documentation is either not updated, unintelligible, or simply does not exist. Another complicating issue is the software size. Reverse engineering techniques are being de-veloped with relative success, but their scalability to large systems is still a challenge.

This work proposes a reverse engineering technique using a sequence alignment algorithm. Sequence alignment algorithms have been applied in Molecular Biology to compare two or more sequences of DNA, RNA or protein in order to find out if there exists some similarity between them. For example, if we have two sequences: `ATGGATGCCC` and `ATGCATCCC`, a possible alignment would result in the following two sequences, respectively: `ATG-GATGCCC` and `ATGC-AT-CCC`. Note that gaps are introduced in the original sequences so that an *i-th* element of the first sequence can match the *i-th* element of the second sequence. The idea of this work is based on aligning similar execution traces in order to find out where the two traces match (common code) and where they mismatch (specific code). The technique is aided by a semi-automated tool to help the identification of specific and common code of similar features. The traces should be captured from similar execution scenarios of the system, otherwise it is not expected to find common code. It is important that the developer knows what are the commonalities and variabilities between two execution scenarios from an observational point of view in order to establish adequate traces for alignment.

2 The Approach

In Figure 1, a general view of our approach is presented using an UML activity diagram.

The first activity is to define suitable scenarios that enables extracting relevant information when comparing two executions traces. This is a manual activity, and information used as input for this activity comprehends new user requests that defines what kind of maintenance will be performed, similar features present in the system and the avail-
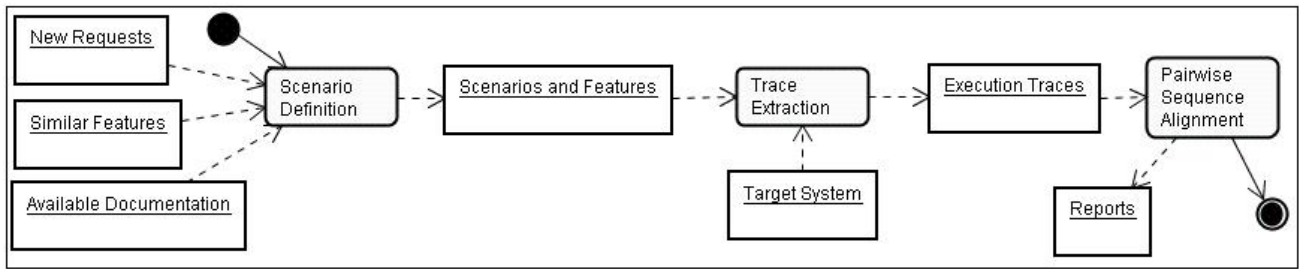
Figure 1. The proposed approach

able documentation of the system. The result of this activity is the definition of an execution scenario that must be performed, and consequently, the input data that enables the desired execution of the target system.

The trace extraction activity is automated. Our trace extractor is implemented in AspectJ. Currently, our target system must be implemented in Java. During the execution of the target system, the extractor intercepts method calls and writes a text file for each thread launched during the execution. Each line of the file corresponds to a method call whose content is the fully qualified name of the called method.

After the traces are collected, the next step is to perform an automatic pairwise alignment with two selected traces. The expected result of the alignment is the information of what is common to both sequences and what is specific to each sequence.

This information can be used to focus on the source code that is important to desired maintenance task.

3   Related Work

Some ideas of this work were inspired in other works in software maintenance.

Ding and Medvidovic proposes an incremental process for the evolution of object-oriented systems with poor or inexistent documentation. [6]. One phase of the process is the architecture recovery of specific fragments of the system. There are three assumptions for this phase: definition of the desired changes, knowledge of the application properties from the user point of view and the understanding of basic architectural features of the implementation plataform. This work was posteriorly revised with the addition of new heuristics for the phase of identifying components and with new case studies [9]. Our work relates to this, in the sense that the result of alignments helps to focus system understanding on the desired points of system evolution.

Rajlich and Silva have studied the reuse and the evolution of *orthogonal architectures*, which are code fragments organized in layers within the same abstraction level [11]. They have developed an application domain independent process aiming at adapting the system architecture to encompass a new requirement set. The authors have concluded that such process have application in small and medium-sized systems, and that the source code modularization was not effective for large scale systems. We expect that sequence alignment can be applied to large scale system in order to help focusing on the most important places to eventually modularize.

Sartipi et al. developed a work that comprehended in recovering system architecture using patterns defined in the AQL language - Architectural Query Language - and together with data mining techniques. The system is translated from source code to a graph model that is suitable for pattern-matching [13]. In other work[12], a framework that combines static and dynamic information is proposed. We also believe that we will need to combine the dynamic information extracted from sequence alignment and combine it with static information in order to achieve a more robust result.

Vasconcelos et al. [15, 16] presents a set of heuristics for class clustering in object-oriented systems from execution traces, using a similar idea of combining dynamic and static information.

Impact analysis techniques are responsible to identify the parts of the system that will be affected by a change. A well-known technique is program slicing. Binkley e Gallagher have presented a survey about this technique[5]. Our work can be used to provide the slicing criteria for understanding the impact of a software change.

Clustering is a data mining technique used for classifying related source code entities using similarity metrics. Wiggerts [17], Anquetil [2] and Tzerpos [14] shows different aspects on the clustering algorithms for source code. Feature location is a common task in software evolution activities. Marcus et al. has presented the application of an information retrieval method - *Latent Semantic Indexing (LSI)* that is used to map concepts written in natural language to relevant fragments of source code [8]. Our work also aims at locating source code fragments that are relevant in a soft-

ware evolution or software restructuring context based on external point of view of behavior.

In the Bioinformatics field, comparing sequences has became a major activity. The identification of similar regions in DNA, RNA or protein sequences can help mapping sequences to functional, structural and evolutionary characteristics. Several algorithms are presented in [7]. However, it is still a challenge to define and adapt sequence alignment algorithms for the software maintenance field. Surprisingly, at the best of our knowledge, we still could not find the use of alignment algorithms to detect similarities and commonalities of execution traces.

4 Sequence Alignment

Sequence alignment is a well-studied problem. Needleman and Wunsch have already proposed an algorithm for analyzing protein sequence in early seventies [10]. Several algorithms have been proposed since then. Indeed there are some issues that must be considered when adapting these algorithms for maintenance purposes.

4.1 Characteristics of Execution Traces

Our execution traces normally can present some patterns that can provide us with some information. For example, consider the sequences "XaaaaY" and "XaaaY". We can suspect that these sequence may be generated from the same code, and they are different just because the method "a" was called inside a loop that executed four times in one trace and three times in the other. Another example, consider the sequences "XaaaaY" and "XaabaY". In this case, we can suspect that some condition enabled the execution of the method "b", possibly inside a conditional command.

4.2 Global Alignment vs Local Alignment

Global alignments attempts to align every element in the sequences. These strategy is most useful when the sequences are similar and of roughly equal size. A general global alignment technique is the Needleman-Wunsch algorithm that is based on dynamic programming. Local alignments are more useful when we are trying to find a smaller sequence inside a larger one. The Smith-Waterman algorithm is a general local alignment algorithm and is also based on dynamic programming. There are also hybrid methods that attempt to find the best possible alignment that includes the start and end of one and the other sequence.

In this paper, we have chosen to study the alignment of almost similar sequences. Our goal was to choose similar features and to find out what is common and what is different between them. In such a situation, a global alignment strategy seems a reasonable alternative.

4.3 Pairwise Alignment vs Multiple Alignment

Pairwise alignment is used to find local or global alignments of two sequences. If it is necessary to compare several sequences, the alignment can only occurs with two sequences at a time, and the user should proceed with an integration step with another technique. Multiple sequence alignment is a generalization of pairwise alignment, in the sense that the alignment algorithm can take as input several sequences at a time. However, general multiple alignment algorithms tend to lead to NP-complete solutions, and thus are not very practical, unless you provide some heuristics or use a very small input.

In this paper, we have chosen to study the pairwise alignment because execution traces are normally large.

4.4 Identity and Similarity

In Bioinformatics, identity and similarity are related but different concepts. The identity is a relation of equality in which a nucleotide or aminoacid of one side must be equal to its complement to produce a match. This relation is too restrictive in Biology, so the alignment algorithm may consider to match two different elements, if these elements have some level of similarity.

In principle, considering that classes may have a reasonable cohesion, we could consider methods in the same class or in the same package to have some level of similarity, and thus apply the same principles of biology. However, in this work we have chosen to consider only the identity relationship as a prerequisite for matching two method calls.

4.5 Gap Penalty

Because in Bioinformatics is reasonable to accept the alignment match between two different elements, a question may arise when deciding if a match based on similarity is better or not than a gap that is inserted in one of the sequences.

In this work, we have decided not to penalize the introduction of gaps in either of the two sequences for two reasons. The first is that since we work only with identity, it seems incoeherent to accept an alignment match with two different elements instead of introducing the gap. The second reason is that the misalignment gives us also an important information: it may represent specific method calls of a sequence and thus contribute to identify specific code.

5 Application and Current Results

In this section, we present an application of sequence alignment to report specific and common code between two

Traces	Length - Prefix	Length - Specific to Rectangle	Length - Suffix
Rectangle	885 matches and gaps	396 specific to Rectangle	18 matches and gaps
Circle	885 matches and gaps	396 gaps only	18 matches and gaps

Figure 2. Length and Characteristic of Aligned Sequences

features in a small graphical editor shown in Figure 3. The total lines of code of the editor is 387, the number of classes is 9, and the total number of methods is 58.
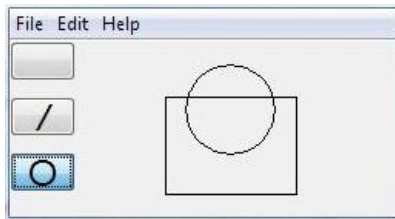

Figure 3. The target system

We have chosen two similar features to execute the system: drawing a rectangle and drawing a circle. The execution traces were collected and two threads were launched for each execution. Each pair of corresponding threads were aligned with a Needleman-Wunsch algorithm, considering only identities and zero gap penalty. Below we present the results of the alignments.

5.1 Results

The first thread was responsible for drawing the main frame and there was only 14 method calls perfectly aligned between each other.

The second thread was more interesting. Although the system is small and the execution scenarios are fairly simple, the thread for drawing a rectangle had 1040 method calls and the thread for drawing a circle had 644 method calls, and thus manual alignment seems unfairly hard. After the gap insertions each sequence had the gaps inserted and grown to 1299 elements.

In Figure 2, we show the tree main subparts of the traces and their correspondence. The interesting alignment is in the first and third part, summing 903 elements. After analyzing manually the traces, we could find out that the 396 elements in the second part, corresponds to gaps in the thread of drawing circle, because the size of the drawn rectangle was greater than the size of the circle and thus demanded more screen updates. In Figure 4, we summarize the quantitative details of the alignment.

After the alignment, we computed the set of common methods between the two features, the set of methods specific to the feature of drawing a rectangle and the set of

Before Alignment	
Length of Rectangle Trace	1040
Length of Circle Trace	644
Difference Rect-Circle	396
After Alignment	
Length of Rectangle Trace	1299
Length of Circle Trace	1299
#Matches	385
#Gaps in Rectangle Trace	259
#Gaps in Circle Trace	655
#Real Gaps in Circle Trace	259
Length of Interesting Alignment	903
%Matches	0.4263
%Interesting Gaps in Rectangle	0.2868
%Interesting Gaps in Circle	0.2868

Figure 4. Quantitative results

methods specific to the feature of drawing a circle. The results are shown below, respectively. False positives have arisen when finding methods specific to draw a rectangle. The reason was that it was not possible to align those 396 method calls with a counterpart in the draw circle feature, as already shown in Figure 2. Nonetheless, the other results seem promising because no false negative has arisen and all called methods were present in at least one of the above three sets.

```
// Common methods
graphicaleditor.MainFrame$1.paint
graphicaleditor.MainFrame.access$0
graphicaleditor.ShapeSet.draw
graphicaleditor.MainFrame.access$1
graphicaleditor.MainFrame.processWindowEvent
graphicaleditor.MainFrame$4.mousePressed
graphicaleditor.MainFrame.drawPanel_mousePressed
graphicaleditor.MainFrame.createShape
graphicaleditor.Point2D.<init>
graphicaleditor.MainFrame$5.mouseDragged
graphicaleditor.MainFrame.drawPanel_mouseDragged
graphicaleditor.Point2D.getX
graphicaleditor.Point2D.getY
graphicaleditor.MainFrame$4.mouseReleased
graphicaleditor.ShapeSet.add
graphicaleditor.MainFrame.drawPanel_mouseReleased
graphicaleditor.MainFrame.jMenuFileExit_actionPerformed

// Methods specific to draw rectangle
graphicaleditor.MainFrame$1.paint
graphicaleditor.MainFrame.access$0
graphicaleditor.ShapeSet.draw
graphicaleditor.MainFrame.access$1
graphicaleditor.Rectangle.<init>
graphicaleditor.Rectangle.setAnchor
graphicaleditor.MainFrame$5.mouseDragged
graphicaleditor.MainFrame.drawPanel_mouseDragged
graphicaleditor.Rectangle.getAnchorX
graphicaleditor.Point2D.getX
graphicaleditor.Point2D.getY
graphicaleditor.Rectangle.getAnchorY
graphicaleditor.Rectangle.draw
graphicaleditor.Rectangle.setDimension
graphicaleditor.Rectangle.getAnchor
```

```
// Methods specific to draw circle
graphicaleditor.Circle.<init>
graphicaleditor.Circle.getAnchorX
graphicaleditor.Circle.setAnchor
graphicaleditor.Circle.setDimension
graphicaleditor.Circle.getAnchorY
graphicaleditor.Circle.getAnchor
graphicaleditor.Circle.draw
```

6 Final Remarks

In this work, we have shown an approach to identify commonalities and variabilities in execution traces. The possibilities of usage of these information are manifold. We can help the introduction new features in the target software based on similar characteristics already present providing information of specific methods that the feature must implement. We can help extracting common components from source code based on information provied by commonalities between execution traces.

There are many questions that still persist, for instance, how the approach will scale up for larger systems, how the extracted information can be more systematically used by developers, how would be the results when working with different versions of the system, how much the trace compression would enhance the approach, and how different alignment methods behave in different situations.

References

[1] A. Abran, P. Bourque, R. Dupuis, and L. Tripp. Guide to the software engineering body of knowledge (ironman version). TR, IEEE Computer Society, 2004.

[2] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, Washington, DC, 1999.

[3] K .Bennett. Software evolution: past, present and future. *Information and Software Technology*, Vol. 38(11):673–680, November 1996.

[4] K .Bennett and V .Rajlich. Software maintenance and evolution: a roadmap. In *Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[5] D. Binkley and K .Gallagher. Program slicing. *Advences in Computer*, 1(43), July 1996.

[6] L .Ding and N. Medvidovic. Focus: a light-weight, incremental approach to software architecture recovery and evolution. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 191–200, 28-31 Aug. 2001.

[7] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, January 1997.

[8] A .Marcus, A .Sergeyev, V .Rajlich, and J .Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proc. of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[9] N .Medvidovic and V .Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[10] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3), 1970.

[11] V. Rajlich and J. Silva. Evolution and reuse of orthogonal architecture. *IEEE Transaction on Software Engineering*, 22(2):153–157, 1996.

[12] K. Sartipi, N. Dezhkam, and H. Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *Proc. of 13th Work. Conf. on Reverse Engineering*, pages 61–70, Oct. 2006.

[13] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Proc. of 4th European Conf. on Soft. Maintenance and Reengineering*, pages 129–139, March 2000.

[14] V .Tzerpos and R. Holt. Software botryology: Automatic clustering of software systems. In *International Workshop on Large-Scale Software Composition*, pages 811–818, 1998.

[15] A. Vasconcelos, R. Cêpeda, and C. Werner. An approach to program comprehension through reverse engineering of complementary software views. In *1st Intl. Workshop on Prog. Comprehension through Dynamic Analysis (PCODA)*, pages 58–62, 2005.

[16] A. Vasconcelos and C. Werner. Software architecture recovery based on dynamic analysis. In *Simpósio Brasileiro de Engenharia de Software*, 2004.

[17] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proc. of the 4th Working Conf. on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, 1997. IEEE Computer Society.

Behavioral Design Pattern Detection through Dynamic Analysis

Francesca Arcelli, Fabrizio Perin, Claudia Raibulet, Stefano Ravani
DISCo – Dipartimento di Informatica Sistemistica e Comunicazione,
Università degli Studi di Milano-Bicocca
{arcelli, raibulet}@disco.unimib.it, {fabrizio.perin, stefano.ravani}@gmail.com

Abstract

The recognition of design patterns in an existing system provides additional information related to the rationale behind the design of the system which is very important for the system comprehension and re-documentation. Several approaches and tools have been proposed for design patterns detection, some approaches are based only on static analysis of the code, other use both static and dynamic analysis.

In this paper we present our approach to the recognition of design patterns based on dynamic analysis of Java software. The idea behind our solution is to identify a set of rules capturing information necessary to identify a design pattern instance. Rules are characterized by weights indicating their importance in the detection of a specific design pattern. The core behavior of each design pattern may be described through a subset of these rules forming a macrorule, which defines the main traits of a pattern. JADEPT (JAva DEsign Pattern de Tector) is our prototype for design pattern identification based on this idea. It captures static and dynamic aspects through a dynamic analysis of software by exploiting JPDA (Java Platform Debugger Architecture).

1. Introduction

The information related to the presence of design patterns [3] in a system is useful not only to better comprehend the system, but also to discover the rationale of its design. This has a significant implication for further improvement or adaptive changes, with various advantages for the overall maintenance process.

In the context of design patterns detection, it is possible to use different approaches both for the identification logic (e.g., searching for subcomponents of design patterns, identifying the entire structure of a design pattern at once) and for the information extraction method (e.g., static, dynamic, or both).

Problems raised by the identification of design patterns are related not only to the search aspects, but also to the design and development choices. There are at least three important decisions that should be taken when developing a design pattern detection tool. These decisions may influence significantly the final results. The first issue regards the evaluation of how to extract the interesting data from the examined software, including the type of the analysis to be performed. The second issue considers the data structure in which to store the gathered information; one important risk is related to the loss of knowledge at the data or the semantic level: this would generate inferences about something that is no more the analyzed software, but an incorrect abstraction of it. The third one highlights the importance to find a way to process the extracted data and to identify design pattern instances. Independently of the adopted data structure for the extracted information (e.g., a text file, XML, database), the following three aspects should be considered: memory occupation, processing rate and, most important, the effective recognition process of design patterns with a minimum rate of false positives and false negatives. While the first two issues could be solved through an upgrade of the machine on which elaboration is performed, the last is strictly related to the efficiency of the recognition logic applied for design pattern detection due to the significant number of possible implementation variants.

In this paper, we present a new approach based on the analysis of dynamic information caught during the execution of the software under analysis in order to detect behavioral design patterns. We consider behavioral design patterns because they are particularly appropriate for dynamic analysis. In fact, their traces may be better revealed at runtime by analyzing all the dynamic aspects including: object instantiation, accessed/modified fields, and method calls flows. The main advantage of using dynamic analysis regards the fact that it is possible to evaluate both the structure of a

design pattern and its behavior. Thus, it is possible to assert that a piece of software represents not only the structure of a design pattern, but its behavior, too. Through dynamic analysis it is possible to observe objects, their creation and execution during their entire life-cycle and overcome part of the limitations of the static analysis which may be determinant in pattern recognition.

JADEPT (JAva Design Pattern deTector) is the software prototype we are developing for design pattern detection which collects structural and behavioral information through dynamic analysis of Java software by exploiting JPDA (Java Platform Debugger Architecture). We validated our approach on canonical design pattern implementations and on the JADEPT itself.

There are various approaches that aim to detect design patterns based on a static analysis of source code such as: FUJABA RE [6], SPQR [12], PINOT [11], PTIDEJ [4] or MAISA [14]. The design pattern detection mechanisms search for information defining an entire pattern or sub-elements of patterns which can be combined to build patterns [1] or evaluate the similarity of the code structure with higher-level models as UML diagrams [2] or graph representations [13]. Other approaches exploit both static and dynamic analysis as in [5, 9] or only dynamic analysis as in [10, 15]. The approach we use in JADEPT is different from the other solutions, but a comparison with the other tools is not possible, since a real benchmark is not yet available.

The paper is organized through the following sections. Section 2 presents the identification rules and an example of their application on a behavioral design pattern. Section 3 describes the overall architecture and the functionalities of the JADEPT prototype. Section 4 introduces several aspects concerning the validation of JADEPT. Conclusions and further work are dealt within Section 5.

2. JADEPT

To detect design pattern in JADEPT we defined a set of rules describing the properties of each design pattern. Properties may be either structural or behavioral and may describe relationships between classes or families of classes. We defined a family of classes as a group of classes implementing the same interface or extending a common class. Weights have been associated to rules indicating how much a rule is able to describe a specific property of a given design pattern. The rules have been defined independently of any programming language.

Nevertheless part of the extracted information can be obtained by a static analysis of the software, JADEPT extracts all the information during the execution of the software adopting an approach based exclusively on dynamic analysis. The extracted information is stored in a database. The advantages of having information stored in database are: (1) the possibility to perform statistics and (2) the possibility to memorize information about various executions of the same software. A rule may be implemented by one or more queries to the database. The database has been designed to model concepts of a generic object-oriented language. The presence of a design pattern is verified through the validation of its associated rules.

2.1. Rules for design pattern detection

We present how rules, weights and macrorules have been defined for the detection of design patterns, and we introduce them through an example for the Chain of Responsibility pattern (see Table 1). First, the identification rules are written using natural language (see Table 1 – Second column). This approach avoids introducing constraints regarding the implementation of rules. In JADEPT, rules are translated into queries, but they can be used also outside the context of our tool and hence, represented through a different paradigm (e.g., graphs).

Then weights have been added to the rules to determine the probability of the pattern presence in the examined code, weights denote the importance of a rule in the detection process of a pattern (see Table 1 – Third column). Weights' range is 1 to 5. A low weight value denotes a rule that describes a generic characteristic of a pattern like the existence of a reference or a method with a specific signature. A high weight value denotes a rule that describes a specific characteristic of a pattern like a particular method call chain that links two class families. Even if each behavioral design pattern has its own particular properties, an absolute scale for the weights value has been defined based on our design pattern detection experience which can be obviously further improved or modified. Rules whose weight value is equal to 1 or 2 describe structural and generic aspects of code (e.g., abstract class inheritance, interface implementation or the presence of particular class fields). Rules whose weight value is equal to 3 or higher, describe a specific static or dynamic property of a pattern. For example, the fifth rule of Chain of Responsibility in Table 1, specifics that each call to the *handle()* method has always the same caller-callee objects pair. This is the way objects are linked in the chain. A weight whose value is equal to 5 describes a native implementation of the design pattern we are considering.

The n ext step reg ards the d efinition o f th e relationship between rules [8]. There a re two types of relationships. The first one is *logical*: if the ch eck of a rule does no t have a po sitive v alue, it do es no t m ake sense to proof the rules related to it. Fo r example, the fifth rule in Table 1 cannot be proved if the fourth rule has not bee n proved fi rst. The second o ne is *informative*: if a r ule depends on another one, and the latter is v erified b y th e soft ware detector, its weig ht increases. T he seco nd t ype of relationship det ermines those rules which are stro nger for th e identification of design patterns.

Finally we have i ntroduced m acrorules. A macrorule is a set of rule s which desc ribes a specific behavior o f a pat tern w hich t hey refe r t o (see a n example in Table 2). If t he rules t hat c ompose a macrorule are verified, t he c ore behavior of a pat tern has been detected so th e fin al probability v alue increases. The v alue ad ded to th e probability is different for each pattern. T his is because the num ber of rules which belongs to a macrorule varies from one macrorule to another.

A question mark a fter a weight val ue indicates a variable weig ht. F or e xample, the fifth rule has a variable weight because of its relation with rule number four. If the fourth is verified then the weight of the fifth rule i s increase d by one, hence as sociating a higher probability to the pattern instance recognition.

The fourth column indicates the type of inform ation needed to ve rify a rule. If a rule desc ribes a static property, which can be ve rified through an analysis of static in formation, th en th e value i n th is co lumn is S (indicating *static*). If a r ule descri bes a dy namic property, which can be ve rified through an analysis of dynamic in formation, th en the value in this column is D (indicating *dynamic*).

In t he case we have t o verify a p roperty by performing analysis of static and dynamic information, then th e value sp ecified is S-D (indicating static and dynamic). Ho wever, i n J ADEPT both st atic and dynamic information are ex tracted through a dynam ic analysis of the software under inspection.

A rel ationship of dependency am ong t wo or m ore rules is i ndicated in t he fifth co lumn. A log ical dependency i s bet ween r ule fo ur a nd five. R ule fi ve cannot be proved if rule four is not previously verified. The informative dependency we have defined for this pattern i nvolves th e fou rth an d th e fifth rules. Ru le 5 can increment by one its weight if rule 4 is verified.

Table 2. T he Macr orule for the Chain of Responsibility Pattern

Macrorule	Rules
Sequential redirection	4, 5

3. JADEPT architecture

JADEPT i s a Java a pplication c omposed of four main modules [7] : Graphic Use r I nterface (G UI), Launcher and Capture Module (LCM), D esign Pattern Detector Module (DPDM) and JDEC Interface Module (see Figure 1).

3.1. Graphic User Interface

JADEPT's GUI allows users: (1) to set u p a JADEPT XML con figuration file, (2) to launch th e software t o be monitored, (3) to start t he an alysis on the sto red info rmation an d (4) t o create the JDEC database.

Table 1. Detection rules for the Chain of Responsibility design pattern

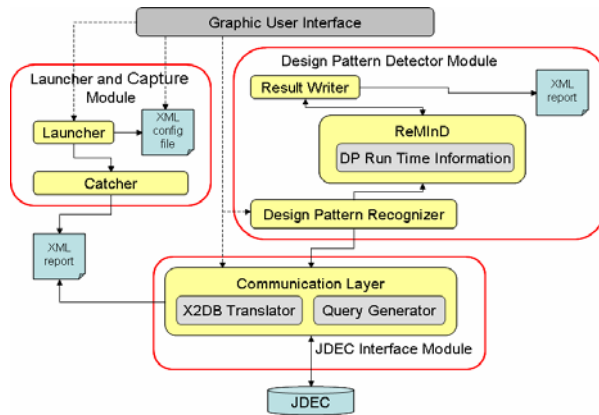| Nr.	Rule	Weight/ Specificity	Type	Dependencies
1	Some classes implement the same interface.	1	S	
2	Same classes extend the same class.	1	S	
3	All classes that implement the same interface or extend the same class, contain a ref erence wh ose t ype is th e s ame of th e implemented interface or the extended class.	3	S	
4	Each class has one method that contains a call to the same method in another class of the same family and this method must contain a parameter.	3	S-D	
5	"*handle*" is defines as the name of the method identif ied by the forth rule. The call to *handle* method of one object is always originated by the same caller object. This property is true for each object of the family.	3?	D	if 4 = +1

Figure 1. JADEPT architecture

The set up of the configuration file is obviously the first op eration to be p erformed to start a n ew work session a nd it is also nece ssary to creat e a JDEC database instance to store t he extracted information if this operation has ne ver been performed before. After these operations, JADEPT is ab le to run th e user application and to launch the design pattern detector on the collected information.

JADEPT GUI has bee n desi gned using t he Command pa ttern [3] . In t his way bot h t he management of t he o perations of al l gra phical components and t he st ructure o f t he e ntire G UI becomes simpler.

3.2. Capture and Launcher Modules

The Capture and Launcher Module is composed of two following modules:

- the *Launcher Module* which starts the execution of the software under analysis, as well as the execution of the Catch er Module. Th e Launcher uses th e XML configuration file to read all the information needed to run the user application. When the Launcher is invoked from the GUI a new Java Virtual M achine (J VM) is created and configured to e xecute the u ser application in the debug mode. Finally, the Catcher is invoked.

- the *Catcher Module* uses JPDA to capture events occurred i n the J VM cre ated by t he Launc her. Essentially, events re gard classes and interfaces loading, method calls, field accesses and modifications. Through these events J ADEPT extracts various types of information exploited in the detection process (e.g., the loaded classes provide information about their field names and types or about their methods). Using JPDA the ex traction o f inform ation is sim plified bu t th e monitored software pays in terms of performance. This performance red uction i s caused by t he JVM suspension needed to read the information contained in the JVM stack when a method call event occurs.

At the end of user's software execution the Catcher Module writes the XML Report File containing all th e collected information and i nvokes the Communication Layer insertion method. Thus, the XML Report File is inserted in the JDEC d atabase. Th e XML Report File will n ot b e deleted at th e end of th e in sertion, i n th is way it is pos sible to keep traces of the softwa re executions independently of the database.

3.3. JDEC and its Interface Module

To store the extracted information we use the JDEC database. In th is way, info rmation is av ailable to the Design Pattern Detecto r Modu le (DPDM). The JDEC structure models both the object oriented code structure and its behavior (e.g., the class fields and methods, the method calls and i ts containing operations). The JDEC structure can be divided in two main parts: the first one is com posed of t hose relations w hich c ontain st atic information (classes, in terfaces, field s, meth ods an d their argum ents); the second one c ontains dynam ic information as method calls, accessed fields or modified and in stantiated obj ects. For e xample, this type of i nformation al lows the reco gnition of different design patterns ha ving ve ry sim ilar st ructures, but different beha viors suc h as t he St ate and St rategy design patterns.

4. Validation

JADEPT has been val idated using di fferent implementation sam ples of desi gn pat terns m ore o r less closer to their definitions given in [3]. The results for th e Ch ain o f Resp onsibility d esign pattern are shown in Table 3.

The fi rst colu mn o f th e tab le co ntains th e identification n ame fo r th e i mplementations considered. Th e remaining columns show th e results provided by the Chain of Responsibility, Observer and Visitor detectors. T he l ast t wo detectors have been used to v erify if th ey provide false negatives. Th e `-' symbol means that JADE PT has not de tected any instance f or a gi ven design pat tern. T he `X' sym bol indicates that t he c onsidered sample does not provide any implementation of a specific pattern.

JADEPT recognizes t he Chain of Respon sibility pattern i n three i mplementations with reliable v alues. The Chain implementation in fluffycat is det ected as a false negative because JADE PT is not able to find a good *handle()* cand idate in t his p attern i nstance. Th is argument indicates the request that should be managed by one of the classes whic h implements the interface. Moreover, ea ch class im plementing the interface declares a fiel d whose type is the type of t he common

interface. T he success or el ement in the chain is assigned to this field during execution.

Table 3. Chain of Re sponsibility implementati on analyzed by three design pattern detectors

CoR	CoR Detector	Observer Detector	Visitor Detector
composite	X	X	x
composite3	X	X	X
cooper	100%	10%	17%
earthlink	76%	-	-
earthlink2	X	X	X
fluffycat	7%	-	-
kuchana	69%	-	-
sun	X	X	X
vis1	X	X	X
visitorcontact	X	X	X

Figure 2 shows th e class d iagram relate d to th e implementation o f th e C hain o f Resp onsibility in th e *fluffycat* example. According to th e Go F's d efinition, this pattern s hould define a comm on interface (e.g., called *Chain*) which is implemented further by two or more classes. The interface d efines a meth od (e.g., called *sendToChain(String)*) which acce pts only one argument.
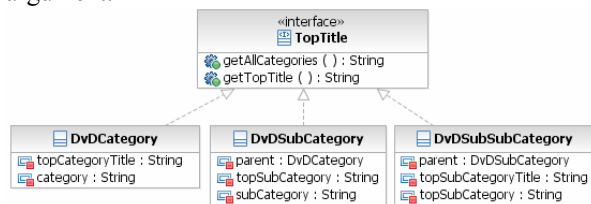

Figure 2. The Chain of Responsibility pattern in the fluffycat implementation example

The fluffycat implementation is n ot clo sed to th e GOF's definition: it define s a comm on interface called *TopTitle*, but this interface declares m ethods which accept no arguments. This aspect does not comply with GoF definition an d it is reflected in th e lo w v alues associated to the fluffycal implementation in Table 3.

Table 4 shows the results of JADEPT that analyzes itself. JADEPT is com posed of 151 classe s. Analysis reveals th e presen ce of Ch ain o f Respon sibility an d Observer, which are actually implemented in the c ode. In JADEPT th ere are no Visitor in stances, and th is analysis was performed o nly to test if an y false positives are revealed.

Table 4. JADEPT analyzed by JADEPT

System Name	CoR	Observer	Visitor
JADEPT	100%	90%	17%

To summ arize, there ar e two m ain reasons why JADEPT ca nnot perform anal ysis o n s ome implementations. Th e first is related to t he q uality o f implementations them selves because the y are very different f rom t he UML st ructure of pat terns d efined by GoF. For example, cla sses do not im plement th e same interface or extend the same class. We mean that such im plementations ca nnot be ret ained as val id instances of design patterns. Common interfaces and classes are used to easily ex tend software and their use is a pri nciple of good p rogramming as m uch as ot her design pattern features.

The sec ond problem con cerns the i nformation partitioning technique of JADEPT. Ou r tool can work on families retrieved from the information collected in JDEC. Before starting the analysis, JADEPT id entifies all th e p ossible families an d assig ns to each fam ily a specific role, according to the design pa ttern it is looking for. If th e an alyzed system is u nstructured, meaning that c ommon interfaces or classes are absent, JADEPT ca nnot b uild c orrectly the families and perform further analysis.

5. Conclusions and future work

In t his pa per we ha ve pre sented o ur a pproach t o detect desi gn pat terns i n J ava a pplications t hrough dynamic analysis to extract all the inform ation needed in the detection process. The defined rules focus on the behavior of the patterns and not on their static aspects. Rules capturing static p roperties have been introduced because they express pre- conditions for the dynam ic ones. Further we have defined logical and informative dependencies am ong rul es, established the importance of rules i n t he det ection process t hrough s cores, an d identified a group of rules characterizing the p articular behavior of each pattern through macrorules.

We ha ve v alidated ou r i dea t hrough t he implementation o f t he JADEPT p rototype. JA DEPT has been developed for Ja va software, while the rul es and the database are l anguage-independent. Thus, it is possible t o a pply and re use t hese co ncepts i n ot her object-oriented l anguages. Modularity is o ne of t he main characteristic of the JADEPT architectural model. It may use alternative ways to extract information or to perform analysis. It is possibl e to exclude the database and t o use an other ap proach t o det ect desi gn patterns due to existence of the XML Report file. Moreover, the database m odel can be used in an other de sign pattern detector or a software architecture reconstruction tool.

The decision to use a database to store the extracted information is d ue t o two main reason s. Th e first is related to t he larg e am ount o f i nformation wh ich should be ex tracted during s oftware ex ecution an d

which should be considered to identify design patterns. The second is related to the traceability/persistence in time of the extracted i nformation, the comparison among two or more executions of the software code or among e xecutions o f different a pplications, an d t he statistics which may be done. The issues related to this second as pect are not im plemented in the curre nt version of our prototype.

We h ave outlined i n th e p aper th e adv antages o f using dy namic analysis for desi gn pat tern detection, but we have also identified two limitations. The first is related to a possible reduction of the perform ance of the analyzed a pplication. T o improve its pe rformance we ha ve use d a filtering syste m to trace only the meaningful e vents. T he execution ti me o f th e monitored applications is still lo nger than the ordinary execution time, es pecially for software ha ving a Graphic User Interface. The second, concerns the code coverage proble m. If the analyzed soft ware needs a user interaction, it cou ld be necessary a human-drive n selection o f code functions t o rev eal al l possi ble behaviors.

Future wo rk will reg ard th e validation o f JADEPT on sy stems of l arger dimensions. M oreover, we are working on t he definition of a b enchmark for t he evaluation of desi gn pat tern detection tools based o n various criteria. Th is will allo w us to compare various tools e xploiting va rious a pproaches based on static, dynamic or h ybrid a nalysis. Fu rthermore, JA DEPT will be extended to detect also creational and structural design patterns.

The rules may be re vised in terms of de finition and the scores ass ociated to t hem based on the experie nce gathered during sy stems validations. F urther, t he translation of the ru les into the design pattern detector module in term of queries and programming logic may be optimized.

6. References

[1] F. Arcelli, S. Masiero, C . Raibulet, and F. Tisato , " A Comparison of Reverse Engine ering Tools based on Design Pattern Decomposition", *Proceedings of the IEEE Australian Software Engineering Conference*, 2005, pp. 262-269

[2] F. Bergenti and A. P oggi, "Improving UML Designs Using Automatic Design Pattern Detec tion", *Proceedings of the 12 th I nternational Con ference on So ftware Engineering and Knowledge Engineering*, 2000, pp.336-343

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: e lements of re usable object-oriented software* , Addison Wesley, Reading MA, USA, 1994

[4] Y. G. Gué héneuc, "PTIDE J: Prom oting Patterns with Patterns", *Proceedings of the 1 st ECOOP Workshop on Building Systems using Patterns,* Springer-Verlag, 2005

[5] D. Heuzeroth, T. Holl, an d W. Löwe, "Combining Static and D ynamic Analyses to D etect In teraction P atterns", *Proceedings the 6th World Conference on Integrated Design and Process Technology*, 2002

[6] U. Nickel, J. Nier e, A. Zündorf, "The FUJABA Environment", *Proceedings of th e 22 nd Internationa l Conference on Software Engineering*, 2000, pp. 742-745

[7] F. Pe rin, *Dynamic analysis to det ect the design patterns in Ja va: gath ering information with JPDA.* M Sc Thesis, University of Milano-Bicocca, Milan, April 2007

[8] S. Ravani, *Dynamic analysis for Design Pattern detecting on Java code: information r elationship modelling* , M Sc Thesis, University of Milano-Bicocca, Milan, April 2007

[9] N. P ettersson, " Measuring P recision for S tatic and Dynamic Desig n Pattern Reco gnition as a Function of Coverage", *Proceedings of th e Workshop on Dynamic Analysis, ACM SIGSOFT Software Engine ering Notes, Vo l. 30, No. 4,* 2005, pp. 1-7

[10] D. M. Shawky, S. K. Abd-El-Hafiz, an d A.-L. El-Sedeek, "A D ynamic Approach for the Id entification of Object-oriented Des ign P atterns", *Proceedings of the IASTED Confer ence on Softwar e Engin eering,* pp. 138-143, 2005

[11] N. Shi, and R. A. Olsson , "Reverse Eng ineering of Design Patterns from Java Sourc e Code", *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 123-134

[12] J. McC. Sm ith, and D. Stotts, " SPQR: Flexible Automated Design Pattern Extr action From Source Code" , *Proceedings of the I EEE In ternational Co nference on Automated So ftware Engineering,* Octob er, 20 03, pp. 215 - 224

[13] N. Ts antalis, A. Ch atzigeorgiou, G. Step hanides, G., and S. T. Ha lkidis, " Design Patte rn Det ection Using Similarity Sc oring", *IEEE Transactions on Software Engineering*, Vol. 32, No. 11, November 2006, pp. 896-909

[14] A. I. Verkamo, J. Gust afsson, L. Neno nen, and J. Paakki, "Design patterns in perform ance predi ction", *Proceedings of t he ACM Second International W orkshop on Software and Performance*, 2000, pp. 143-144.

[15] L. Wendehals, "Improving Design Pattern Instance Recognition by Dynamic Analysis", *Proceedings of the ICSE Workshop on Dynamic Ana lysis*, Portland , USA, 2003, pp . 29-32.

TAG (TrAce+Grep): a Simple Feature Location Approach

Dapeng Liu, Monica Brockmeyer
Department of Computer Science
Wayne State University
[dliu, mbrockmeyer]@wayne.edu

Shaochun Xu
Department of Computer Science
Algoma University
simon.xu@algomau.ca

Abstract

Orthogonality of LSI query results and traces makes hybrid feature location approaches, such as PROMISIR and SITIR, very useful. In this paper we propose a new feature location approach, TAG (TrAce+Grep), which takes advantages of the orthogonality, by exchanging the order of static and dynamic components of SITIR and replacing LSI with GREP. We re-conducted the case studies that validated SITIR using TAG and compared performances of the two approaches. The analysis of various observations and deep discussion on the cross cut of orthogonal information are also presented thereby. We conclude that simple feature location strategies, such as TAG, can compete with complex ones in term of effectiveness, even for big software.

1. Introduction

Software change is unavoidable since it is impractical to develop the program in one round without modification; especially with the increasing volume and complexity and elongated development of current software. Feature location [1], the first step of software change, is to locate parts of code that are related to a specific feature that is extracted from the change request.

In application of feature location, hybrid approaches that combines tracing and LSI, such as Probabilistic Ranking Of Methods based on Execution Scenarios and Information Retrieval (PROMESIR) and SIngle Trace and Information Retrieval (SITIR) [2], have been proved to be helpful. We accredit their success to the crosscut of search result and traces as their noises are orthogonal.

Based on our observation that in practice programmers prefer simple words to lengthy sentences when using LSI, we propose a new feature location approach TrAce+Grep (TAG), which simplifies SITIR by exchanging the order of static and dynamic approaches and replacing LSI with GREP.

We also re-conduct case studies of [2] using TAG and found that, in the best cases which nonetheless are reasonably easy to achieve, TAG worked comparably well as SITIR, especially with consideration of its much less time overhead. Our case studies demonstrated that during feature location simple approaches can achieve promising precision and speed.

The rest of the paper is organized as follows: section 2 discusses the orthogonality of search result and traces; section 3 presents the case studies and analyzes the results; section 4 enumerates related work; finally section 5 concludes the paper and presents future work.

2. Cross cut of static and dynamic information

Hybrid feature location approaches that combines tracing and LSI have been proved to be helpful, and the representatives are SITIR and PROMISIR. We accredit the success of combining tracing and LSI to the crosscut of search results and traces as their noises are orthogonal. While noise in trace comes along time axis, noise in LSI comes from semantic similarity. In one imaginary but common case, as shown in Figure 1, we can see the orthogonality of the search result and traces. When we use LSI to search for "popup menu", many methods that are related to the concept will be retrieved, such as "organize", "internalize", "show", "hide", etc. Please note that there may be some overlap of different query results. For example, "divide menu items into groups" might be shared by "show" and "organize". When we trace a program, prior and subsequent activities are hard to be completely avoided, even we can indicate when to start or stop tracing, partially due to the speed of program execution. With close observation, we can see that the result of using LSI contains only a few methods that are used to preparing for menu display; and the result of using traces do not have methods that are related to other operations of popup menu, such as hiding. Cross cut of the two sets will make the final

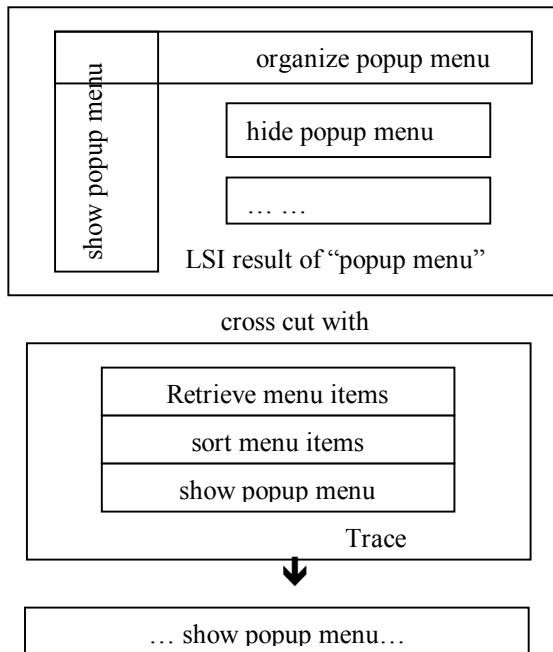result much smaller thus precision will be improved remarkably.

show popup menu	organize popup menu
	hide popup menu
	… …

LSI result of "popup menu"

cross cut with

Retrieve menu items
sort menu items
show popup menu

Trace

⬇

… show popup menu…

Figure 1. Orthogonal LSI results and Traces

Although LSI can provide programmers the convenience of using fuzzy query, it, to some extent, weakens the performance brought by the orthogonality, as the resulted ranked list contains all unrelated methods. Moreover, in practice, we noticed that programmers prefer simple words to lengthy query sentences that construct vectors in semantic space and thus lose the advantage of LSI. There is no need to mention the lengthy time needed for LSI to parse code corpus. Therefore, we replace LSI with GREP for simplicity and increasing performance. Here, we take the risk of loosing recall in favor of performance.

Our new method is named TrAce+Grep (TAG). In summary, its differences from SITIR are: tracing is done first and LSI is replaced with GREP.

3. Case studies

To verify our conjectures and to evaluate the performance of TAG, we have re-conducted the six case studies that were done for SITIR [2] and compared the results of the two different approaches. Following SITIR, we apply TAG on full traces as well and demonstrate the advantages of as-needed tracing. Finally, based on observations on those case studies, we discuss about cross cut of static and dynamic information to deeper extent.

TAG and SITIR cannot be compared directly. While formats of result sets are different btween TAG and SITIR, we define their performance measurement formulas respectively. For SITIR, the performance of feature location is defined as the highest rank of relevant methods; we consider a method relevant to the change request if it will be modified in response to it. For TAG, the performance is defined as half size of the result set, since in the result methods are not ordered.

When TAG is able to dig out sought methods, its result is labeled as Successful; otherwise, if viewers believed the result contained related information that was close to the sough methods, TAG was labeled as Potential; in the rest cases, TAG was labeled as Failed. This new classification was based on the consideration that feature location is used for programmers to identify methods that are related to specific features. Even though no method in the result is directly related to the sought feature, the result can still be helpful. To constrain subjectivity of judging TAG as Potential, programmers must be able to present the reasoning in one sentence when deciding to do this. Overlong reasoning would not be honored.

Difference of working principles requires distinct reasoning when making queries for TAG an SITIR. In the case studies of SITIR, new words were added to queries to refine them. Whereas, adding unrelated words to GREP query will purge all results, therefore, TAG requires more precise queries and is very sensitive to them. To compare performance of TAG and SITIR, we will refer to queries used in SITIR for TAG; however, we have to change the way in which we use the query tokens; more important, in some cases TAG was not used in their best favor.

The granularity of our traces is at method level. During the search process of TAG, only method names are used, which contain much less information than in SITIR, in which method body is fully parsed. With the expectation to analyze how much the loss of information affects performance of TAG, we construct complementary method names by concatenating package name, class name, and method name together. For each feature location assignment, we used GREP twice: once on complementary method names and the other time we used original method names. The expected benefit can be explained in the following imaginary but realistic example: both MessageBox and SearchDialog classes have methods named as OkButtonPressed, however, the two methods have totally different meanings; one to complete an action, the other to start a new action.

3.1 Case study setup

We chose the same software applications, JEdit 4.2 and Eclipse 2.1.3, that were used in [2] for case studies using TAG. JEdit consists of approximately 500 classes and about 5,000 methods with 88,000 lines of Java source code and internal comments, Swing was used for its GUI components and user event transference. Eclipse is mostly written in Java, with some C/C++ code used mainly for the widget toolkit, which we did not analyze here We used version 2.1.3, which contains approximately 7,000 classes and about 89,000 methods in more than 8,000 source files implemented in nearly 2.4 MLOC. Eclipse uses its own GUI package named JFace and transfers all user events by itself. We chose them with expectation to show the scalability of TAG, to allow replication of our case studies, and to make our observations more representative of reality.

Case study assignments were borrowed from [2] without modification; and we still use the same labels for them. Constrained by limited space, readers are referred to the previous paper for details.

3.2 Results and observations

We summarize all case study observations into Table 1. In six title rows, there are three substrings separated by commas, the first ones label the case study, the second ones indicate the performance of SITIR on the same case study, the third ones are the sentences that were used in SITIR.

Except for title rows, the rest of the table is partitioned in seven columns. The first columns list query tokens; columns 2~4 display results returned on complementary methods names; and the rest columns display results working on original method names.

The second columns, titled as #FM/#C, list two values that are separated by a back slash: number of found methods and number of the containing classes, including inner and anonymous ones. The third columns present whether TAG identified sought methods: S means Successful, P means Potential, and F means Failed. The forth columns show the results using full traces, i.e. the object programs were traced from startup to termination.

Columns 5~7 present in the same way as columns 2~4, except that methods were presented in their original names.

To create as-needed trace, we traced object program once for each case study. We can see full traces may

enlarge the result by 13 times, such as in case study JEdit #2 using "add" on both full and original method names. However, we noticed that, if we choose the best queries, though full traces still enlarge results in most cases, many times the result sets are still very small.

Table 1. Summary of case studies

GREP token	#FM/ #Class	S?	TAG using ft	#M	S?	ft&m
JEdit #1, 9, "search find next"						
find	2/2	S	12/3	2	S	12
search	65/8	P	58/12	6	P	11
next	2/2	P	15/8	2	P	15
JEdit #2, 1, "add marker"						
addMarker	2/2	S	2/2	2	S	2
add	5/4	S	65/38	5	S	65
marker	14/6	S	42/12	8	S	18
JEdit #3, 5, " whitespace text area visible paint"						
whitespace	3/2	S	12/3	2	S	10
show	4/2	S	9/7	4	S	9
textarea	82/11	S	295/11	3	F	6
Eclipse #1, 2, " mouse double click up down drag release select text offset document position"						
doubleclick	10/8	S	20/14	8	S	12
double	12/10	S	23/17	10	S	14
select	90/32	S	392/34	77	S	269
drag	0/0	F	60/30	0	F	19
mouse	42/12	S	80/12	36	S	56
Eclipse #2, 2, "unified tree node file system folder location"						
filesystem	19/6	S	50/7	5	S	8
system	24/9	S	107/13	10	S	33
file	69/21	S	432/23	31	S	158
unified	33/2	S	53/2	0	F	0
unifiedtree	33/2	S	53/2	0	F	0
add	56/39	S	471/40	51	S	417
node	111/14	S	384/14	21	S	92
tree	199/14	S	655/14	15	F	81
treenode	47/3	S	125/5	0	F	0
folder	14/8	S	101/11	3	F	6
Eclipse #3, 2, "search query quoted token"						
token	25/13	S	64/26	8	S	39
search	128/20	S	338/21	15	S	33
query	49/12	S	76/12	13	S	27
quoted	0/0	F	0/0	0	F	0

If we focus on case studies using original method names, we can see that maximum amplification factor is 6, happened in JEdit #1, whereas maximum

absolute value is only 12, except for Eclipse #3 using. This means, given best queries, traces do not affect TAG performance to critical extent. An adventurous conjecture of this observation is that if programmers are familiar with naming conventions of the program, they can use only one full trace for multiple feature location assignments, when the sought features are not related. In the underneath analysis, we ignore the data obtained from full traces.

String tokens used for GREP were excerpted from queries used for SITIR and were used individually. When using the best query tokens, TAG turns out very effective: performances of TAG are 1, 1, 1, 4, 2.5, and 4, respectively. As Eclipse is much more complex than JEdit, best performances on Eclipse do not degrade more than those on JEdit; this observation shows that the best performance of TAG is scalable.

Whereas, with the specific way in which we composed queries, overall performance of TAG is not that attractive for Eclipse. When using complementary method names, TAG failed twice; when using original method names, TAG failed 7 out of 19 times, i.e. success ratio is merely 63.16%. Though using complementary method names increases recalls remarkably, it may not be a helpful approach in practice due to big result sizes. In SITIR, we noticed that most programmers only viewed top 10 methods in the result; if here we only consider those results whose size is no greater than 20, only 7 out of 19 times TAG works out; the success ratio is merely 36.84%. By no means, we could consider this as evidences of the deficiency of TAG; since in reality it will not be used this way.

One interesting phenomenon that is worthy pointing out is the possibility of small number of classes behind large number of found methods with complementary names. For example, in Eclipse #2 using query "unified", 33 found methods aggregated in 2 classes. This observation implies the possibility that top-down manner can be helpful in feature location. This suggests that TAG results can be used in creative ways.

Another observation about complementary method names is that in best cases there is no added performance compared to original method names; otherwise, they obviously increase recall.

3.3 Discussion

First of all, we have to emphasize that not all queries used for TAG were practically reasonable. To use TAG, we have to be both conservative and aggressive.

As GREP is very strict on queries, e.g., even plural and singular of the same words are considered different by GREP, choosing query tokens is very conservative. For example, in practice, while programmers are not sure whether "quoted" or "quotation" is used in source code, they may search "quot" first to probe the existence of either of the words; and then make their decision of further search accordingly. Those details were not covered in the case studies. In fact, a primitive LSI system may suffer from the same problem [3].

On the other hand, we do not prefer to use words that are generally used, since they cannot serve for the purpose of filtering. For example, searching for "mouse" generally makes no sense.

We consider best cases in our case studies as reasonable evaluations that are close to real performance of TAG. This opinion can be justified by the imaginary situation where programmers compose queries directly from change requests and bug reports. For all the three case studies on JEdit, exact words of "find", "addMarker", and "whitespace" exist on the user interface. In case studies Eclipse #1 and #2, words "doubleclick" and "filesystem" appear in the bug report and both are common code identifiers; in Eclipse #3, novice programmers may use "query" while experienced programmers will know that the bug arises when Eclipse dissects the query sentence into an array of query tokens; nonetheless, using either of the two words produces satisfactory performance, which are 4 and 6.5, respectively.

One important discussion that has been missed in previous publications is why sometimes the cross cut does not work out small result sets as expected. There are two different causes, one is that the words used in queries are commonly used in different situations or for different components; we name this phenomenon word overloading. The other reason, which is totally different from the former one, is due to the substantive details in object programs implementing the sough feature. Eclipse belongs to the latter case since it handles all infrastructural processes by itself.

In summary, our case studies turned out that TAG could perform as well as SITIR with less overhead, although deep discussion reveals that TAG needs strict prerequisite to be successful, fortunately, it can be easily achieved in reality.

4. Related work

Feature location approaches fall into three categories based on how they collect information: static, dynamic, and hybrid. A good overview of static techniques is presented in [4] and a complete survey of dynamic and hybrid approaches is presented in [2].

Wilde and Scully [1] introduced software Reconnaissance in which the program was traced twice: once with the sought feature exercised and the other time without it; the difference set was expected to implement the sough feature. The approach was later formalized by Deprez and Lakhotia [5] and further developed in [6] and adapted to be applied in distributed systems in [7].

Eisenbarth et al. [8] proposed a first hybrid technique, by combining static and dynamic analysis to identify features in source code. The dynamic analysis is performed similarly to Reconnaissance.

The Reconnaissance approach was also extended to Scenario-based Probabilistic Ranking (SPR) by Antoniol and Guéhéneuc [9] with statistical hypothesis testing based on the events that occured in marked traces, knowledge-based filtering, and support for multi-threaded applications using processor emulation techniques, such as Valgrind for C/C++ trace collection and Jikes RVM for Java programs.

PROMESIR [10] combined two existing techniques: SPR [9] of events and Latent Semantic Indexing [11]. The developer traced the program in at least two scenarios for SPR to produce a set of ranked methods relevant to the feature. In addition, the developer used LSI on a query that described the sought feature in natural language. The rankings of the two approaches were combined via an affine transformation. In case studies, PROMESIR showed significant improvements over either SPR or LSI if they were used standalone.

SITIR [2] used one single-scenario trace and LSI to achieve comparable effect of PROMESIR and conducted a thorough survey of various concept location approaches.

5. Conclusion and future work

In this paper we discussed the orthogonality of static and dynamic information in terms of feature location and proposed an improved hybrid feature location approach: TAG. We re-conducted case studies that have been done to evaluate SITIR. In the best cases that at the same time are reasonably easy to achieve, TAG performs as comparably good as SITIR. Through analysis of case study observations, we learn how to improve the performance of TAG and why sometimes hybrid feature location can hardly work out.

While composing queries depends on programmer individuals, there are still many other possible factors that affect the performance of feature location. We will conduct more case studies on various software programs to initialize quantitative analysis of them.

6. References

[1] N. Wilde and M. Scully, "Software Reconnaissance: Mapping Program Features to Code," Software Maintenance: Research and Practice, vol. 7, pp. 49-62, 1995.

[2] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE2007), Atlanta, Georgia, USA, 2007, pp. 234-243.

[3] D. Liu and S. Xu, "Challenges of using LSI for concept location," in the 45th ACM Annual Southeast Regional Conference Winston-Salem, North Carolina, USA, 2007, pp. 449 - 454.

[4] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev, "Static Techniques for Concept Location in Object-Oriented Code," in 13th IEEE International Workshop on Program Comprehension (IWPC'05), St. Louis, Missouri, USA, 2005, pp. 33-42.

[5] J.-C. Deprez and A. Lakhotia, "A formalism to automate mapping from program features to code," in the 8th International Workshop on Program Comprehension (IWPC'00), 2000, pp. 69-78.

[6] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble, "Industrial tools for the feature location problem: an exploratory study," Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, pp. 457-474, November 2006.

[7] D. Edwards, S. Simmons, and N. Wilde, "An Approach to Feature Location in Distributed Systems," Software Engineering Research Center, 2004.

[8] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," IEEE Transactions on Software Engineering, vol. 29, pp. 210 - 224, March 2003.

[9] G. Antoniol and Y.-G. Guéhéneuc, "Feature Identification: An Epidemiological Metaphor," IEEE Transactions on Software Engineering, vol. 32, pp. 627-641, 2006.

[10] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval," IEEE Transactions on Software Engineering, vol. 33, pp. 420-432, June 2007.

[11] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in 11th IEEE Working Conference on Reverse Engineering (WCRE2004), Delft, The Netherlands, 2004, pp. 214-223.

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).
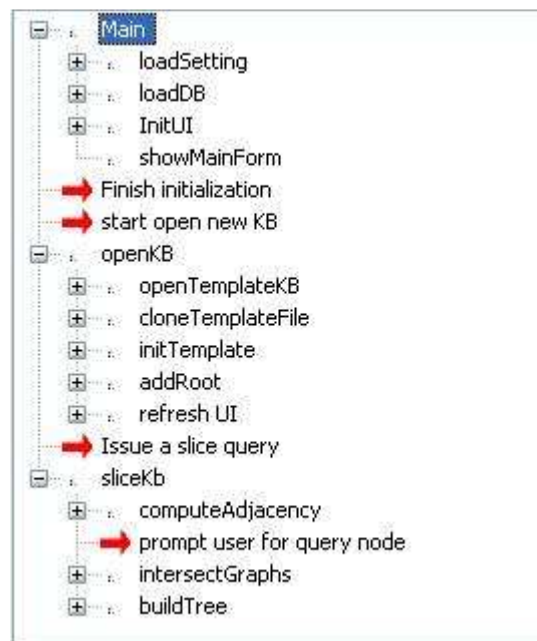

Figure 1. Snapshot of DynaMapper call hierarchy

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.

Towards Seamless and Ubiquitous Availability of Dynamic Information in IDEs

David Röthlisberger and Orla Greevy

Software Composition Group, University of Bern, Switzerland

{roethlis, greevy}@iam.unibe.ch

Abstract

Software developers faced with unfamiliar object-oriented code need to build a mental model of the system to understand its dynamic flow. Development environments typically provide static views of the source code (e.g., classes and methods), but do not explicitly represent dynamic collaborations. The task of revealing how static source artifacts interact at runtime is thus challenging. To address this we have developed several techniques to represent dynamic behavior at various levels of granularity directly in the IDE. In this paper we outline these various techniques towards a seamless integration of dynamic information in the IDE. We elaborate on user feedback we have gathered and on our empirical experiments to validate our work. We derive several ideas and visions of further potential representations of dynamic behavior from this analysis of our approach. The missing representations we identify serve to enrich our proposed IDE, so as to provide the developer from within the IDE with a readily available and complete picture of a software's dynamics.

Keywords: dynamic analysis, dynamic collaborations, development environments, program comprehension

1 Introduction

Maintaining or enhancing object-oriented software systems requires developers not only to understand static source artifacts, but also their dynamic interaction. The primary tool available to developers, the integrated development environment (IDE), typically focuses on a static view of a system. It does not explicitly represent dynamic collaboration between static artifacts (*e.g.*, classes or methods). In the absence of IDE support developers are forced to build up a mental model of a system's dynamic behavior. Integrating explicit representations of dynamic behavior directly in the IDE would prove helpful in gaining a more accurate understanding for a system under investigation.

To achieve the goal of representing dynamic behavior seamlessly in the IDE, we are faced with several challenges, such as:

- *How can we efficiently gather dynamic information and immediately make it available from within the IDE?*
- *How do we represent dynamic behavior of a system in an IDE?*
- *How do we validate that our proposed representations are useful for developers?*

Our key focus is to present our experience to date and to identify our visions for further IDE enhancements to exploit seamless integration of dynamic information in various forms, providing developers with relevant information to understand a software's dynamics.

In this paper we report on the techniques we devised to address the above challenges. In Section 2 we present an overview of our techniques to represent dynamic behavior explicitly in the IDE, such as (i) visualizations, (ii) enrichments to the source code view, or (iii) techniques to query dynamic information from within the IDE. We present a summary of developer feedback and results of our evaluations in Section 3. Based on this, we have identified representations of dynamic behavior to support developers In Section 4, we outline ideas for further enhancements to an IDE encompassing dynamic information.

2 Existing Approaches Integrating Dynamic Analysis in IDEs

In our work to date, we have developed four different approaches to reason about dynamic information directly in the IDE. Each approach works on different levels of granularity, from the fine-grained source code level, the dynamic interaction of static artifacts to a coarse-grained representation of user-identifiable features of a system. Our techniques provide the developer with several entry points for gaining an understanding of software system, *e.g.*, to correct a specific defect. If a defect occurs in a specific feature, the developer may first gain an overview of the feature's dynamics, then locate candidate entities (*e.g.*, methods) that

may contain the defect. In a next step, the developer reasons about the specific communication patterns between the candidate entities and finally drills down to the source code level to study the dynamics on a fine-grained level to pinpoint and correct the defect.

We provide a brief overview of our four proposals to represent dynamic information in the IDE. We implemented our IDE enhancements in the Squeak Smalltalk IDE [7] as it provides an extensible framework to adapt and extend its tools, We take advantage of our previously implemented a technique, partial behavioral reflection, to efficiently and selectively gather runtime information [3].. Applying our enhancements to other IDEs, *e.g.*, Eclipse could be achieved using similar techniques.

2.1 Feature Representation

To explicitly represent features, *i.e.*, behavioral entities of a software system, we introduce our Feature Browser [10], an enhancement to a traditional IDE.

We describe our Feature Browser taking as an example a Wiki application. The developer first specifies within the IDE that dynamic data should be recorded for the application (*i.e.*, at the package level) and then associates names with the features (*i.e.*, external user-understandable units of behavior of the application) under investigation, *e.g.*, "wikiEditPage". Then she exercises a feature in the application. The IDE takes care of gathering and storing dynamic data of the feature. The IDE now provides the developer with an explicit feature representation of behavioral data for "wikiEditPage". To study the features of interest, the developer selects them either in our Feature browser or invokes an action we added next to the class and method browser to open all features that use a particular class or method.

Figure 1 depicts our feature browser's core components. The *Compact Feature Overview* (1) enables visually comparison of several features . The small nodes in a feature view can represent either methods or classes and are colored according to the feature affinity metric proposed by Greevy [5]. Entities used in only one feature (colored blue) can be distinguished from entities used in several or all features (colored orange or red). The coloring scheme makes it easier to quickly grasp similarities between features, anomalies or to locate erroneous behavior.

The *Feature Tree* (2) provides the developer a more detailed view on a feature by representing the method call tree triggered while it was exercised. The root of the tree is the first, *e.g.*, the "main" method of the feature, child nodes are methods being invoked by this main method. All nodes in this tree are colored according to the feature affinity metric. To make this tree navigable for reasonable sized execution traces we applied several compression techniques such as subexpression removal [8] or sequence and repetition re-
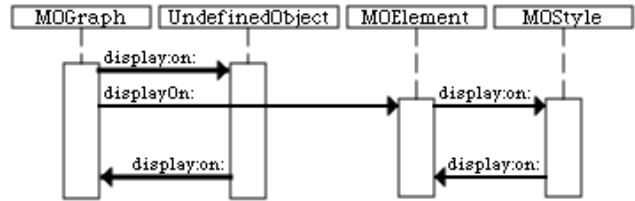

Figure 2. Class collaboration chart for class Graph.

moval as proposed by Hamou-Lhadj [6]. A developer can open this feature tree by clicking on a node (*i.e.*, a method) in the compact feature overview or by selecting a feature in which a method opened in the IDE participates.

The *Feature Artifact Browser* (3) shows all entities used in a particular feature in a dedicated source browsing environment. Only entities (*e.g.*, packages, classes, or methods) which are actually used in the selected feature are shown so the developer can focus on parts of the code responsible for the feature's behavior.

2.2 Representing Dynamic Collaboration

To refine their mental model of a feature's behavior, developers typically want to reason about more fine-grained interactions to reveal how classes communicates with each other. Studying this kind of dynamic interaction may uncover unwanted behavior, such as incorrect or missing communication between instances. To study this level of interaction we provide a range of *collaboration charts*. A class collaboration chart of the class *Graph* of a visualization tool is shown in Figure 2. Similar charts exist for packages or methods.

Our charts show compact representations of package, class or method runtime communications. Our class collaboration chart is similar to a UML sequence diagram, although the order of calls is not preserved, To avoid cluttering the chart with too much information, we show communication paths between classes, *i.e.*, message sends occurring in an instance of a class with an instance of another class as a receiver, as edges in the chart. The thickness of an edge reflects the relative frequency of the interaction, as in the work of Ducasse *et al.* [4].

Our charts are directly accessible either from within the feature browser, or from the static view on source code of the IDE, *e.g.*, by selecting a particular class and opening a class collaboration chart for this class. In the latter case, the application has to be executed before the class collaboration chart can be shown. The charts are always dedicated to a specific run of the subject system triggered by the developer, either by running scripts to exercise behavior or by
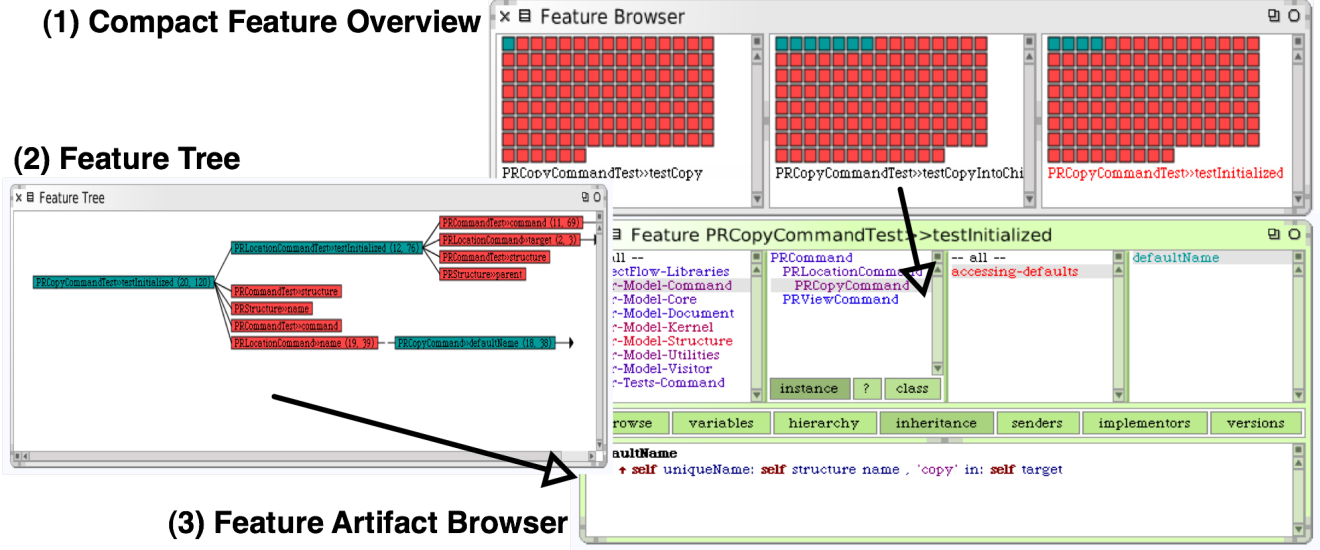
(1) Compact Feature Overview

(2) Feature Tree

(3) Feature Artifact Browser

Figure 1. Schema of the Feature Browser.

manually interacting with the subject application.

2.3 Dynamic Data Querying

Dynamic analysis approaches need to deal with vast amounts of data [1]. In the two previous approaches, we addressed this by focusing on one particular execution of a system and by compressing the resulting execution trace. However, developers often want to understand the dynamic behavior of the application "in general", *i.e.*, for as many different executions as possible, although full coverage is of course not achievable for any reasonable big system [1]. For this reason, we keep the data generated by observed entities in a central database accessible from within the IDE [9].

To effectively reason about the permanently stored dynamic data, we extended the IDE's search capabilities consider both static and dynamic information. Our extended search enables the developer for instance to search for senders of messages to a specific receiver type, *e.g.*, only for methods invoking the *size* method of class *Graph*. The query to solve this problem is shown in the code section below (query 1).

```
SHOW senders OF Graph.size
SHOW collaborators OF Graph
SHOW method invocation IN Wiki ORDER BY
    frequency
```

The query language syntax is similar to SQL. Query 2 returns all classes collaborating with *Graph* at runtime, while query 3 provides a list of all methods being invoked in the package *Graph*, ordered by invocation frequency.

Other search facilities are dynamic implementors, package or method collaborators, or method execution times. The results of such queries are directly embedded in the IDE and can be browsed using IDE functionalities.

2.4 Dynamic Information Integrated in Source Code

On the lowest level of granularity, we embed dynamic information directly into the source code of methods [11]. When reading source code of dynamically-types languages such as Smalltalk, it may be difficult to completely understand the code as there is no type information. Polymorphism further complicates the task. It is unclear which methods are invoked at runtime and what kind of objects are stored in variables. We enrich the source code view to feed in information obtained by dynamic analysis. The code statement in Figure 3 highlights our enhancements to the source view. We add icons to message sends and variables accesses in source code. Clicking on an icon either reveals what methods were executed for a message send or show all type of objects a variable stored at runtime. Of course the developer can directly navigate to a method or a variable type shown in the respective list by clicking on the item. An interesting side-effect of these enhancements is that they also reveal which parts of a method have never been executed, as these parts will be missing these icons for dynamic information.

To obtain the dynamic data for these extensions we query the database mentioned in Section 2.3. We apply caching strategies: as soon as a method's source code has been

```
element ▣ do:↵ [:each ▣ | each ▣ someTask ➡].
```

Figure 3. Dynamic information embedded in source code.

Statement	Av. rating
Impact of feature browsing in program comprehension	4.2
Impact of collaboration charts on program understanding	3.2
Effect of collaboration charts on execution overview	4.3
Impact of querying dynamic inf. on prog. understanding	3.4
Impact of querying dynamic inf. on navigation of static artifacts	3.8
Effect of source code enrichments on execution overview	4.0
Effect of source code enrichments on navigation of static artifacts	3.9
Impact of source code enrichments on program comprehension	3.3

Figure 4. Answers obtained from our questionnaires

displayed once, including dynamic information icons. We cache results of various queries submitted for this method until either the method's source code has changed or more dynamic information has been gathered.

3 Validation of Existing Techniques

We validate our various approaches to integrate dynamic information in the IDE from a user perspective. In previous works we also validate it from an efficiency and performance perspective [10, 11].

3.1 User Validation

We validated out *Feature Browser* (Section 2.1) by means of an empirical study involving twelve developers familiar with both the Smalltalk language and IDE. We asked the subjects to correct two defects of similar complexity in a Wiki system. For one defect the developers used the traditional Squeak IDE, for the other we provided them with our IDE-embedded feature browser. The order in which they used each environment was randomly assigned. We then compared both, the efficiency (*i.e.*, time spent) to correctly locate the cause of the defect in source code and to actually correct the defect entirely. The performance of the subjects was in average 30% better with the feature browser concerning defect location and 10% better concerning defect correction. Both figures are statistically significant. For more details of this study we refer the reader to our previous work [10].

We validated the other techniques, collaboration charts, dynamic information querying, and enriched source code view, by means of providing a questionnaire to several developers and asked them to apply our techniques in a controlled experiment we defined. We also involved the subjects of the feature browser. For all techniques, we used set of general questions as well as specific questions for each technique. Every questionnaire was answered by at least three subjects. We used the same Wiki application (*i.e.*, Pier) for each experiment as none of the subjects had prior knowledge of this system. We assigned the subjects specific tasks to solve, providing them with just one of our four techniques. The tasks were for instance to describe the role of an key model class, to enhance the system with a feature similar to an already existing feature, or to adapt a feature without impacting any other system behavior. After solving

three tasks we gave the subjects the questionnaire. Table 4 provides a selection of answers from the questionnaires.

We obtained many suggestions, ideas, or wishes for future enhancements to represent dynamic information in the IDE, this feedback incorporates in Section 4.

4 Competing the Representation of Software Dynamics in IDEs

We elaborate on several opportunities to extend our existing work on integrating dynamic information in the IDE. We identify shortcomings, problems, or issues in the current work and present ideas and suggestions obtained from developers that participated in our experiments.

Identifying Missing Features. A shortcoming of the current solution is the requirement to select specific static artifacts of the subject system (*e.g.*, packages or classes) to collect dynamic data, and to then run one or many system's features. The IDE should *automatically* take care of gathering dynamic information from all system entities. Dynamic data should be as readily available as static information (*e.g.*, list of methods or instance variables of a class). Moreover, developers want to be able to associate a particular execution with the dynamic data it generated, but for other scenarios they also want to access all gathered information about an artifact in order to achieve a *high level of coverage*. If the IDE were to automatically collect dynamic information, developers would be freed from this responsibility and would be more likely to incorporate views on system's dynamics in their daily work, in particular when these views show reliable, complete and accurate information.

To gather dynamic data, a system first needs to be executed. Instead of relying on the developer to run the application manually or with scripts, the IDE could *continuously run the system* in the background, in particular after changes to the system's code base. The developer could record some scripts on a high level (*e.g.*, by recording user

actions in the application) that could be fed into the IDE so it could run the system. The IDE could easily determine code (*e.g.*, methods or source code statements) that have never been executed and either try to find execution paths for this code or alert the developer to refine the provided scripts. This procedure would improve code coverage. The general, idea is to empower the IDE and to relieve the developer of the responsibility to ensure that as many parts of the system as possible are covered by dynamic analysis. The IDE is the appropriate tool to assume this responsibility as it is very familiar to developers (they spend most of their working time in this environment), and as it already provides sophisticated means to work with static code. We understand dynamic views as being orthogonal to static views and hence nicely completing IDE's mostly static perspective.

Developer Suggestions. One suggestion of developers was to use dynamic information not only to enhance and complete the static perspective of a system, but to build means and concepts to browse, develop and maintain software in a environment that *primarily display entities by their dynamic relationships*, *e.g.*, a browser that shows classes on a two-dimensional map showing communication as paths while the distance between any two classes represents how heavily they communicate with each other. Entities are placed closer to each other the more they collaborate. Another developer mentioned the importance of having *full coverage*, *i.e.*, he often wants to know whether two entities will ever communicate to each other in any possible system execution. Of similar importance is a *big picture view*: While focusing on a particular feature or execution is interesting in many scenarios, there is often also a need to get an overview of all possible dynamic communication occurring in an application, *e.g.*, to present to a new developer how the system generally functions at runtime.

5 Conclusions

In this paper we described four different techniques to seamlessly integrate dynamic information in IDEs to reason about software's dynamics. These four techniques are (1) a feature browser to reason about features, (2) collaboration charts to visualize dynamic communication between static artifacts in the IDE, (3) facilities to query dynamic information, and (4) enrichments to source code to embed information of its dynamic behavior. We performed several user experiments to evaluate these techniques and to solicit feedback from developers about ideas for future enhancements. We presented both the results from the various studies (*e.g.*, results of questionnaires) and a comprehensive list of issues in the current approach. Finally we identified further opportunities for extend and complete the represen-

tation of software's dynamics in IDEs.

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, 1999. Springer Verlag.

[2] M. Denker and S. Ducasse. Software evolution from the field: an experience report from the Squeak maintainers. In *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, volume 166 of *Electronic Notes in Theoretical Computer Science*, pages 81–91. Elsevier, Jan. 2007.

[3] M. Denker, O. Greevy, and M. Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[4] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[5] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[6] A. Hamou-Lhadj and T. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of 1st International Workshop on Dynamic Analysis (WODA)*, May 2003.

[7] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[8] J.-M. S. Philippe Flajolet, Paolo Sipala. Analytic variations on the common subexpression problem. In *Automata, Languages, and Programming*, volume 443 of *LNCS*, pages 220–234. Springer Verlag, 1990.

[9] D. Röthlisberger. Querying runtime information in the ide. In *Proceedings of the 2008 workshop on Query Technologies and Applications for Program Comprehension (QTAPC 2008)*, 2008. To appear.

[10] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007)*, pages 79–100. ACM Digital Library, 2007.

[11] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the ide. In *Proceedings of the 2008 International Conference on Program Comprehension (ICPC 2008)*, 2008. To appear.

Using Dynamic Analysis for API Migration

Lea Haensenberger and Adrian Kuhn and Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
lhaensenberger@students.unibe.ch, {akuhn,oscar}@iam.unibe.ch

Abstract

When changing the API of a framework, we need to migrate its clients. This is best done automatically. In this paper, we focus on API migration where the mechanism for inversion of control changes. We propose to use dynamic analysis for such API migration since structural refactorings alone are often not sufficient. We consider JExample *as a case-study.* JExample *extends* JUnit *with first-class dependencies and fixture injection. We investigate how dynamically collected information about test coverage and about instances under test can be used to detect dependency injection candidates.*

Keywords: API migration, automatic software engineering, dynamic analysis, inversion of control.

1. Introduction

Large software systems are not written from scratch, but rather reuse functionality offered by third-party frameworks. Frameworks provide their functionality through an application programming interface (API) that typically inverts the control between client and framework.

"The framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application." – Ralph Johnson *et al* [6]

An API is a contract between framework and client that guarantees stability. No changes to the client are required when updating the framework, or even, when moving to a framework implementation of another vendor. However, sometimes comes the moment when we must migrate the client to a different API.

Migrating client code from one API to another is tedious work and thus best done automatically. Sometimes this can be done using a series of refactorings that map the structure of one API to the other. However, when the mechanism for inversion of control differs, a mere structural mapping is often not sufficient. Therefore, we propose to use dynamically collected information for automatic migration of APIs with different mechanism for inversion of control.

In this paper, we consider JExample as a case-study [7, 4]. JExample extends JUnit with first-class dependencies and fixture injection. We identified the following migration steps that require information obtained from dynamic analysis in order to be done.

- For detection of dependencies we propose to record the coverage set of each test, such that the partial order, *i.e.,* subset relationship, of coverage sets can be used to introduce dependencies.

- For detection of injection candidates we propose to use record the state of the instances under test, such that redundant setup code can instead be replaced with fixture injection.

The remainder of this paper is structured as follows. Section 2 introduces JExample. We propose why and how to use dynamic analysis to detect dependencies (Section 3) and candidate fixtures (Section 4). We discuss other that might require dynamic migration in Section 5, and Section 6 concludes.

2. JExample in a Nutshell

JExample introduces producer-consumer relationships to JUnit unit testing.

- A producer is a test method that yields an instance of its unit under test as return value.

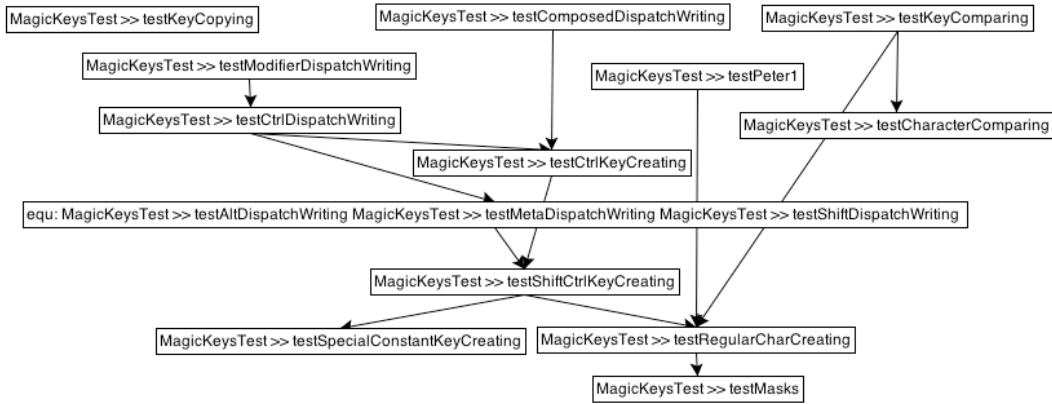- A consumer is a test method that depends on one or more producers.

Figure 1. Partial order of the coverage sets of the test suite of `MagicKeys`. Figure courtesy of Gaelli et al [3].

JEXAMPLE caches the return values of producer methods and injects them when a consumer is about to be executed. Producer-consumer relationships are first-class dependencies: when running a test suite, JEXAMPLE will skips any test method whose producers have previously failed or been skipped.

For example, when testing a stack class, you may declare that `testPop` depends on the unit under test of `testPush` as follows.

```
@Test
public Stack testPush() {
    Stack $ = new Stack();
    $.push("foo");
    assertEquals("foo", $.top());
    assertEquals(1, $.size());
    return $;
}


@Test
@Depends("#testPush")
public void testPop(Stack $) {
    Object top = $.pop();
    assertEquals("foo", top);
    assertEquals(0, $.size());
}
```

We refer to producers and consumer methods as *example methods*. They do more than just test the unit under test. Producer methods consist of source code that illustrates the usage of the unit under test, and that may return a characteristic instance of their unit under test. Thus, producer/consumer methods are in fact examples of the unit under test.

As such, example methods tackle the same problem as mock objects, *i.e.,* "How to test a unit that depends on other units?" When working with mock objects, you solve this problem by creating a mock for each dependency. When working with examples, you solve this problem by declaring producer-consumer dependencies. Thus, instead of testing against mocks, you test against the previously created return values of other tests. Since example methods are both producers and testers of their returned value, all return values are guaranteed to be valid and fully functional instances of the corresponding unit. In addition, JEXAMPLE will clone example objects to take care that no side-effects are introduced when two or more consumers use the same return value.

An example method may depend on both successful execution and return values of other examples. If it does, it must declare the dependencies using an `@Depends` annotation. An example method with dependencies may have method parameters. The number of parameters must be less than or equal to the number of dependencies. The type of the n-th parameter must match the return type of the n-th dependency.

Dependency declarations uses the same syntax as the `@link` tag of the Java documentation tool. References are either fully qualified or not. If less than fully qualified, JEXAMPLE searches first in the declaring class and then in the enclosing package. The following table shows the different forms of references.

```
#method
#method(Type, Type, ...)
class#method
class#method(Type, Type, ...)
package.class#method
package.class#method(Type, Type, ...)
```

Multiple references are separated by either a comma (,) or a semicolon (;). As listed above, the hash character (#), rather than a dot (.) separates a member from its class. However, JEXAMPLE is generally lenient and will properly parse a dot if there is no ambiguity. This is the same as the Java documentation tool does.

3. Detecting Dependencies

Test methods in JEXAMPLE can have explicit dependencies on other test methods. If dependencies are properly declared, a failing method directly points to the defect location (since all dependents that would otherwise fail as well are skipped) whereas in JUNIT many dozens of test methods covering the same defect location fail and it is often not obvious where to start fixing the bug. A test method m_a should depend on a test method m_b, if m_a covers at least the same code as m_b. The execution of method m_a can be skipped, if the framework already knows that m_b fails.

Gaelli *et al* have shown that a partial order of test methods by means of *coverage sets* helps developers to locate a defect by pointing out the test method with the most specific debugging context [3]. Figure 1 illustrates the partial order of one test suite.

Thus, we propose to migrate JUNIT tests to JEXAMPLE by running the tests and recording the coverage set of each test. The coverage set of a test contains all methods that get invoked when running the test. In the main step we make a partial order of coverage sets, in order to detect coverage dependencies between test methods. If a JUNIT method m_a is found to cover a superset of a JUNIT method m_b, then m_b is migrated to a JEXAMPLE method with a `@Depends` annotation to m_a.

Consider the following two test methods of a JUNIT test case testing Java's Stack.

```
@Test
public void testPush() {
    Stack stack = new Stack();
    stack.push("foo");
    assertEquals("foo", stack.top());
    assertEquals(1, stack.size());
}

@Test
public void testPop() {
    Stack stack = new Stack();
    stack.push("foo");
    Object top = stack.pop();
    assertEquals("foo", top);
    assertEquals(0, stack.size());
}
```

In the example above `testPop` covers `testPush`, since the set of methods invoked by `testPop` is a superset of the methods invoked by `testPush`. Thus, in the JEXAMPLE implementation of the Stack test, as given previously in Section 2, `testPop` declares itself to explicitly depend on `testPush` as follows:

```
@Test
@Depends("#testPush")
```

```
public void testPop() { ...
```

Please note, as we add a `depends` annotation but no methods parameters, a dependency without fixture injection is added. Detection of fixture injection candidates is covered in the next section.

4. Detecting Candidates for Fixture Injection

Test methods in JEXAMPLE can pass an instance of the unit under test (instance under test) from one test method to another. If a test method m_a returns a value, the JEXAMPLE framework caches this return value. If later the framework is about to execute a test method m_b that depends on m_a, the cached return value of m_a is cloned and injected as a parameter to the method invocation of m_b. As such, in JEXAMPLE a test method may provide the fixture for its dependent methods. Thus, we refer to the former as the *producer* and to the latter as its *consumers*.

Again we consider the Stack example as given in Section 3. In JUNIT both `testPush` and `testPop` create a new instance of Stack, the unit under test. Both methods push the same String onto their Stack instance, thus they share the same setup of the instance under test. The method `testPush` ends at this point, whereas `testPop` continues with further operations on its instance. In JEXAMPLE we can rewrite this so that `testPush` returns its instance under test as return value and `testPop` expects this return value to be injected as a method parameter.

We propose to migrate JUNIT tests to JEXAMPLE by running the tests, but this time recording the created instances under test. If at any moment during the execution of test method m_a the instance under test has the same state as method m_b's instance under test at the end of method m_b, then we have found a candidate for fixture injection.

There are two possible techniques to check if the instances under test are the same:

- All fields of both instances are equivalent.
- The path that produced the instance is the same.

For example, even if the String pushed by m_a and m_b is not the same, we might consider it as a fixture injection candidate. This candidate might however be a false positive if m_a or m_b tests a boundary condition that particularly depends on the pushed value.

In the same way, it is possible to create an empty Stack instance by many different paths that might not all be equivalent. For example, a freshly created Stack might have a different modification count than one that has been filled and later emptied again.

We propose to use both techniques, since singeling out false positive candidates is straightforward: if the migrated tests do not run we have obviously introduced an error. Thus, if we migrate the candidates one by one we can make sure no false positives are migrated.

The migration script will collect all candidates, as well as the dependencies detected above in Section 3, in order to migrate JUNIT test suites to JEXAMPLE test suites. If both a dependency from m_a to m_b and a fixture candidate has been detected, the method m_a is rewritten to not only depend on m_b but to also take m_b's return value as a parameter as follows:

```
@Test
public Stack testPush() {
    ...
    return stack;
}

@Test
@Depends("#testPush")
public void testPop(Stack stack) {
    ...
}
```

In addition, if a JUNIT test-class has a `@Setup` method, this method will be migrated to a JEXAMPLE test method that is a producer, and all other methods in the same test-class become its consumers.

5. Current and Emerging API Trends

In this section, we provide current and emergent API trends where the mechanisms for inversion of control is affected, and suggest how dynamically obtained information might be useful in order to migrate these APIs:

- XML frameworks offer a wide range of APIs with different control mechanisms. The main divisions are tree- and streaming-based APIs, with the streaming APIs further subdivided into push- and pull models. For example, the DOM model is tree-based [11], whereas SAX uses a push-model streaming API [10]. In addition, non-imperative APIs are emerging that enrich XML processing with the functional and logical paradigm. For Example, LINQ uses functional queries to map objects to XML or SQL and back [8].

- The latest release of J2EE, Java's enterprise application framework, moves from EJB's heavy-weight applications servers to light-weight technologies such as Hibernate and Spring [5]. Both approaches use inversion of control[1], but employ different mechanisms in order to do so. EJB hard-wires application code into the application server framework by passing around explicit references to the container. Spring on the other hand uses *dependency injection* to inject container-provided objects into annotated fields of the application code. At this moment, many J2EE systems are about to be migrated from EJB 2.0 to EJB 3.0, that is from conventional application servers to Spring and Hibernate.

- In the field of unit testing, frameworks with first-class dependencies are emerging. For example, both TESTNG and JEXAMPLE extend conventional unit tests with dependencies between tests [7, 4]. When running tests, the framework can skip tests whose dependencies have failed. In addition, JEXAMPLE introduces producer-consumer relationships, where the return value of a producer test is cached by the framework and later injected into the consumers as their fixture.

- Web frameworks are another field where "inversion of control is inverted back" [9]. For example, the Seaside framework uses continuations rather than the goto-like style of page-centric programming [1]. Rather than writing the web application page by page, one (or more) main methods capture the complete flow of the application, using call-backs and control flow structures provided by the Seaside API to handle page transitions.

Migration between APIs with such different mechanisms for inversion of control, some even based on conflicting paradigms, cannot necessarily be done with a simple set of structural refactorings. Additional runtime information might be required.

For example, to migrate from a tree- to a streaming-based XML framework, we might dynamically record all operations performed on the tree and then check if these operations can be re-ordered such that they can be applied in streaming fashion.

For example, to migrate a conventional XML query to a LINQ query, we might dynamically record the imperative sequence of instructions (which will certainly include many `for` loops and `if` statements) performed during the query and then check if we can find

---

1 "There is some confusion these days over the meaning of inversion of control due to the rise of IoC containers; some people confuse the general principle here with the specific styles of inversion of control (such as dependency injection) that these containers use. The name is somewhat confusing (and ironic) since IoC containers are generally regarded as a competitor to EJB, yet EJB uses inversion of control just as much (if not more)." – Martin Fowler [2]

a LINQ equivalent that returns the same elements as the recorded `for` loops and `if` statements.

For example, XML namespaces in LINQ are represented with a dedicated `Namespace` class rather than plain strings. Thus, when migrating towards LINQ, we might dynamically record the usage of all strings in order to single out those that can be migrated as XML namespaces rather than strings.

For example, to migrate a J2EE application, we might dynamically record the resources requested from the EJB container in order to replace these calls with corresponding Spring injection annotations.

For example, to migrate from a conventional web framework to Seaside, we might dynamically record the page transitions for different tasks in order to generate high-levels methods that capture this flow of pages in one method. For example, given a Wiki application, one could record tasks such as login, create page, edit page, remove page, and generate a corresponding high-level method that captures the entire page flow of each of these tasks.

6. Concluding Remarks

In this paper, we investigate API migration where the mechanism for inversion of control changes. We propose to use information obtained from dynamic analysis for such API migration.

We provided examples taken from current and emerging industry trends. As a case-study we investigated in further detail how to migrate JUnit tests to JExample using information obtained from dynamic analysis, and propose two particular migrations steps that require dynamic analysis.

The first author of this paper is currently realizing the proposed steps as part of her Master's thesis.

References

[1] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[2] M. Fowler. Inversion of control, obtained from Martin Fowler's wiki, June 2005.

[3] M. Gaelli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[4] L. Haensenberger. JExample. Bachelor's project, University of Bern, Mar. 2008.

[5] R. Johnsohn and J. Hoeller. *Expert One-on-One J2EE Development without EJB*. Wrox, 2004.

[6] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[7] A. Kuhn, B. V. Rompaey, L. Haensenberger, O. Nierstrasz, S. Demeyer, M. Gaelli, and K. V. Leemput. JExample: Exploiting dependencies between tests to improve defect localization. In P. Abrahamsson, editor, *Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008*, Lecture Notes in Computer Science, pages 73–82. Springer, 2008.

[8] Language Integrated Queries. http://plone.org/products/archgenxml.

[9] C. Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[10] Simple API for XML. http://www.saxproject.org/.

[11] L. Wood, J. Sorensen, S. Byrne, R. Sutor, V. Apparao, S. Isaacs, G. Nicol, and M. Champion. *Document Object Model Specification DOM 1.0*. World Wide Web Consortium, 1998.

Applying Static and Dynamic Analysis in a Legacy System to Study the Behaviour of Patterns of Code during Executions: an Industrial Experience

Rim Chaabane, Françoise Balmas
Laboratoire LIASD, Université Paris 8, France
{rchaabane, fb}@ai.univ-paris8.fr

Abstract

This paper describes our work for detecting and analysing the performances of some patterns of code during their execution. This work was realized for a legacy software written in a proprietary, procedural and compiled language. The approach we present here uses static analysis techniques to detect patterns of code in source and in bytecode without modifying any of them. We also share our experience on dynamic analysis on pattern analysis in this specific legacy system.

1. Industrial issues

Our work is performed in the context of a financial company that develops and maintains a legacy software [1]. This software was initially written more than twenty years ago, in a proprietary, procedural and 4th generation language (4GL) called ADL[1] (Application Development Language). This software was initially developed under the VMS operating system and is based on a previous generation of database. For commercial reasons, the software was ported to UNIX systems and adapted to newer relational databases, Oracle and Sybase. It was also extended to offer a web interface. Currently the software has 10 million lines of ADL source code, and some expanded source files or procedures called by the main program can reach 400,000 lines of code each.

This legacy software has to face some new challenges like database growth. These last 20 years some client companies merge together, which made their data grew to more than one Terabyte and will extend even more during the next years. This made that the software shows critical decrease in performance. Some ADL database access statements or patterns of code (POC) were suspected by developers to be responsible for this performance decrease.

In order to identify which POC has shown poor performance, code maintainers should inspect all instances of POC to check them. Due to the size of the software, this would mean inspecting hundreds and hundreds of instances. Since this is clearly unfeasible, we developed a combined approach to search for all possible instances and to analyze them from a behaviour point of view.

We developed the "Adlmap" tool, based on static analysis, which searches for the suspected instances of POC in source code. We can find hundreds of instances for a given procedure. Not all the instances are called during a product run of the legacy software. Also not all called ones have poor performance. To identify those instances which really decrease the performance, we need to analyse their execution behaviour. For that we developed the "Pmonitor" tool that executes instances and measures their performances in order to classify them. Maintainers can then identify which instance need to be focused on.

In this way, we can know which instances of these POC decrease performances in a 'real-life' customer execution. Maintainers therefore have only a few instances to improve.

In Section 2, we describe the technique we use to detect instances of POC in source code, while in Section 3 we explain our technique to dynamically analyse their performance. In Section 4 we give a general survey on our current work.

2. Pattern detection in source code with Adlmap

To be able to measure the performance of POC instances during their execution, we need to be able to detect them inside the compiled code. Most of the time two techniques are used:

Figure 1. Adlmap architecture

1. Modifying the source code to insert instructions before and after each instance of a POC [3, 4] which perform measurements.

2. Modifying the compiler to generate an instrumented bytecode to measure the performances of each encountered instance of pattern. While the bytecode is executed, the additional instructions generate information about their performance [5].

In our context, these two solutions cannot be applied because we do not want to modify either the source code or the bytecode. The first technique would increase the number of lines in the source code, whereas our compiler shows limitations when compiling very large source files. The second technique would require a change in the bytecode which would mean modifying the compiler as well. Since the ADL compiler has not been maintained for years, this solution cannot be considered.

Our solution is centered on identifying all suspected POC instances in a given procedure and in its corresponding bytecode, modifying neither source code, bytecode nor compiler. For this purpose, we rely on the fact that all the suspected POC are database accesses (DBA), since DBA are easier to detect in bytecode than POC. We developed the Adlmap tool that scans both source code and bytecode to list all the database accesses.

Adlmap relies on mapping rules to match the DBA extracted from source code to those extracted from bytecode. During the extraction of DBA from source code, Adlmap checks if each of them verifies the definition of a suspected POC, if that is the case, the DBA is flagged with the name of the POC[2]. Since the list of extracted DBA from source code, maps the list extracted from bytecode, we can link the flagged DBA to their corresponding bytecode instructions. This way we can identify the suspected instances of POC in the bytecode before its execution. Let's see this in more details with exmple in figure 1.

To extract and flag the DBA from the source code, Adlmap uses Delia [6, 7] a tool for ADL static analysis. Delia builds an abstract syntax tree (AST) from the source code that can be transformed into an XML tree (cf. Part A in figure 1), which it scans with Xpath requests [8] to extract all the DBA. Xpath requests are also used to identify those DBA that are suspected POC. The DBA that are suspected POC are then flagged (cf. Part B in figure 1).

To extract the DBA from bytecode we need to load the binary code produced by the compiler into the Runtime (cf. Part C in figure 1), in which each instruction has its own program counter (PC). The Runtime offers many tools to navigate inside the bytecode and to detect DBA instructions. Adlmap uses these features to extract all DBA (cf. Part D in figure 1).

To find the suspected POC instances in bytecode, Adlmap makes *all* the DBA extracted from source code match those extracted from bytecode. Thus Adlmap maps the suspected POC instances source line numbers to their corresponding PC in bytecode and reports these results in a mapping table (cf. Part E in Figure 1).

This mapping allows us to identify the suspected POC instances in bytecode before its execution. The next Section explains how Adlmap results are used during dynamic analysis.

3. Pattern monitoring with Pmonitor

Since one given suspected POC can have more than one hundred instances in a procedure with near 400 000 lines of ADL source code, we need to use dynamic analysis to identify those instances with poor

performances and the contexts that make them behave this way.

For this, we use a Runtime feature that executes the bytecode with an activated trace mode. This mode creates huge log files arduous to analyse. The Runtime has an other feature that groups bytecode instructions by sequences. This feature can be used on trace mode to obtain log files of reduced size. Reducing the size of log files permit to reduce the time to analyse them

We developed the Pmonitor tool to analyse these log files and to extract a report containing the performances of each executed bytecode instruction. The suspected POC instances are flagged to be compared to the rest of executed bytecode sequences.

In the example given in Figure 2, we used our Adlmap tool to extract from the procedure called "NEXMO" a mapping table. The mapping table contains information on all the DBA found in procedure's source files, such as the source line number, PC, the name of the accessed relation and it's suspected pattern (cf. Part A in Figure 2). Adlmap identified 12 000 DBA in "NEXMO" source code, 75 of them are suspected POC of type P1[3]. In the table shown in figure 2, only 11 instances of pattern P1 are shown (the colored ones are those that are called during code execution).

The "NEXMO" procedure is then compiled to obtain the bytecode that is loaded in the Runtime which executes it in trace mode. A log file is created at the end of the execution (cf. Part B in Figure 2). This log file contains the different sequences of bytecode, the time taken for execution and the PC of the first instruction of the sequence.

We then use Pmonitor to analyse this log file. It first counts the number of calls for each sequence of bytecode and the average time its execution takes. Pmonitor uses then the results of the Adlmap tool to match the executed sequences of bytecode that contain a DBA to the corresponding static information such as the line number or the relation name accessed and if it's a suspected POC instance. A combining report file is created (cf. Part C Figure 2) with detailed information on executed instances of POC that can be compared to other DBA executions and non DBA bytecode sequences. We can see in this example that only 9 P1 patterns were executed. The most expensive one in terms of execution time is called 830 times and

---

[2] ADL maintainers established a list of suspected POC which are named P1, P2, P3, etc...
Unitl now, only P1 and P2 are detected by Adlmap

[3] P1 is a table scan and P2 a table scan containing as first instruction an if statement.
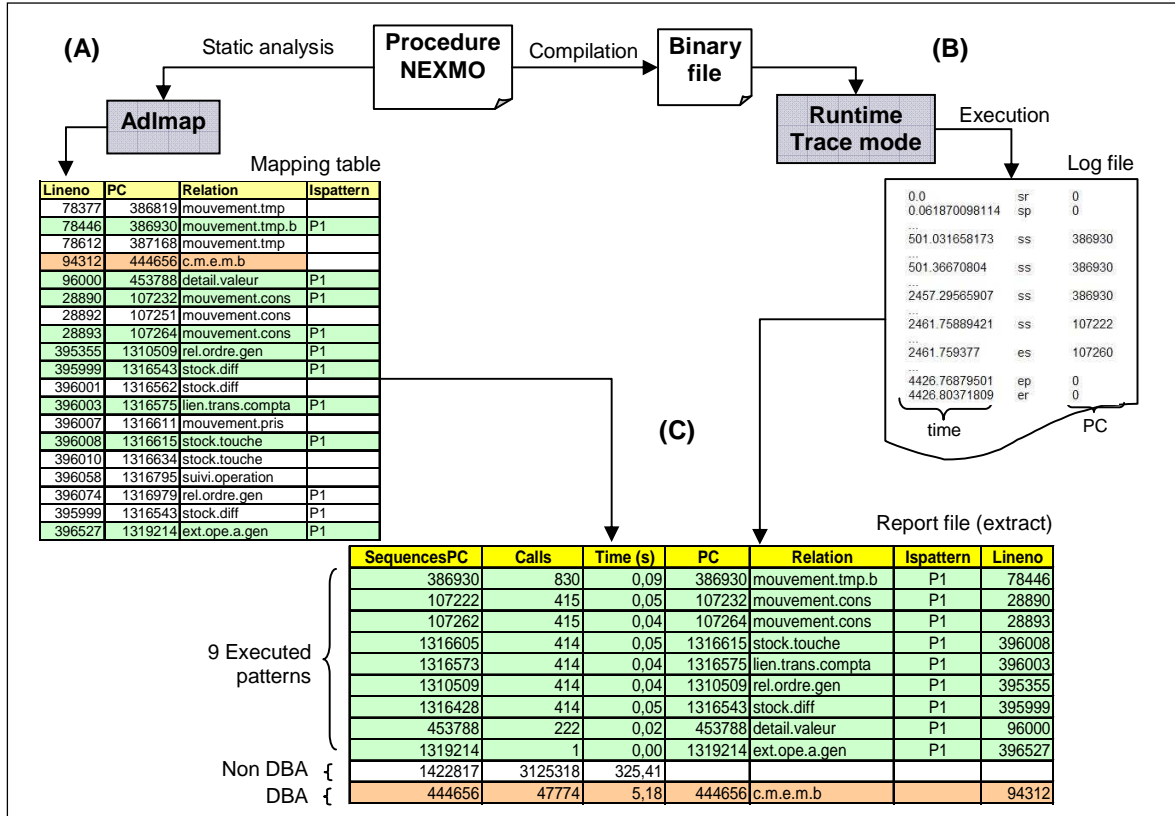
Figure 2. Pmonitor architecture

takes 0,09 seconds. We show in this report that the most expensive sequence of bytecode is a non DBA (non colored one) that takes 325,41 seconds and is called more than 3 million times. We also can see in the last row of the report, that a DBA takes 5,18 seconds and is called more than 47 000 times. We can expect it to be part of a pattern Adlmap is not get able to handle.

So this report allows developers to get a view on the behaviour of the suspected POC in comparison to the other instructions. This way they can know suspected POC involvement in performance decrease for a given execution context.

In the next Section we present our current work to improve our analysis tools and techniques.

4. Current work

Our tools were tested for 2 types of patterns (P1 and P2) and for a large variety of procedure executions. All tested executions do not show bad performances for our suspected POC, but show that they can point on new inexpected ADL patterns with bad performances.

For a better analysis of suspected POC, we need to extend our tools to the other POC of the list. Currently the maintainers have identified 6 more patterns which we will test in real costumer contexts. But this needs some preliminary work in order to first optimize our own analysis tools. We present here the current axes of work to apply our tools on real customer contexts and to reduce analysis execution time.

The static analysis with Adlmap is based on a research in an XML tree which is very expensive in system resources and also in time of analysis (19h07mn to obtain the mapping table for the procedure given in Figure 2). To reduce this time we are working on refactoring Adlmap with Python generator expressions [9] allowing the scanning of AST trees by small sub-trees instead of

loading and scanning the full XML tree. These generators allow Adlmap to use less memory (near 400Mo of RAM instead of 1Go) which speeds up considerably the analysis time.

The dynamic analysis takes even more time, for the example given in Figure 2 the Pmonitor report was obtained in 5h15mn. First, the trace mode increases the procedure execution time for procedure NEXMO of Figure 2. This increase in time of execution is due to Runtime that saves much information in log files. Second, Pmonitor spend more time to analyse a log file. To improve our dynamic analysis, we are working on a solution based on a full mapping between the instructions of source code and the sequences of bytecode. For doing this, a new static analysis tool, Adlmatch, is under development. Using this method will reduce the time to execute a procedure with trace mode activated as well as the time to analyse the log file with Pmonitor. This solution will also permit Pmonitor reports to be closer to the source code and by the way render them more concise for a better analysis.

5. Conclusion

Our work is on analysing POC instances performances. The static analysis identifies a very large number of suspected POC instances that, in real customer contexts, may behave differently according to some set like actual size of database or versions of source code and libraries. Combining static and dynamic analysis allows us to reduce the number of POC instances to analyse, thus to know those instances of POC that are really called and to have detailed information about the time taken by each one to execute. The final report produced by our tools allows maintainers to see which the instances which need a priority optimization are. These tools also allow identifying new patterns of ADL code that decrease execution performance. To identify the line number of these new patterns of code a full mapping between source code and bytecode is needed, which represent a first step for a future ADL debugger. Although these tools are already useable they need to be improved to make dynamic analysis more abstract and closer to the source code in order to improve the pattern analysis.

References

[1] Sungard Asset Arena Investment accounting (GP3). http://www.sungard.com/GP3

[2] Rim Chaabane, "Poor Performing Patterns of Code : Analysis and Detection". In proceedings of the 23rd IEEE International Conference on Software Maintenance in Doctoral Symposium, IEEE Computer Society, Paris (France), Oct. 2007, pp 501-502.

[3] A. Aggarwal and P. Jalote, "Integrating static and dynamic Analysis for Detecting vulnerabilities". In proceedings of the 30th IEEE annual int. COMPuter Software and Applications Conference, IEEE Computer Society, Chicago (USA), 2006, pp 343-350.

[4] M. Mock, M. Das, C. Chambers and S. Eggers, "Dynamic Point-To sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization". In the proceedings of the ACM SIGPLAN-SIGSOF workshop on PASTE, ACM, Utah (USA), 2001, pp 66-72.

[5] W. Binder, J. Hulaas, P. Moret, "Reengineering Standard Java Runtime Systems through Dynamic Bytecode Instrumentation". In the proceedings of the 7th IEEE int. Working Conf. on Source Code Analysis and Manipulation, IEEE Computer Society, Paris (France), Oct. 2007, pp 91-100.

[6] NICT team, "Documentation de Conception: Module Delia", version 4, Sungard Asset Arena Investment Accounting, Saint-Cloud (France), Copyright © 2003 by SunGard, 2003.

[7] NICT team, "Compilateur Spécifications Préalables (Option vers J2EE)", version 3, Sungard Asset Arena Investment Accounting, Saint-Cloud (France), Copyright © 2003 by SunGard, 2003.

[8] J. Clark, S. DeRose, "XML Path Language (Xpath)", Version 1.0, W3C, Massachusetts (USA), 1990. http://www.w3.org/TR/xpath

[9] Neil Schemenauer, Tim Peters and Magnus Lie Hetland, "Simple Generators", Version 2.2, Python Software Foundation, 2001. http://www.python.org/dev/peps/pep-0255/