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

PCODA'07

3rd International Workshop on

Program Comprehension through Dynamic Analysis

co-located with the 14th International Working Conference on Reverse Engineering (WCRE'07)

Technical report TUD-SERG-2007-022
Software Engineering Research Group
Delft University of Technology
The Netherlands

TU Delft
Delft University of Technology

Université Concordia University

b
UNIVERSITÄT BERN

Contents

Coping with large-sized execution traces

Modelling and knowledge extraction

Application of dynamic analysis

Program Chairs

Orla Greevy
Software Composition Group
Institut fur Informatik und angewandte Mathematik
University of Bern
Switzerland
greevy@iam.unibe.ch

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

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

Program Committee

Daniel Amyot
University of Ottawa, Canada
Bas Cornelissen
Delft University of Technology, The Netherlands
Wim De Pauw
IBM Research, USA
Serge Demeyer
University of Antwerp, Belgium
Arie van Deursen
Delft University of Technology, The Netherlands
Tudor Girba
University of Bern, Switzerland
Adrian Kuhn
University of Bern, Switzerland
Leon Moonen
Delft University of Technology, The Netherlands
Vassilios Tzerpos
York University, Canada

Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer

Chris Bennett
University of Victoria
cbennet@uvic.ca

Del Myers
University of Victoria
delmyers.cs@gmail.com

Margaret-Anne Storey
University of Victoria
mstorey@uvic.ca

Daniel German
University of Victoria
dmg@uvic.ca

Abstract

In this position paper, we survey and identify tool features that provide cognitive support for reverse engineering and program comprehension of very large reverse engineered sequence diagrams. From these features we synthesize user requirements for a sequence diagram viewer, to which we add system requirements such as memory and processing scalability. We briefly describe a pluggable sequence viewer that meets these requirements and discuss some open questions that we are currently exploring.

1. Introduction

Sequence diagrams are an aid to understanding system behaviour in the form of scenarios (the +1 view in Krutchen's 4+1 architectural view model [1]). While originally devised as a notation to capture scenarios during analysis and design, sequence diagrams can also aid understanding of existing software through visualization of execution call traces. Their power lies in their ability to represent selected behaviour at a suitable level of abstraction. As Kruchten notes [1], scenarios illustrate how elements from the four primary architectural views come together, highlighting the most important requirements of a system.

Reverse engineered sequence diagrams based on call traces are typically huge, sometimes running to thousands or even hundreds of thousands of calls. Designing tools that help the user cope with the size and complexity of such traces is a major problem. In addition, tools need to be able to physically handle such traces within the memory and processing constraints of typical computers. Approaches to address these issues include reducing information overload through pre-processing, support for presentation and user interaction, and techniques to deal with partial sequences. Automatically reducing arbitrary traces to a manageable size is probably not realistic. Consequently, effective user interaction that allows the user to reduce clutter, navigate the sequence, and focus on relevant details is critical.

This position paper is structured as follows. Section 2 provides a brief background to research on reverse engineered sequence diagrams. In Section 3 we describe presentation and interaction features of sequence diagram viewers as derived from the research literature. In Section 4 we identify related cognitive support requirements and categorize features in terms of the cognitive support they provide. In Section 5 we address system requirements (such as scalability and performance issues) that arise from the huge volume of information contained in a reverse engineered sequence trace. We end with a brief discussion of a sequence viewer we designed (called Zest) and propose future work.

2. Background

Sequence diagrams used in reverse engineering can be abstracted at various levels including statement, object, class, architectural, and inter-thread [2]. Statement-level diagrams include intra-procedural calls and typically make use of an extended notation that supports conditions, loops, and branches (e.g., see [3]). High-level sequence diagrams are typically used as an aid to program understanding (e.g., see the work of [4] on filtering utility methods to reduce trace complexity). Sequence diagrams can be created through static or dynamic analysis, the advantages of the latter being increased precision, control over inputs, as well as resolution of polymorphic behaviour and runtime binding in object-oriented languages [2]. Regardless of the creation method or abstraction level represented on a diagram, there is a need to cope with large amounts of reverse engineered data. This problem has been approached primarily in two ways: through pre-processing to reduce the size of the initial sequence, and through tool support for user interaction.

Pre-processing techniques include reduction at the source through data collection techniques and sampling [2], collapsing similar sequences using pattern matching (to identify loops, recursion, and non-contiguous repetitions), and automatic detection of utility functions (using fan-in/fan-out metrics) [5]. Other pre-processing techniques include removing abstract operation calls [5], hiding constructors and getters/setters [6], and limiting the depth of the call tree [5,6]. While pre-processing may be necessary to reduce the complexity of a sequence, considerable tool support is needed to help the user explore and understand the resulting diagram. We refer to this category of tool support as "cognitive support" - support that allows the user to offload some of their cognitive processing, such as their need to memorize details or to perform tedious calculations that the tool could do for them [8].

3. Presentation and Interaction Features

We divide sequence diagram user interface features into two categories 1) presentation or display facilities, and 2) features that allow the user to interact with and explore the diagram. We note that there may be overlap between presentation and interaction features, presentation often being both the result of interaction and a necessary precursor to it (e.g., highlighting and hiding could be considered interaction as well as presentation features). In the next subsections, we survey a number of reverse engineering tools that display sequence diagrams, summarizing their common presentation and interaction features.

3.1. Presentation

We first consider the presentation features these tools provide. Presentation concerns the layout of the diagram, as well as facilities for showing multiple views, hiding information and making the most effective use of animation and visual attributes.

3.1.1. Layout. Perhaps the most important presentation feature is the layout of sequence diagrams according to some notational standard. Many research tools use their own layout format or some variation on a standard format (e.g., UML 2.1), perhaps adding proprietary extensions to address a specific problem (e.g., how to capture conditional branches). *Scene* [9] produces sequence diagrams according to Rumbaugh's OMT notation [10]. *SCED* [11] uses its own UML-like notation that provides constructs for nested sub-scenarios and repetition. *TPTP* [12] also uses UML.

3.1.2. Multiple Linked Views. It is often necessary to provide multiple views [1] as well as an overview of an underlying model. Views can be of the same type (e.g., to allow comparison of different parts of a trace) or different types (e.g., linked class diagram and sequence diagram views). *Ovation* [13] adopts an approach to viewing sub-trees, whereby a subtree may be rendered using a number of alternative 'charts', including a static class list or a class communication graph. *SCED* supports sequence diagrams as well as state charts that show transitions within a selected object.

Linking these views so that they remain synchronized and can be easily navigated is another useful feature. *SEAT* [7] provides links between sequence and source code views. Similarly, *Scene* links between sequence views and static class diagrams or source code views. An overview is provided by many tools. *ISVis* [14] provides a two-window scenario view consisting of an information mural overview and a temporal message-flow diagram and *Scene* displays a summary call matrix view alongside a sequence view.

3.1.3. Highlighting. Highlighting a section of a sequence diagram is often the expected visible outcome of a user selection or search. Tools that support manual selection of components usually use highlighting to indicate selection. Highlighting can go beyond single components to show related objects or messages.

3.1.4. Hiding. Hiding selected information is commonly used for controlling complexity in sequence diagram tools. Hiding supports abstraction by removing detailed sub-message calls from below a parent call. Components can be hidden following pre-processing, a search (filtering), or a manual selection. *ISVis* supports hiding of classifiers within a subsystem, *SEAT* supports manual hiding, and *VET* [15] hides elements following filtering. Similarly, when grouping occurs (described in more detail below) the grouped elements are hidden 'under' a summary element [2]. When components are hidden as a result of filtering, it is important to indicate this so that the user can redisplay these components if required. There should also be an indication of why a set of components was hidden (e.g., as a result of loop detection or pruning of utility functions) [5]. The authors in [6] propose hiding null return values or abbreviating return values and parameter lists.

3.1.5. Visual Attributes. Colour and shape are useful ways to code additional information about a sequence. *Ovation* uses colour to differentiate objects and bevelling to indicate that components are grouped (hidden) under the bevelled component. *TPTP* uses colour to indicate the length of time spent inside a method execution.

3.1.6. Labels. Classifiers, messages, and return values are usually labelled. Occlusion and legibility are challenges when displaying larger sequences. Techniques to cope with this include hiding labels, replacing them with rectangles when zoomed out (e.g., as implemented by the *VET* tool), or using mouse hovers (e.g. as in *Ovation*).

3.1.7. Animation. Many tools support animation. This comes in two varieties – one that supports stepping through a sequence diagram, message by message, and another that uses animation to morph between diagram states to help the user maintain context. *Scene* supports single step animation between trace calls and *AVID* supports animation between component groupings.

3.2. Interaction

Interaction features allow the user to communicate with the tool while they navigate, query, and manipulate the sequence diagram to improve their understanding.

3.2.1. Selection. Manual selection of elements is a prerequisite for further interaction such as

manipulation, filtering, and slicing. This is supported by most tools.

3.2.2. Navigation. Rapid, simple movement between components (traversing the call tree) is important to usability [5] as is the ability to move between instances of the same type of pattern (e.g., subscenarios) in tools that support grouping of similar patterns (e.g., *SEAT*).

3.2.3. Focusing. User focusing has been identified as a problem when dealing with large traces [2]. The authors of the *Scene* tool note that it can be solved by techniques such as collapsing calls, partitioning sequences into manageable chunks, and selecting an object such that only related messages are shown. Single-step animation can also be used to focus on individual messages.

3.2.4. Zooming and Scrolling. Zooming and scrolling are standard techniques to cope with more information than can be legibly shown in a single window. *VET*, *Ovation*, *TPTP* and *Jinsight* [16] support zooming and scrolling [2].

3.2.5. Queries and Slicing. Queries identify and optionally filter information within a sequence. *Scenariographer* [17] supports both relational SQL and set-based SMQL (Software Modelling Query Language) queries on underlying structured data. *ISVis* allows exact, inexact, and wild-card searches. *VET* provides graphical support for selection of objects based on class and name as well as selection of methods by method type or time range. While these are more limited than language-based queries they provide a much simpler solution. Slicing can be performed on either objects or methods and is a specific form of query that selects only objects or methods related to the selected component (a slice through the sequence flow).

3.2.6. Grouping. Grouping can be the result of slicing or it can be done manually (e.g., *AVID*'s manual clustering and *Ovation's* flattening and underlaying). This is usually indicated by some sort of icon or visual attribute of the summary component (behind which grouped components are hidden). Grouping of objects will result in collapsing the sequence horizontally but may leave all messages visible (no vertical compaction). However, Cornelissen *et al.* [6] describe a technique to collapse lifelines that would eliminate calls between the merged objects. Grouping at the message level will hide messages called by the summary message (vertical compaction). Grouped items can also be annotated with a label (and optionally comments) describing the grouped abstraction. Riva and Rodriguez propose a technique to collapse packaging activations within these packages [18]. In addition to preprocessing to detect repeating patterns, interaction support can allow manual selection and collapsing of repeated patterns such as loops. *TPTP* supports grouping of life lines using pre-determined levels of abstraction (host, process, thread, class, and object), grouping of method calls, and arbitrary user-defined groupings.

3.2.7. Annotating. Annotating can be used for many purposes: to describe why components were grouped [4], to capture user understanding during exploration of a sequence diagram, and to provide waypoints [19] and messages to oneself and others when the diagram is to be shared. Few tools provide annotation mechanisms, but our initial experiences show this to be a useful feature.

3.2.8 Saving views. Saving views, either to share or to revisit, is also important when documenting a user's understanding of the diagram. A tool should be able to save the entire state of the visualization so it can be restored at a later time. Together with annotations, a saved view can tell a story about the diagram being visualized. In [5] the authors discuss the need to save both the original trace and the transformations that were applied to reduce its complexity, although saving a record of user interactions is not discussed.

4. Cognitive support requirements for tools that present very large sequence diagrams

Even after preprocessing, interacting with and understanding a reverse engineered sequence diagram can be a daunting task. Tools should provide cognitive support for the user to effectively and efficiently explore and interact with the sequence diagram view. Through our experiences developing and using customized sequence diagram views, and an extensive review of the literature, we have synthesized six high level cognitive support requirements that these tools should meet: (1) The tool needs to **present a diagram that is intuitive and coherent** to the end user. Since these diagrams are typically large and screen space is limited, the layouts need to use available visual attributes, such as position, size and color effectively and efficiently. (2) The tool should present **multiple perspectives** of the underlying sequence. It may be necessary to display a related static view (e.g., a class diagram) in addition to the dynamic sequence view, or some combination of the two. (3) The user needs to be able to **navigate the diagram** and explore a focus area or navigate to other elements on or off the screen. During navigation, the tool should help the user maintain context and help build and maintain a mental model of the navigated sequence. (4) Since sequence diagrams are typically very large, the user needs tool assistance as they **filter and drill down** on the salient features they wish to understand. Filtering can be supported through interactive querying techniques and presentation facilities for hiding information. (5)

Related to filtering, the user may need to **abstract details** in the viewer**.** This will remove visual details but maintain some visual cues on the abstractions created during the understanding process. (6) **Documenting the user's understanding** for future use or to share with colleagues is also an important feature.

Hamou-Lhadj *et al.* have also discussed high level user requirements, specifically requirements to support exploration, abstraction and filtering [2].

In Table 1 we map the tool features identified in Section 3 with these cognitive support requirements. The main advantage of this approach is that it organizes requirements into different groups, linking each tool feature with a clear cognitive support goal. This mapping may also be useful when comparing tools that might not have the same feature set, but attempt to solve similar problems. In particular, we have used this table to identify and prioritize the features of the Zest sequence diagram viewer (described below).

Cognitive Support Requirements	Presentation and Interaction Tool Features
1. Visualize diagram	• Layout (positioning) • Visual attributes such as Colour and shape • Labels
2. Multiple perspectives	• Multiple and linked views (e.g., overview views, split panes, static and dynamic views)
3. Navigating (while maintaining context)	• Selection • Highlighting • Focusing • Multiple and linked views • Zooming • Scrolling
4. Filtering	• Querying • Hiding information
5. Abstracting	• Grouping • Annotating
6. Documenting (e.g., for sharing)	• Annotating • Saving views

Table 1: Mapping presentation and interaction features to the cognitive support requirements for sequence diagram views

5. System requirements for coping with very large sequence diagrams

While computer systems continue to increase in processing and memory capabilities, large diagrams of any kind can be taxing on even very powerful machines. This leads to the question of whether it is even possible to render the diagrams that we would like to see. With the right optimizations, many of the interaction features previously described can reduce memory load and improve performance. Techniques such as lazy-loading of visual elements can be combined with grouping and filtering. However, trade-offs between performance and memory requirements must then be made and it is difficult to find an optimal solution.

Large diagrams require massive amounts of memory to render – sometimes more than is available with, for example, a Java virtual machine. Caching visible pages for the display can help, but it is not obvious if it is useful to display more information than a modern machine can handle at one time. The cognitive load on the human may be the limiting factor.

6. The Zest Sequence viewer

In the previous sections, we synthesized a list of features and requirements that are needed to build a general, scalable sequence diagram viewer that can be used across different applications. In order to explore the effectiveness and completeness of this list, we developed and are now evaluating the Zest Sequence Viewer (see Figure 1).

The Zest Sequence Viewer was designed from the outset to be easily pluggable into various end-user applications. The viewer is written in Java, using the SWT framework [20], so it can be plugged into any SWT application. We have explored using it as a viewer for visualizing dynamic program traces and for visualizing debug stack states. The Zest Sequence Viewer has been used to load upwards of a thousand objects, but trace size is limited by the memory required

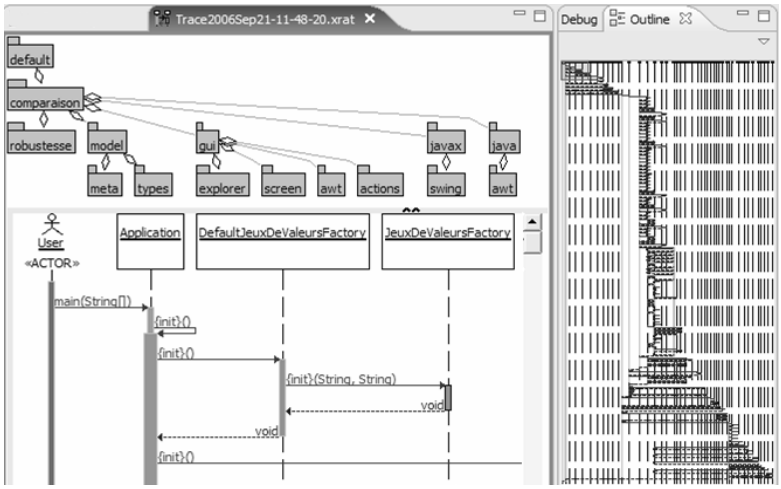


Figure 1: A portion of a sequence diagram in the Zest Sequence Viewer with an overview of the sequence on the right

to render large drawing areas. Such graphs can require hundreds of megabytes of memory, and may be larger than the Java virtual machine will allow.

7. Discussion

Our preliminary exploration has demonstrated the usefulness of the Zest Sequence Viewer. It has also helped us understand important requirements and tool features. However, more research must be done on the limitations of visualizing large sequences. A number of questions need to be resolved, e.g., what is the limiting factor: computer memory or human cognitive load? What kinds of visual inconsistencies can users cope with when displaying an incomplete sequence (e.g., changes in layout, hiding of visual elements)? Are humans able to understand and/or remember what elements have been hidden from the view? If not, what additional support can we provide for this? We are currently designing a case study that will involve observing professionals in their reverse engineering tasks using the Zest Sequence Viewer. We wish to observe their response to the viewer so that we can evaluate its usefulness and determine human factors in understanding sequence traces. We expect the results from this case study to further inform the cognitive support requirements for sequence diagram viewers.

Acknowledgments

We are grateful to Martin Salois, David Ouellet and Philippe Charland of Defence Research and Development Canada (DRDC) for their input into and review of this work. This work was funded by DRDC contract W7701-52677/001/QCL.

References

[1] P. Kruchten, “The 4+1 view model of architecture” *IEEE Software*, vol. 12, no. 6, Nov. 1995, pp. 42-50.

[2] A. Hamou-Lhadj and T. C. Lethbridge, “A survey of trace exploration tools and techniques”, in *Proceedings of the 2004 Conf. of the Centre For Advanced Studies on Collaborative Research*, IBM Press, 2004, pp. 42-55.

[3] A. Rountev, O. Volgin, and M. Reddoch, “Static control-flow analysis for reverse engineering of UML sequence diagrams”, *SIGSOFT Softw. Eng. Notes* 31, 1, Jan. 2006, pp. 96-102.

[4] 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 14th IEEE international Conference on Program Comprehension*, Washington, DC, 2006, pp. 181-190.

[5] A. Hamou-Lhadj, T.C. Lethbridge, and L. Fu, "Challenges and requirements for an effective trace exploration tool", in *Proceedings of the 12th IEEE International Workshop on Program Comprehension*, Washington, D.C., 2004, pp. 70- 78.

[6] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman, "Visualizing test-suites to aid in software understanding", in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, 2007, pp. 213-222.

[7] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu, “SEAT: a usable trace analysis tool”, in *Proceedings of the IEEE 13th international Workshop on Program Comprehension*, Washington, DC, 2005, pp. 157-160.

[8] A. Walenstein, “Cognitive support in software engineering tools: a distributed cognition framework”, Ph.D. dissertation, Simon Fraser University, B.C., Canada, 2002, p. 87.

[9] K. Koskimies and H. Mössenböck, “Scene: using scenario diagrams and active text for illustrating object-oriented programs“, *Proceedings of the IEEE 18th international Conference on Software Engineering* , Washington, DC, 1996, pp. 366-375.

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, 1990, Prentice Hall.

[11] T. Systä, “Understanding the behavior of Java programs”, *Proceedings of the Seventh Working Conference on Reverse Engineering* , IEEE Computer Society, Washington, DC, 2000, p. 214.

[12] The Eclipse Foundation, “Help—eclipse SDK: using UML2 trace interaction views”, http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.tptp.platform.doc.user/tasks/tesqanac.htm [Sept. 2007]

[13] W. DePauw, D. Lorenz, J. Vlissides, and M. Wegman, “Execution patterns in object-oriented visualization”, in *Proceedings Conference on Object-Oriented Technologies and Systems* , USENIX, 1998, p. 9.

[14] D. Jerding, J. Stasko, and T. Ball, “Visualising interactions in program executions”, in *Proceedings of the 19th International Conference on Software Engineering*, Boston, USA, 1997, pp. 360-370.

[15] M. McGavin, T. Wright, and S. Marshall, “Visualisations of execution traces (VET): an interactive plugin-based visualisation tool”, in *Proceedings of the 7th Austr-alasian User interface Conference–Vol. 50*, 2006, pp. 153-160.

[16] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, “Visualizing the execution of Java programs”, *Lecture Notes In Computer Science; Vol. 2269*, Springer-Verlag, London, 2001, pp.151-162.

[17] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos, “Scenariographer: a tool for reverse engineering class usage scenarios from method invocation sequences”, *21st IEEE Int. Conference on Software Maintenance*, 2005 pp. 155-164.

[18] C. Riva and J. V. Rodriguez, “Combining static and dynamic views for architecture reconstruction”, *Proceedings of the 6th European conference on Software Maintenance and Reengineering*, 2002, pp. 47-55.

[19] M. Storey, L. Cheng, I. Bull, and P. Rigby, “Shared waypoints and social tagging to support collaboration in software development”, *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, 2006, pp. 195-198.

[20] S. Northover and M. Wilson, *SWT: The Standard Widget Toolkit, Volume 1 (The Eclipse Series)*, New York: Addison-Wesley Professional, 2004.

Visualizing Similarities in Execution Traces

Bas Cornelissen
Delft University of Technology
The Netherlands
s.g.m.cornelissen@tudelft.nl

Leon Moonen
Delft University of Technology
The Netherlands
Leon.Moonen@computer.org

Abstract

The analysis of execution traces is a common practice in the context of software understanding. A major issue during this task is scalability, as the massive amounts of data often make the comprehension process difficult. A significant portion of this data overload can be attributed to repetitions that are caused by, for example, iterations in the software's source code.

In this position paper, we elaborate on a novel approach to visualize such repetitions. The idea is to compare an execution trace against itself and to visualize the matching events in a two-dimensional matrix, similar to related work in the field of code duplication detection. By revealing these similarities we hope to gain new insights into execution traces. We identify the potential purposes in facilitating the software understanding process and report on our findings so far.

1. Introduction

In the field of dynamic analysis, the post-mortem analysis of execution traces has been an active research topic for a long time. While traces can be rich in information and offer more accurate data than static analysis, they are typically rather large and not very human-readable. Significant efforts have been made to tackle this scalability issue (e.g., [11]) and many techniques and tools have been developed over time: in earlier work, for example, we proposed a set of techniques and a tool aimed at rendering execution traces more navigable [3].

One of the main causes for the information overflow in execution traces is the repetitive nature of certain event sequences. These sequences, which typically stem from loop constructs, consume huge amounts of space while offering little additional information to the reader. As a result, the development of summarization techniques has been conducted to the present day: Hamou-Lhadj et al. [6], for example, have addressed this issue by first identifying utility routines and consequently summarizing these routine. Kuhn et al. [8] represent traces as signals, which (to an extent) visualizes repetitions. De Pauw et al. [10] and Hamou-lhadj et al. [5] propose algorithms to identify similar subtrees in traces.

In this position paper, we propose to adopt a visualization technique to gain more insight into large execution traces. Taken from the domain of code duplication detection, we use a technique involving *similarity matrices* to effectively depict the repetitive nature of a trace. Our focus is not so much on pattern identification or summarization as it is on the *visualization* of (large) groups of recurrent events. We identify the potential purposes of such visualizations, and present the results of our preliminary experiments.

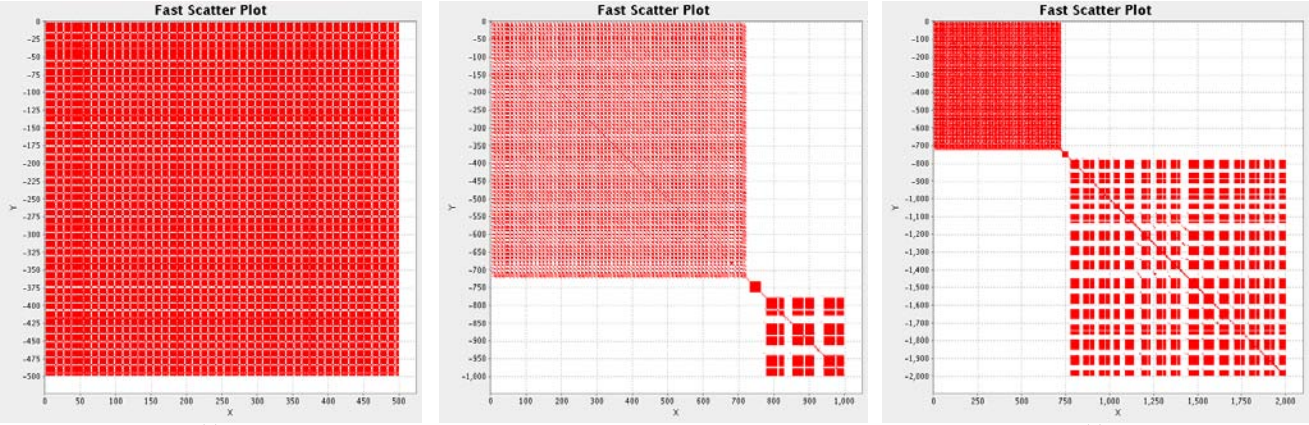
2. Approach

The technique that we propose to use is similar to the work of Ducasse et al. who, in the field of code duplication visualization, proposed to use a scatter plot to compare source code to itself [4]. Their strategy is to perform a line-by-line comparison using a simple string matching function and to visualize the matches as dots in a two-dimensional matrix. The resulting matrix shows diagonal lines in case of duplicate code fragments.

It is our opinion that the same method is applicable in the context of execution trace analysis. We propose to compare a trace against itself and to visualize the (partial) matches, with the intent of showing the patterns that are formed by recurrent call sequences, thus providing more insight into the program's behavior.

In particular, it is our belief that a matrix visualization of this comparison process has the following purposes:

- **Recurrent pattern visualization.** Similar to the case of duplicated code detection, we expect recurrent (sequences of) calls to show up in the visualization as clearly visible patterns. Insight into repetitive behavior makes it easier to grasp large amounts of trace information, and is a first step towards the development of new trace abstraction techniques that exploit these repetitions.



(a) (b) (c)

Figure 1. Similarity matrices of three partial Checkstyle traces: (a) 500 events, (b) 1000 events, and (c) 2000 events.

- **Execution phase visualization.** The interleaving between trace fragments that bear *different* degrees of similarity indicates execution phases and phase transitions [9, 3]. Initial knowledge of a program's phases of execution alleviates the trace comprehension process, and can be useful as a first step towards a more focused examination.

- **Polymorphism visualization.** By varying the matching criterion used to compare two events, we can also detect recurrent call sequences of which the calls differ only slightly, e.g., in case of late binding. Additionally, studying occurrences of late binding can provide information about the program's input and/or output [1].

3. Preliminary experiments

To assess the feasibility of our approach, we have performed a series of preliminary experiments in which we analyzed (parts of) an example trace. The subject system is CHECKSTYLE[1], an open-source tool that validates Java code. The program is instrumented, and executed according to a typical scenario during which all method and constructor calls are registered. The resulting execution trace contains roughly 32,000 events.

3.1. Visualizing repetitive behavior

Since our early prototype tools can not cope with traces of this magnitude – i.e., performing and visualizing 30,000 * 30,000 comparisons within a reasonable timeframe – in the first experiment we have processed three trace *fragments*. These fragments pertain to the first 500, 1,000, and 2,000 events of the CHECKSTYLE trace, and serve to provide a rough indication of the usefulness of our approach.

The matching criterion that we use is simple: we consider two calls to be similar if they involve the same caller, callee, signature, and runtime parameters.

In visualizing the resulting matches, we use the `FastScatterPlot` that is part of JFREECHART[2]. While this visualization solution is not particularly efficient, it is sufficiently fast for the initial experiments described here.

3.1.1. Results

Figure 1 shows the similarity matrices for each of the three trace fragments. The axes each symbolize the trace, whereas the red dots represent the data points, i.e., similarities between events according to our matching criterion. We now take a closer look at the matrices and attempt to clarify their contents.

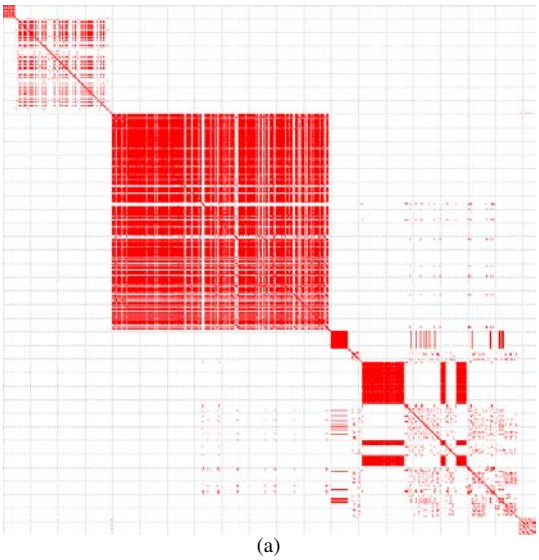
1. Judging by the density of the data points in Figure 1(a), the first 500 calls display a great degree of similarity. Upon closer inspection, we learn that a series of similar initialization tasks lies at the basis of these calls.

2. When considering the first 1,000 events (Figure 1(b)), we observe that the aforementioned stage ends somewhere between the 700th and 800th call. What follows almost instantly is another short repetition sequence. Finally, a very characteristic shape that looks like a grid of solid squares is shown at the lower right.

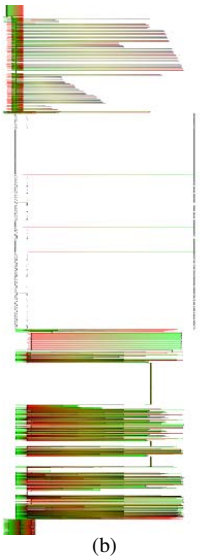
3. By broadening our perspective even more, we obtain the similarity matrix in Figure 1(c). The main conclu-

[1] Checkstyle 4.3, `http://checkstyle.sourceforge.net/`

[2] JFreeChart 1.0.6, `http://www.jfree.org/jfreechart/`



(a)



(b)

Figure 2. Visualizing the entire Checkstyle trace, using (a) a sampled similarity matrix, and (b) a massive sequence view.

sion here is that the characteristic pattern that we discovered in the previous step is continued all the way until the 2,000th event; close inspection reveals that a long series of exceptions is being created at this point.

3.2. Visualizing execution phases

In the second experiment, we focus on the visualization of execution phases. The idea is to process the entire CHECKSTYLE trace and to use the matrix visualization to recognize execution phases. To address the scalability issue that was described in the previous experiment, we choose to employ a *sampling* technique in the data generation process: we calculate datapoints for *sets* of calls. The rather straightforward criterion that we use for now is to consider the similarities between every n-th call with $n = 16$, thus reducing the 32,000 events in our trace to a dataset of 2,000 by 2,000 points.

Additionally, we employ an alternative visualization method that enables the visualization of execution phases. This second method is the *massive sequence view* from our earlier work [7, 3], which can easily cope with traces of up to 500,000 calls. Viewing both visualizations side by side allows for the comparison between the two techniques in the context of phase detection.

3.2.1. Results

The results of this experiment are shown in Figure 2, which shows the sampled matrix on the left and the massive sequence view on the right. Based on the views, we can draw several conclusions:

1. The matrix visualization clearly shows the various stages that are negotiated during the execution. While initially we had suspected the sampling criterion to be too strict, it turns out that many datapoints satisfy both the sampling and the matching criteria, resulting in a series of very distict shapes.

2. The datapoints are less dense in certain phases than they are in others. In the former case (i.e., less colored datapoints), either the matching criterion or the sampling criterion is satisfied less often, or a combination of both. Using a lower value for n would render the latter criterion more lenient, thus producing more colored datapoints.

3. The matrix view bears a striking resemblance to the massive sequence view, in that each phase in the one view can easily be identified in the other. In particular, the rather solid shapes in the matrix view correspond to the vertical lines in the massive sequence view. The similarity between the two types of views supports our claim that execution phases can be (largely) attributed to recurrent patterns, and that the visualization of these patterns is an effective tool in identifying those phases.

3.3. Visualizing polymorphism

As an example of visualizing occurrences of polymorphism, we analyzed a series of trace fragments while utilizing a different matching criterion. As a suitable criterion in this con-

text, we consider two calls to be polymorphic in case they have common callers and signatures, but different types of callees.

3.3.1. Results

One fragment in the CHECKSTYLE trace turned out to be particularly rich in such calls. The resulting similarity matrix of this fragment has been visualized in Figure 3, which shows six red "columns". A closer look at the execution trace in this interval points out that calls with two certain signatures are being invoked on a series of `Check` instances; Moreover, in the case of six certain checks, these calls lead to additional non-polymorphic interactions, which accounts for the five "interruptions" in between the aforementioned columns.

3.4. Observations

From the results in this Section, we can formulate a series of general observations.

Observation 1. The similarity matrices effectively show the degrees of repetition within trace fragments. The repetitiveness is reflected by the density of the data points: the less whitespace there is between the data points, the smaller and more repetitive the call sequences are.

Observation 2. The matrices allow for the recognition of *phases* in the program's execution. In the matrix that shows 2,000 events, we can already distinguish between various stages. By sampling the data points and showing all 32,000 events, the identification of CHECKSTYLE's major phases and their transitions requires little effort.

Observation 3. When using different matching criteria, various types of similarities can be highlighted in the execution trace. Our experiments point out that occurrences of polymorphism are an example type of events that can easily be distinguished in this manner.

Observation 4. With regard to data generation, our current prototype implementation is not very scalable: processing the entire trace requires 900,000,000 string comparisons. This task would greatly benefit from the use of more involved data structures.

Observation 5. Sampling the input data seems to be a promising technique: even the very straightforward sampling criterion with $n = 16$ in our experiment yields meaningful results.

Observation 6. The visualization aspect of our technique is not trivial, as (in case of no sampling) large traces will typically produce massive amounts of data points. For this reason, more effort is required to develop or reuse techniques that make optimal use of screen real-estate.

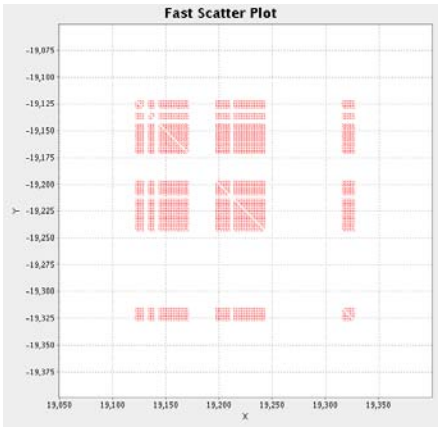


Figure 3. Matrix visualization of a trace fragment involving polymorphism.

4. Open Issues

In order for our technique to be applicable in practice, we need to address several important issues.

4.1. Matching criterion

In our preliminary experiments we have employed a visualization that only allows for one type of data point, i.e., one color. For this reason we have utilized a very strict matching criteron: either two calls match, or they do not match.

It would be interesting to use a criterion that is more lenient. As an example, one could consider assigning scores to partial matches (e.g., in case only the runtime parameters do not match) and using different colors for those data points.

4.2. Scalability

An important observation in our experiments was the lack of scalability. If we are to deal with real-life execution traces we can not resort to matching every single call, as this requires (1) too many calculations, and (2) too much screen real-estate.

4.2.1. Data generation

One of the potential solutions to the data generation problem is to consider blocks of calls, i.e., to group a number of calls according to some criterion and to compare these blocks. The difficult part is to come up with a suitable selection criterion: simply considering mutually exclusive blocks of fixed numbers of calls is dangerous, as it potentially separates repeating call sequences. To devise a selection criterion we will examine the role of *stack depths* during the execution, and investigate whether depth changes can offer hints in selecting suitable call blocks.

Another solution is the optimization of the comparison operations themselves. By calculating hashes for the trace events and storing these hashes in hash buckets, the comparison process becomes significantly faster as string comparisons are no longer necessary.

4.2.2. Sampling criterion

While the sampling experiment provided good results, we suspect that this technique would benefit from more elaborate sampling criteria. The criterion used so far involves the consideration of every *n*-th event; another could be to consider groups of *n* events and to calculate a mean value for each group, or to introduce a minimum threshold for the amount of events in a group that satisfies the matching criterion.

4.2.3. Visualization

In order to visualize the potentially large amounts of data points that result from the comparison process, we need abstraction techniques to visualize the information in a meaningful way. Various techniques from the domains of information visualization and computer graphics (e.g., mipmapping, interpolation, or event clustering) can be used to handle such larger visualizations.

5. Conclusions and Future Work

In this position paper we have elaborated on a technique to visualize similarities in program executions. By comparing an execution trace to itself on an event-by-event basis and by showing the matches in a similarity matrix, little effort is required from the viewer to detect repeated call sequences and to determine the degree of similarity in certain execution phases. Moreover, such visualizations can lead to a greater understanding of a program's execution phases and occurrences of polymorphism. We conducted preliminary experiments that pointed out the issues that are to be tackled for our technique to be useful in practice.

Having overcome these issues, we have several future directions for our research. As a first direction we seek to investigate the *value* of information on repeated sequences, e.g., to find out how information on such sequences can be utilized to (automatically) shorten traces. Among the example applications are the dynamically reconstructed sequence diagrams from our earlier work [2] which could be rendered more compact.

Secondly, we want to determine to which extent the matrix visualization allows for the detection of phases during the execution, as was done using an alternative visualization in earlier work [3]. As a potential solution to the scalability issue, we will investigate and propose suitable sampling criteria.

Finally, another application that we consider to be useful is the visualization of polymorphism. The matching criterion can be adjusted such that polymorphic calls are visualized, and the visualization of such occurrences may provide a deeper insight into the program's behavior and, by extension, its inputs and/or outputs.

Acknowledgments

This research is sponsored by NWO via the Jacquard Reconstructor project.

References

[1] T. Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Eng. Notes*, 24(6):216–234, 1999.

[2] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[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 *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[4] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. 3rd Int. Conf. on Software Maintenance (ICSM)*, pages 109–118. IEEE, 1999.

[5] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proc. 1st ICSE Int. Workshop on Dynamic analysis (WODA)*, pages 1–6, 2003.

[6] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006.

[7] D. Holten, B. Cornelissen, and J. J. van Wijk. Visualizing execution traces using hierarchical edge bundles. In *Proc. 4th Int. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 47–54. IEEE, 2007.

[8] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proc. 22nd Int. Conf. on Software Maintenance (ICSM)*, pages 320–329. IEEE, 2006.

[9] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proc. 3rd ICSE Int. Workshop on Dynamic analysis (WODA)*, pages 1–6, 2005. ACM SIGSOFT Sw. Eng. Notes 30(4).

[10] J. F. Morar W. De Pauw, S. Krasikov. Execution patterns for visualizing web services. In *Proc. Symposium on Software Visualization (SOFTVIS)*, pages 37–45. ACM, 2006.

[11] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.

Exposing Side Effects in Execution Traces

Adrian Lienhard, Tudor Gîrba, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Abstract

We need to understand the impact of side effects whenever changing complex object-oriented software systems. This can be difficult as side effects are at best implicit in static views of the software, and typically execution traces do not capture data flow between parts of the system. To solve this problem, we complement execution traces with dynamic object flow information. In our previous work we analyzed object flows between features and classes. In this paper, we use object flow information to analyze side effects in execution traces and to detect how future behavior in the trace is affected by it. Using a visualization, the developer can study how a selected part of the program accessed program state and what side effect its execution produced. Like this, the developer can investigate how a particular part of the program works without needing to understand the source code in detail. To illustrate our approach, we use a running example of writing unit tests for a legacy system.

1 Introduction

With object-oriented programs, the gap between static structure and runtime behavior is particularly large. Unlike pure functional languages where the entire flow of data is explicit, in object-oriented systems the flow of objects is not apparent from the source code. Through reference fields, objects may outlive the execution scope in which they are created and thus may influence behavior of another part of a system at a later point in time. This characteristic of object-oriented systems represents *side effects* on the program state. As this is a key characteristic of object-orientation, it is crucial when analyzing a program functionality, to take side effects into consideration.

While the concept of data flow has been widely studied in static analysis [11], it has attracted little interest in the field of dynamic analysis. Most approaches either analyze traces of method execution events [8, 22] or they analyze the interrelationships of objects on the heap [4, 10]. However, to detect side effects and how they affect future behavior in the trace, we need to also capture fine-grained information about the transfer of object references.

Side effects are difficult to understand, not only because of implicit information flow, but also because complex chains of method executions can hide where they are produced [20]. In this paper we explore how exposing side effects in execution traces can support a developer to better understand and to maintain an object-oriented system. Before making a change to complex object-oriented legacy system, a developer needs to identify and understand side effects produced by the behavior he intends to change, and the parts of the system are potentially affected by it.

To facilitate the detection of side effects we adopt our Object Flow Analysis technique [14]. We demonstrated in previous work the usefulness of this technique to identify dependencies between features [15] and to discover relationships between classes by analyzing how they exchange objects [13]. The use of Object Flow Analysis as presented in this paper takes a different angle. Our focus here is to relate object flows to method execution events to reason about side effects in object-oriented systems.

Motivating example. As a motivation for side effect analysis, we present an example use case where knowledge of side effects supports the production of tests. Writing regression tests for legacy systems is a crucial maintenance task [5]. Tests are used to assess if legacy behavior has been preserved after modifying the code. They also document reengineering efforts. However, the task of writing tests is nontrivial when there is a lack of internal knowledge of a legacy system [6].

Without prior knowledge of a system, a test writer needs to accomplish the following steps to produce a unit test:

- *Creation of a fixture.* This involves determining which objects should be initialized so that the behavior to be tested can be successfully executed.
- *Execution of a method under test.* Once the fixture is established, this just involves executing the method under test using the appropriate receiver and arguments.
- *Verification of the expected results.* We need to know which conditions to test, *i.e.*, what the expected side effect is and what the return value of the method is as a result of execution.

We propose a visualization to expose side effects, which serves as a blueprint to set up a minimal fixture and to verify the expected test results.

Outline. In Section 2 we introduce our approach and subsequently in Section 3 we illustrate how it can be applied to facilitate the generation of unit tests. We outline related work in Section 4 and we conclude in Section 5.

2 Approach

To analyze side effects, we complement execution traces with dynamic object flow information by tracing the transfer of object references (*i.e.*, a dynamic pointer analysis). With this additional behavioral information, we can detect for a selected part of an execution trace, the precise effect it had on the program state and which future behavior in the trace was affected by the resulting heap modifications. We consider the term *program state* to be limited to the scope of the application under analysis and the system classes it uses. We do not take changes outside this scope into consideration, *e.g.*, writing to a network socket or updating the display. Therefore, we refer to the *side effect* of some program behavior as the set of all heap modifications it produces.

We structure the discussion as follows. First we present our analysis of information flows in execution traces and how we detect side effects. Then, in Section 2.2 we describe how to expose the side effects. A visualization shows how program execution used and affected the program state. In Section 2.3 we present how side effects were propagated so as to influence behavior in the trace at a later point in time.

2.1 Detecting side effects

Typically, UML sequence diagrams are used to visualize execution traces (or parts of them) [7]. We base our analysis of side effects on an adaptation of a more scalable view introduced by De Pauw *et al.* [3], which was later also implemented in the Jinsight tool [4].

Figure 1 (left) illustrates a small portion of an execution trace represented by the experimental tool we built for side effect analysis. The trace is presented as a tree where the nodes represent method executions. The layout emphasizes the progression of time; messages that were executed later in time appear further to the right on the same line or further down than earlier ones. For a comparison with sequence diagrams we refer the reader to work by de Pauw *et al.* [3].

This visual representation emphasizes the underlying model of our approach — namely, to consider a method execution as including the transitively executed methods. We refer to a partial execution trace as a *sub-trace*. This corresponds to the call-return procedure abstraction of most programming languages. Figure 1 illustrates a selected sub-trace indicated by a rectangle in the execution trace.

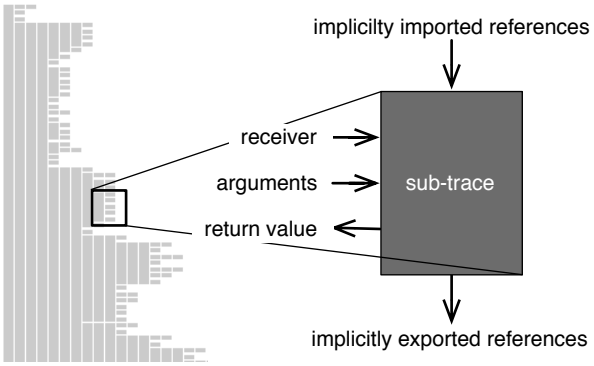


Figure 1. Flow of objects into and out of a sub-trace.

To contribute to the computation of a program, the method executions of a sub-trace must have some effect on information flow. A sub-trace defines an encapsulation boundary with respect to object references being passed in and out of it. The out going flows are represented by the returned value (of the first method) and all objects stored in fields. We refer to those flows as *exports*.

During execution, the methods of the sub-trace also use existing program state. Accessible objects are the receiver and the arguments (of the first method execution) and further objects obtained from fields and global variables. We refer to those in going flows as *imports* (see Table 1).

	import	export
explicit	receiver and arguments	return value
implicit	field/global read	field/global write

Table 1. Flows at the sub-trace boundary

The receiver, arguments and the return value are *explicitly* passed at the sub-trace boundary. However, the flows out of or in to fields or global variables are hidden in the methods of the sub-trace. Therefore, it is often complex to grasp those *implicit* flows when studying an object-oriented system.

The implicitly exported flows represent the *side effects* of the execution of the sub-trace on the program state. The implicitly imported flows show us which objects have been used to produce those effects.

The strategy we adopt for detecting the object flows described above is based on the concept of Object Flow Analysis. The core of this analysis is the notion of object *aliases* (*i.e.*, object references) as first class entities [14, 13], as shown in the Object Flow meta-model in Figure 2.

The Object Flow meta-model explicitly captures the fol-

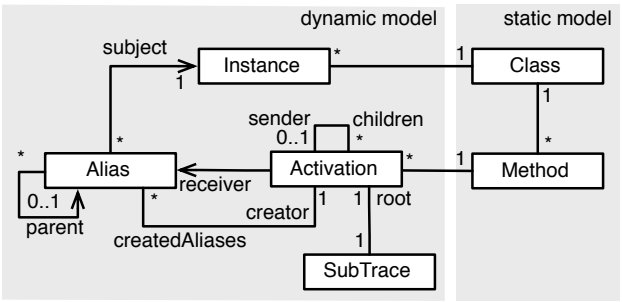


Figure 2. Core Object Flow meta-model.

lowing object references (represented as alias entities in the meta-model) created in a method execution (represented by the *activation* entity in the meta-model). An alias is created when an object is (1) instantiated, (2) stored in a field or global variable (including indexable fields), (3) read from a field or global variable, (4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution. The transfer of object references is modeled by the *parent-child* relationship between aliases of the same object.

Once we have established our Object Flow meta-model, we can detect the import and export sets of a sub-trace. The implicit object flows are defined as follows:

- The implicitly exported references are exactly those that are represented by the field and global *write* aliases that are created in activations of the sub-trace.
- The implicitly imported references are exactly those that are represented by the field and global *read* aliases that (i) are created in activations of the sub-trace and that (ii) do not have a parent write alias in the set of exported references.

The constraint (ii) makes use of the object flow information. That is, for each field read alias the corresponding field write alias is known (parent relationship). This constraint ensures that field references that are defined in the same sub-trace where they are used, are not considered as imports.

2.2 Exposing side effects

In the previous section we discussed how exported and imported object references are detected. The implicitly exported object references represent the side effects produced by a sub-trace, while the imported object references indicate which previously existing program state is used during its execution.

In this section we present our approach to expose side effects. Our experimental tool provides two interactive views, which are both illustrated in Figure 3. On the left side of Figure 3 the view with the execution trace is shown (as discussed previously). When clicking on a sub-trace in this view, a new window is opened (see Figure 3 right). We refer to the view it shows as the *side effect view*.

This view is similar to a UML object diagram [7] in that it shows objects and how they refer to each other. The key differences are: (1) it is scoped to the behavior of a selected sub-trace, and (2) it provides additional information based on the side effect analysis.

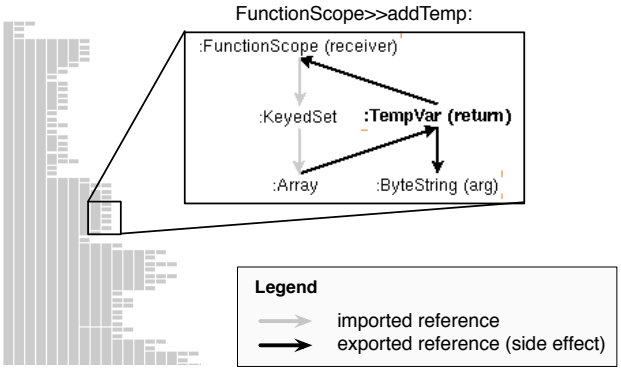


Figure 3. The side effect view of a selected part of the execution trace.

Figure 3 shows all objects being accessed (but not necessarily receiving messages) during the execution of a selected sub-trace. We annotate the class name of objects that have been explicitly passed into a sub-trace, *i.e.*, the receiver, the arguments, and the return value. We use regular typeface to indicate objects that existed before the execution of the sub-trace, whereas we use bold typeface to indicate objects that are instantiated during its execution.

An edge between objects indicates that one object has a field reference to another object. Gray edges indicate references that already existed before the sub-trace was run, *i.e.*, they refer to imported object references. If a gray edge is dashed this means that the reference is deleted during the execution of the sub-trace. Black edges indicate the references that are established during the execution of the sub-trace, *i.e.*, the ones that are exported. Thus, the black edges represent references that are the side effects of the sub-trace.

The main goal of this view is to (i) show which objects are used by the sub-trace, (ii) what side effects are produced on those objects, and (iii) how the objects refer to each other, *i.e.*, to make the reference paths between objects visible.

In Figure 3 we see, for example, the execution of a method addTemp: by an instance of FunctionScope (receiver). A ByteString is passed as argument. This execution

produces a side effect: a new TempVar instance is created (it is displayed with bold text), and the instance is not only returned but also stored in an already existing array. Another side effect is that the returned object is assigned a back reference to the receiver. Also, the object passed as argument, a ByteString, is stored in a field of the TempVar instance.

In our prototype, the name of a field (object reference) is displayed as a tooltip when moving the mouse over it.

2.3 Exposing the impact of side effects

In the previous section we discussed how we expose the side effects produced by the execution of a sub-trace. In this section we show which future behavior in the trace is affected by the side effect.

The side effects of a sub-trace are essentially the implicitly exported references (field or global stores). Other sub-traces, which occur later in the trace, may then import these references. The importing sub-traces may then proceed to further export the references. Therefore, to detect also method executions indirectly affected by the side effects of a sub-trace, we need to track how imported references are further propagated in the trace.

In the execution trace under analysis we highlight methods that are affected by a side effect. When a sub-trace is selected, we highlight all method executions that contain references originating from its exported references.

An execution trace with affected method activations is illustrated in the subsequent section (see highlighted methods in Figure 4), which exemplifies how the detection of side effects can be used to support the task of writing tests.

3 Case Study: Using the Side Effect View as a Test Blueprint

With our approach we make use of dynamic information captured from instrumented example runs of the system. The side effect view serves as a blueprint for writing tests by making explicit:

- The minimal fixture: only the gray objects and gray references are expected to exist before executing the method to be tested.
- What results to verify: the black objects and black references, which are produced as side effects of executing the method.

We motivate our work by presenting an example to illustrate how knowledge of side effects supports writing unit tests. The example is taken from an open-source Smalltalk bytecode compiler. The compiler works in three phases: (1) scanning and parsing, (2) translating the Abstract Syntax

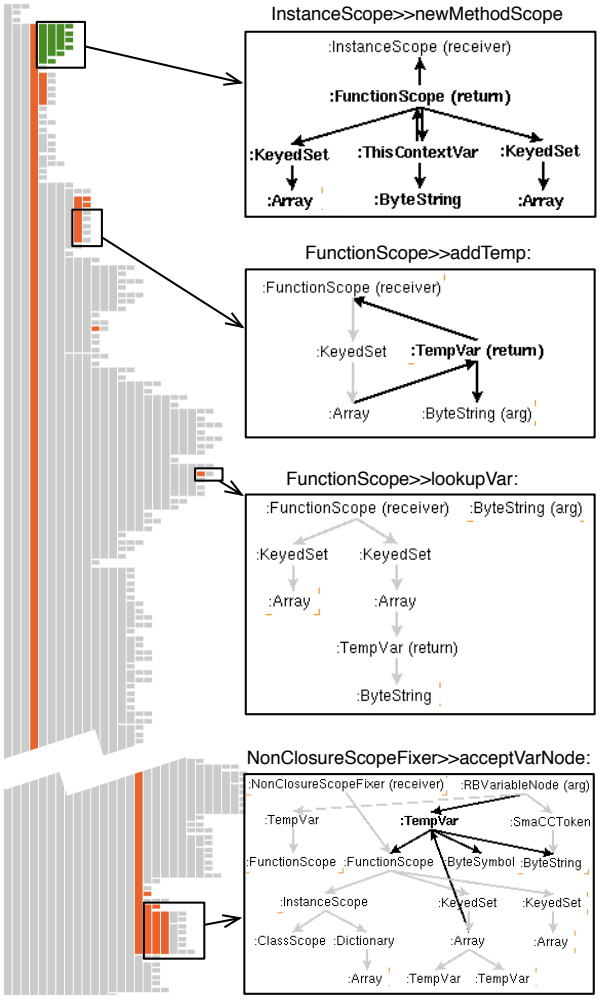


Figure 4. Side effect views serving as blueprints for writing tests.

Tree (AST) representation to the intermediate representation (IR), and (3) translating the IR to bytecode.

Let us assume we want to test the implementation of how variables are captured in the AST to IR transformation phase. Since variables are always defined in a specific scope (method, block closure, instance, or global scope), classes like InstanceScope or FunctionScope look like interesting classes to test. However, they are complex to understand without studying the source code in detail.

First, we identify in the source code the method InstanceScope≫newMethodScope, which looks promising as a starting point. Thus, we first query the trace for one of the executions of the method. Figure 4 on the top right shows the side effect view of an execution. On the top left the corresponding sub-trace is highlighted in green.

For the test we want to write, setting up the fixture only requires the creation of an InstanceScope (this is the only object that is used but not created in the sub-trace and there are no gray references). Then we can execute the method we want to test.

```
instance := InstanceScope new.
function := instance newMethodScope.
```

Next, we investigate the side effect view to determine what conditions we need to verify. First we want to assert that the return value actually is a function scope object. Then we check whether the function scope correctly references the instance scope as its outerScope (this is the name of the field, which in our prototype is obtained by a tooltip and hence is not shown in Figure 4). Both keyed sets, tempVars and capturedVars, are assumed to be empty. Also a new instance of ThisContextVar is created, which is stored in the function scope and has a back reference.

```
self assert: function class = FunctionScope.
self assert: function outerScope = instance.
self assert: function tempVars isEmpty.
self assert: function capturedVars isEmpty.

var := function thisContextVar.
self assert: var class = ThisContextVar.
self assert: var scope = function.
self assert: var name = 'thisContext'.
```

With the assertions above, we have tested all side effects that the method, together with the 9 methods it indirectly executes, is expected to produce. Now, what further tests can we write for this part of the system? We can answer this question by investigating the methods that are affected by the side effects of the method under test. In the execution trace, the affected methods are marked with orange. In our example (see Figure 4), we identify five locations in the trace where methods are affected, the last one being much later in time than the others. These method executions are examples of which other behavior uses the state that is changed as a result of running the method we are testing.

For instance, addTemp: is called on the function scope we created. We can now take its side effect view (see Figure 4) to write the next test. It shows that for the fixture the function scope as it is created in the previous test is sufficient. Additionally, we need the string 'x' as an argument. We can now test the expected side effects, for instance, that the function scope includes the new instance TempVar and that this instance correctly references the name of the temporary variable we passed as an argument.

```
...
var := function addTemp: 'x'.

self assert: var class = TempVar.
self assert: var name = 'x'.
self assert: var scope = function.
self assert: (function tempVars includes: var).
```

Along the same lines we can write tests for the remaining usage examples. For instance, to write a fixture for FunctionScope≫lookupVar:, we see that it depends on the TempVar produced as a side effect of the previous test. Therefore, we only need to add the following lines to it.

```
result := function lookupVar: 'x'.
self assert: result = var.
```

The method lookupVar: is special in that it produces no side effects (there are no bold instance names nor black references).

The last side effect view shown in Figure 4 is more complex than the previous ones. It is also an example for the deletion of a reference. The reference from the RBVariableNode to the TempVar instance is deleted (dashed arrow) and replaced by a reference to a newly instantiated TempVar object.

```
function := InstanceScope new newMethodScope.
block := function newFunctionScope.
var := block addTemp: 'x'.
node := (RBVariableNode named: 'x') binding: var.
fixer := NonClosureScopeFixer new scope: method.

fixer acceptVariableNode: node.

newVar := method lookupVar: 'x'.
self assert: newVar class = TempVar.
self deny: newVar = var.
self assert: node binding = newVar.
self assert: newVar scope = method.
```

4 Related work

Typically, dynamic analysis techniques focus on execution traces, which capture method execution events [3, 9, 21]. As dynamic analysis implies large amounts of data, much research effort has been concerned with the accessibility of large traces using filtering or summarization techniques [3, 9], or by identifying recurring execution patterns [12]. While those approaches mainly analyze method execution sequences, our approach additionally takes into account the object flow, and hence is capable to relate program execution and its effect on the program state.

Another dynamic analysis research area is concerned with the structure of object relationships. For instance, Tonella *et al.* extract the object diagram from test runs [19], Super-Jinsight visualizes object reference patterns to detect memory leaks [4], and the visualizations of ownership-trees proposed by Hill *et al.* show the encapsulation structure of objects [10]. To support debugging, tools like the GNU Data Display Debugger [23] visualize program state. The key difference to our approach is that we not only extract the object reference relationships, but in addition we detect how reference relationships are modified by a specific part of the program execution.

Dynamic data flow analysis is a method of analyzing the sequence of actions (define, reference, and undefine) on data at runtime. It has mainly been used for testing procedural programs, but has been extended to object-oriented programming languages as well [1, 2]. Since the goal of those approaches is to detect improper sequences on data access, they do not capture how objects are passed through the system, nor how read and write accesses relate to method executions. To the best of our knowledge, Object Flow Analysis is the only dynamic analysis approach that explicitly models object reference transfers.

In the area of static analysis, there is a large body of research on interprocedural side effect analysis. More recent research addresses the precision problem of static side effect analysis (or the analysis of pure methods) in object-oriented programs [16, 17, 18]. As static analysis does not take a concrete execution scenario into account, it provides a conservative view (which may even include infeasible execution paths of the program). Dynamic analysis on the other hand produces a precise under-approximation. Our approach makes use of this property by accurately detecting the side effects as they are produced during example runs of the program. This allows for directly relating side effects to where they occur in an execution trace. Another advantage of our approach is that it handles reflection, multi-threading, or dynamic code updates, which typically pose problems in static analysis.

5 Conclusions

In this paper we propose to expose side effects in execution traces through Object Flow Analysis. We use the dynamic object flows to define the side effect view, and we exemplify how we can use this information to guide the developer when writing tests, in particular in the case of legacy object-oriented systems.

As with most other dynamic analysis approaches, scalability is a potential limiting factor. Object Flow Analysis gathers both object references and method executions, thus it consumes about 2.5 times the space of conventional execution trace approaches [13].

Apart from the amount of data gathered, the side effect view is most vulnerable to large amount of data because it shows single objects and references between them. Our initial case studies indicate that also sub-traces of the size of several hundred method executions can be analyzed. However, the view does not scale for the analysis of truly large parts of the execution in which hundreds of objects are modified. We plan to tackle this problem by collapsing and summarizing parts of the object reference graph shown in the view. In this way, implementation details such as the internal structure of collections can be hidden to yield a more concise side effect view.

In our current studies we limit the analysis of program state to the application and the system library classes. However, side effects outside this scope, for instance, writing on a network socket or updating the display, are not captured. For future work we plan to extend the analysis to take also side effects into account that are outside this scope (*e.g.*, to capture how data in a RDBMS is affected).

Acknowledgments:. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Dynamic data flow analysis for Java programs. *Information & Software Technology*, 42(11):765–775, 2000.

[2] T. Y. Chen and C. K. Low. Dynamic data flow analysis for C++. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 22, Washington, DC, USA, 1995. IEEE Computer Society.

[3] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[4] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[7] M. Fowler. *UML Distilled*. Addison Wesley, 2003.

[8] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.

[9] 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 International Conference on Program Comprehension (ICPC 2006)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[10] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS '00*, June 2000.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[12] D. J. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of ICSE '97*, pages 360–370, 1997.

[13] A. Lienhard, S. Ducasse, and T. Gîrba. Object flow analysis — taking an object-centric view on dynamic analysis. In *International Conference on Dynamic Languages (2007)*, 2007. To appear.

[14] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[15] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC 2007)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[17] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 82–91, Washington, DC, USA, 2004. IEEE Computer Society.

[18] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[19] P. Tonella and A. Potrich. Static and dynamic c++ code analysis for the recovery of the object diagram. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

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

[21] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[22] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 329–338, Los Alamitos CA, Mar. 2004. IEEE Computer Society Press.

[23] A. Zeller and D. Lütkehaus. DDD – a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.

Applying Grammar Inference Principles to Dynamic Analysis

Neil Walkinshaw and Kirill Bogdanov
Department of Computer Science, The University of Sheffield, UK
E-mail: {n.walkinshaw, k.bogdanov}@dcs.shef.ac.uk

Abstract

Grammar inference and dynamic analysis share a number of similarities. They both try to infer rules that govern the behaviour of some unknown system from a sample of observations. Deriving general rules about program behaviour from dynamic analysis is difficult because it is virtually impossible to identify and supply a complete sample of necessary program executions. The problems that arise with incomplete input samples have been extensively investigated in the grammar inference community. This has resulted in a number of advances that have produced increasingly sophisticated solutions that are much more successful at accurately inferring grammars. This paper investigates the similarities and shows how many of these advances can be applied with similar effect to dynamic analysis problems.

1. Introduction

Techniques that are based upon dynamic program analysis are appealing because of their inherent precision. A dynamic technique is supplied with a collection of program traces (a trace records the state of a program throughout its execution), and this concrete information can be used as a basis to make precise conclusions about the program behaviour. However, these conclusions are only valid with respect to the set of supplied traces and do not necessarily generalise to every possible program execution [10].

Dynamic analysis results have to be interpreted with a degree of skepticism. They can only be regarded as representative of general program behaviour (regardless of input or environment) if the supplied set of program traces is 'complete', i.e. it provides a total coverage of program behaviour, as is the purpose of functional test sets. Depending on the complexity of the program, this set of necessary program traces can be extremely large. If the additional assumption is made that the developer has no prior familiarity with the program (which is probable in domains such as program comprehension), the precondition that she can provide a set of traces that is complete becomes unreasonable. Instead, dynamic analysis techniques are commonly provided with an incomplete set of traces, in the hope that they are sufficient to result in a model that is an approximation of general program behaviour.

This problem is not unique to dynamic analysis. Grammar inference is an example of another field that is subject to a similar weakness. Here the challenge is to identify the grammar of a language by analysing a sample of sentences that belong to (and optionally do not belong to) it. The sample of sentences can often be sparse, which means that the resulting grammar is inevitably only partial or even false. However, a substantial amount of research in grammar inference has focused on minimising this problem, and has largely achieved this by using simple solutions, such as the provision of negative strings as well as positive ones to avoid unsound results, and the use of oracles to guide inference by answering simple questions about system behaviour.

This paper looks at some of the similarities between grammar inference and dynamic analysis. It presents some of the established theoretical limits on solutions to grammar inference problems, which also apply to dynamic analysis techniques that infer models equivalent to deterministic finite automata (this is the case for a large proportion of dynamic analysis techniques). It also shows how certain principles that have been successful in increasing the reliability of inferred grammars can as easily be used to improve the soundness of dynamic analysis results.

2. Regular Grammar Inference and its Limits

This section introduces the grammar inference problem, provides an overview of some inherent limits on certain traditional approaches, and introduces some of the most successful recent solutions. It does not provide an in-depth overview of the underlying mechanics of grammar inference techniques. The purpose of this section is to provide a high-level view of some of the key insights that have led to the most substantial advances in the field. For a more comprehensive overview, there are recent authoritative surveys

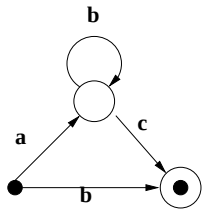


Figure 1. DFA that recognises the grammar corresponding to (ab*c)+b

by Parekh and Honavar [17] and De la Higuera [7].

2.1. The Grammar Inference Problem

The problem of grammar inference (also known as grammar induction) was formalised by Gold [11] in 1967. It is concerned with the identification of a language grammar from a finite sample of valid and (optionally) invalid sentences. Regular grammars have received the greatest amount of attention from the inference community. They are the simplest form of grammar in terms of Chomsky's hierarchy [4], and can be equivalently represented as a Deterministic Finite Automaton (DFA), where transitions are labelled by words and the automaton represents every valid ordering of those words in a sentence. The DFA representation is particularly appealing because [17]: (a) DFAs are easy to understand and (b) there exist several efficient DFA algorithms that are useful for a number of inference techniques (such as minimisation, determining the equivalence of two DFAs, and determining whether the language produced by one DFA is the super set of the language produced by another).

To illustrate the regular grammar inference problem, figure 1 illustrates the DFA for a simple regular grammar. Positive samples of the grammar correspond to sequences that would be accepted by the machine, and negative samples correspond to strings that would be rejected. As an example, a positive sample could consist of $\{abbbbc, b, abc\}$ and a negative sample could consist of $\{c, aba, ba\}$. Given that we do not have any prior knowledge of the structure of the DFA, a grammar inference technique would attempt to guess it, given only the sets of positive and negative sequences.

2.2. Traditional Solutions and their Limits

The regular grammar inference problem has been shown to be NP-complete in general [1, 12] and was even compared to problems such as breaking the RSA cryptosystem [13]. However, since Gold's initial research into the subject, a number of techniques have emerged that can correctly infer a grammar in polynomial time by placing restrictions on certain factors, such as making assumptions about the initial sample of sentences, or adding oracles that can provide additional information to the inference algorithm. Some of those advances have been prompted by a substantial body of theoretical work that establishes the inherent computational limitations on particular solutions to the grammar inference problem. As an example, Gold [11] proved that (in the worst case) an inference algorithm will require an infinite number of positive input sentences to determine the target grammar – thus establishing that any practical finite solution would require some quantity of negative sentences as well.

Subsequent research by Angluin [2] proved that, given a random initial sample of positive and negative sentences, the exact language can be inferred in polynomial time by using a 'minimally adequate teacher'. This teacher is expected to answer two types of queries:

(1) If the technique suggests sequences that do not already belong to the initial positive or negative samples, the oracle can state whether or not they are valid in the target grammar - these are called *membership queries*.

(2) If the technique produces a suggested grammar (i.e. arrives at a tentative DFA), the oracle can determine whether it is the target DFA or not - these are called *equivalence queries*.

Angluin's minimally adequate teacher, in particular the requirement to be able to answer equivalence queries, is not always practical for certain application domains. The ability to answer equivalence queries implies a substantial amount of prior knowledge of the grammar (or state machine). If, for example, the teacher is a human and the state machine consists of hundreds of states and transitions, the process of establishing whether the inferred hypothesis corresponds to the human's impression of a correct machine becomes unrealistic. Besides the practicalities of comparing two grammars, there is the additional problem that the approach presumes that the oracle already has a prior knowledge of the system.

An alternative approach that does not necessarily depend on an oracle is to adopt the *state merging* approach, originally developed by Trakhtenbrot and Barzdin [18]. These broadly operate by using the sets of positive and negative samples to produce an *augmented prefix tree acceptor* - a state machine that represents exactly the samples of positive and negative sequences provided. This is illustrated in figure 2. The inference process then consists of iteratively merging pairs of nodes in the tree, producing a more general language acceptor in the process. The challenge lies in *not* merging pairs of nodes that would represent different states in the target DFA, because this would produce a machine that is too general (unsound). A worklist of node pairs to be merged is created by traversing the prefix tree in a structured fashion (e.g. breadth-first search), and this list is processed

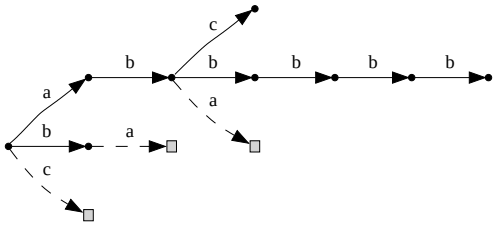


Figure 2. Augmented Prefix Tree Acceptor for positive sequences abc,abbbbb,b and negative sequences aba,c,ba

in order until a merge is found that is consistent with the supplied sample of sequences. The original state merging approach by Trakhtenbrot and Barzdin requires a complete sample of all strings up to a particular length to correctly infer the exact grammar in polynomial time.

Subsequent work by Oncina and Garcia [16] resulted in the RPNI (Regular Positive Negative Inference) algorithm. Instead of requiring an exhaustively complete set of samples up to a given length, the RPNI algorithm will exactly identify the target DFA in polynomial time, provided that the samples contain a *characteristic* sample of the target grammar. Informally, this means that, given the minimal DFA for some target language, a sample is characteristic if it contains positive sequences that cover every transition, as well as negative sequences that differentiate between any pair of non-equivalent states (see Dupont [8] for a formal definition).

Although the RPNI algorithm does not require an oracle, the requirement for a characteristic sample of the target grammar can be deemed just as impractical. Depending on the size and complexity of the target machine, the number of required sequences in the characteristic sample can be extremely large. As is the case with answering equivalence queries, constructing a characteristic sample requires a substantial amount of prior knowledge about the underlying system, ultimately rendering the requirement for a characteristic sample unrealistic for a large number of practical applications.

2.3. Pragmatic Solutions

The results for the techniques presented above are generally discouraging. Only if there exists an oracle that invests a substantial amount of effort, or the initial positive and negative sentence samples are sufficiently abundant that they happen to include an exhaustive or characteristic sample, is it possible to identify the exact target DFA in polynomial time. In the majority of practical applications these assumptions are unrealistic. However, recent techniques produce vastly improved results by relaxing these assumptions. They accept that the supplied set of samples might only be sparse, and apply various heuristics to identify states that might be equivalent. They accept that the resulting grammar might be only approximate, but ensure that it is at least an accurate approximation.

Traditional state-merging techniques fail when the supplied sample is sparse. They use simplistic techniques to construct work lists of possible state merges. If the provided sample of sentences is not complete or characteristic, there is not going to be enough information to prevent a wrong merge from happening, and there is a high probability that the algorithm will generate an erroneous machine as a result. At any point in the traditional merging algorithm, a worklist of candidate state merges is generated by simple strategies such as carrying out a breadth-first search of the successor tree to a node. This creates an arbitrary list of merge candidates that will only not be merged if there is a negative string to show that they are not equivalent. If this negative string is not present, a false merge occurs that in turn produces an incorrect state machine. Subsequent merges compound the error, resulting in a highly inaccurate final machine. Lang [14] reinforces this point by demonstrating empirically that for a sparse (incomplete / non-characteristic) sample of sequences, a traditional merging algorithm will only *approximately* identify the correct target machine if the size of the (random) sample is exponential in the size of the target machine.

A number of authors have realised that the key to improving the performance of state merging algorithms, particularly in the case of sparse samples, is to improve the way candidate pairs of nodes are selected. This has resulted in a number of heuristic approaches - the most popular of which is Price's Evidence-Driven State Merging (EDSM) algorithm [15]. The EDSM algorithm constructs a list of possible merges in a two-step process. The first step compares every possible pair of states and assigns a 'similarity' score to each pair, which indicates how much evidence there is in the sample to suggest that the pair are equivalent. This is computed by counting the number of overlapping outgoing labels in the prefix tree. An invalid merge (where a suffix from one state leads to an accept state but the identical suffix from the other state leads to a reject state) results in a negative score. The second step merges the highest scoring pair and the search process is restarted on the merged machine. Unlike the arbitrary merge sequences produced by traditional state-merging approaches, the merge order for the EDSM approach depends on the characteristics of the supplied samples. The EDSM approach was originally developed as a winning entry to the Abbadingo competition, which posed various grammar inference challenges for large random target machines with sparse data sets, where it excelled at identifying the largest class of machine (~512 states).

The EDSM algorithm can arrive at a close approximation to the target DFA even if the initial sample of sequences is not complete or characteristic. Nonetheless, the requirement for a relatively large number of samples is still inevitable. Dupont realised that it is easier to supply these samples iteratively, by employing an oracle. His Query-driven State Merging (QSM) approach [6, 9] is equipped with a question generation component that generates questions with the aim of guiding the oracle towards providing a (complete) characteristic sample. The algorithm uses the same heuristics as the EDSM algorithm to select states to merge, and for every merge the question generator produces a set of sequences that would have been valid *if* the merge was correct. These are posed to the oracle, and are then added to the appropriate positive or negative sample sets. Dupont shows that, through its targeted acquisition of necessary samples, the QSM technique arrives at an accurate solution with a much lower amount of initial sample sequences than the EDSM algorithm.

3. Augmenting Dynamic Analysis

There is an obvious overlap between the problems that are addressed in the grammar inference and dynamic analysis communities. Both attempt to derive facts or models from a finite sample of observations. (Regular) grammar inference aims solely to infer a DFA, whereas dynamic analysis has a broader range of applications. This paper argues that dynamic analysis has not taken advantage of the aforementioned advances in the field of grammar inference. If the purpose of dynamic analysis is to discover a model that is equivalent to a DFA, the analysis problem can be recast as a grammar inference problem, and the solution can take advantage of the many advances that have made regular grammar inference more tractable. Even if the target of dynamic analysis is not to produce a DFA-equivalent model, there are still many sufficiently general principles that can be applied regardless of the target model.

This analogy between grammar inference and dynamic analysis was first realised over 30 years ago when Biermann and Feldman [3] proposed their $k - tails$ state-merging algorithm that could generate state machines from sample executions. Since then a number of papers with similar aims have emerged that explore the analogy (e.g. Cook and Wolf [5]). However, these papers invariably restrict themselves to a rigid dynamic analysis framework that does not permit them to take advantage of some of the substantial advances that have taken place in grammar inference.

The conventional dynamic analysis framework assumes the provision of a single random selection of execution traces, from which the analysis technique must generate a hypothesis model in a single step. Invariably, the implicit assumption is made that this set of traces is in some sense 'representative'. Given that in the conventional case dynamic traces provide no negative data, it cannot be a characteristic sample (which necessarily contain negative sequences to distinguish between states). The only alternative requirement on the sample that can guarantee an exact result is that *every* positive sample up to a given length is provided, which in the case of dynamic analysis becomes impossible for any non-trivial system. Consequently, the result of a dynamic analysis is inevitably an approximation of the target system which, given only positive samples, is inherently prone to over-generalisation.

In practice, this means that the model that is presented by such an analysis will exactly show some set of rules that govern the provided set of program traces, but will not be able to make any useful inferences about the general system behaviour; it cannot infer impossible behaviour and, due to a combination of insufficient negative information and incomplete set of samples, any steps that attempt to infer behavioural rules beyond the supplied set of samples will probably be false. However, a number of the advances that have vastly improved the performance of grammar inference algorithms can be adapted to dynamic analysis. These are summarised below:

(1) Negative samples: Dynamic traces (positive samples) need to be accompanied by negative samples if the model inference is to be accurate and efficient. Negative samples can be obtained by a variety of techniques, such as proposing test sequences (if they fail / cannot be executed then they are negative) or by static analysis techniques which, although often too conservative to produce accurate positive input, are a powerful means of determining what is *impossible* in a software system, and are therefore a useful source of negative input.

(2) Accuracy wrt. incomplete traces: It can often be analytically shown that a dynamic analysis technique will produce an accurate model of the target system *if* the provided set of traces is complete. However, the ability of a dynamic analysis technique to cope with incomplete sets of traces is rarely evaluated. It is only through such empirical evaluation (often in the context of competitions such as the Abbadingo competition [15]), that real progress has been made in the field of grammar inference. These have produced the advanced heuristic techniques, such as EDSM, that excel in the average case, by making the most out of the sparse samples that they are given.

(3) Active dynamic analysis: Dynamic analysis, particularly in the context of program comprehension, is not necessarily restricted to passive techniques (which expect a single initial set of traces and produce a result in a single step). Oracles are available that can iteratively guide the analysis process, both in the form of the human program developer, as well as the software system itself. In the field of program comprehension, where it is unreasonable to expect

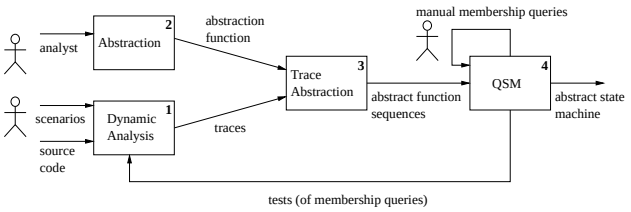


Figure 3. Combining dynamic analysis with QSM

the developer to know much about the system, queries about the system can be formulated as system tests instead (i.e. "Is the sequence of calls *xyz* possible?"). This removes the necessity that the initial set of traces is complete, because the set of samples can be grown iteratively, by guiding the oracle(s) to providing the further missing samples.

4. Small Example: Reverse Engineering State Machines

The authors are currently applying the principles mentioned above to the process of reverse engineering state machines from software [19]. The ability to reliably reverse engineer a state machine from a software system is particularly valuable for a range of development and maintenenace tasks such as model-based testing, documentation, and comprehension of system behaviour. The aim is to generate the necessary program traces to produce a state machine that is as accurate as possible.

The problem of reverse engineering state machines by dynamic analysis is an obvious application for regular grammar inference techniques because the target model is a state machine in each case. Our approach is based upon Dupont's QSM grammar inference technique [6, 9]. QSM espouses the three features mentioned in the previous section; it can accept negative samples, is relatively accurate when provided with incomplete samples, and is active (i.e. will try to obtain more information if it is missing from the initial set of input samples).

Figure 3 shows how we integrate QSM into the reverse engineering process. Essentially, the developer develops a set of mappings from sequences of particular method invocations in the traces to abstract functions. Thus a low-level dynamic trace can be lifted to a series of high-level functions (e.g. <open_file, enter_text, save_file>). Each trace is made into a string of abstract functions and fed into the QSM algorithm. If more information is required, it generates a question, in the form of a sequence of abstract functions that might be valid for the hypothesised machine (e.g. <open_file, close_file, enter_text?>). The question can either be answered directly by the user, *or* automatically, i.e. the sequence can be executed as a system test. Either way, if unsuccessful the new negative string is fed back into the QSM algorithm and the process repeats until no further questions are generated.

The benefits of active dynamic analysis are manifold. If queries are answered solely by the developer, the questioning process fosters a greater understanding of how the system works and forces them to consider aspects of system behaviour that they might not have envisaged. If, instead of the developer, the software system itself is used as an oracle (i.e. queries are posed as system tests), it is straightforward to simply supply new traces that do or do not correspond to the questions. Given a suitable test harness this process could be entirely automated, however at the moment our implementation takes manually generated traces.

Our initial evaluation of the technique shows that the approach is reasonably scalable in terms of number of questions generated and the accuracy of the final model. For specific results the reader is referred to our paper [19]. What is important to point out in the context of this paper however is the fact that the grammar inference evaluation methods provide a well established means to establish how accurate and scalable a given technique is. Grammar inference techniques are usually evaluated by generating a collection of random machines and an accompanying set of random paths across these machines. The technique in question can be run with respect to these different sets of paths of varying degrees of sparsity. This enables the precise quantification of the accuracy and scalability of the technique, with respect to increasingly populated samples of inputs. This evaluation approach is more systematic and accurate than common evaluations of dynamic analysis techniques, which are rarely evaluated with respect to the accuracy or completeness of the final model.

5. Conclusions

Many of the problems faced by dynamic analysis and grammar inference are similar. In order to construct a sound model of system behaviour, current dynamic analysis techniques place unrealistic requirements on the initial set of traces which, when unsatisfied, result in a model that can be highly inaccurate. However, by recasting the problem of inferring a model as a grammar inference problem, there are a number of straightforward techniques that can be adopted to substantially improve the efficiency and reliability of dynamic analysis results.

The domain of software analysis presents a number of advantages for inference techniques that need to be investigated. Static analysis presents an abundance of reliable negative information. Whereas grammar inference traditionally relies solely on human oracles, software engineering tools such as automated testing frameworks and model

checkers can be used as oracles alongside the developer to answer queries about hypothetical models. We have applied grammar inference techniques to the problem of reverse-engineering abstract state machines, and are currently investigating the use of static analysis (in the form of call graphs) to increase the amount of negative information about sequences of methods that are definitely impossible, in order to improve the efficiency of our current technique.

References

[1] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39:337–350, 1978.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[3] A. W. Biermann and J. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers*, 21:592–597, 1972.

[4] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[5] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, 1998.

[6] C. Damas, B. Lambeau, P. Dupont, and A. van Lamsweerde. Generating annotated behavior models from end-user scenarios. *IEEE Transactions on Software Engineering*, 31(12):1056–1073, 2005.

[7] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[8] P. Dupont. Incremental regular inference. In *International Colloquium on Grammatical Inference and Applications (ICGI'06)*, 1996.

[9] P. Dupont, B. Lambeau, C. Damas, and A. van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 2007. to appear.

[10] M. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proceedings of the International Workshop on Dynamic Analysis (WODA'03)*, 2003.

[11] E. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[12] E. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[13] M. Kearns and L.Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41, 1994.

[14] K. Lang. Random DFA's can be approximately learned from sparse uniform examples. In *COLT*, pages 45–52, 1992.

[15] K. Lang, B. Pearlmutter, and R. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference; 4th International Colloquium, ICGI-98*, volume 1433 of *LNCS/LNAI*, pages 1–12. Springer, 1998.

[16] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, volume 1, pages 49–61. 1992.

[17] R. Parekh and V. Honavar. *The Handbook of Natural Language Processing*, chapter Grammar Inference, Automata Induction and Language Acquisition, pages 727–764. 2000.

[18] B. Trakhtenbrot and Y. Barzdin. *Finite Automata, Behavior and Synthesis*. North Holland, Amsterdam, 1973.

[19] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*, Vancouver, October 2007. IEEE.

Mining Temporal Rules from Program Execution Traces

David Lo[†] and Siau-Cheng Khoo[†] and Chao Liu[‡]
[†]Department of Computer Science, National University of Singapore
[‡]Department of Computer Science, University of Illinois at Urbana-Champaign
{dlo,khoosc}@comp.nus.edu.sg, chaoliu@cs.uiuc.edu

Abstract

Software evolution incurs difficulties in program comprehension and software verification, and hence increases the cost of software maintenance. In this study, we propose a novel technique, to mine from program execution traces a sound and complete set of statistically significant temporal rules of arbitrary lengths. The extracted temporal rules reveal invariants that the program observes, and will consequently guide developers to understand the program behaviors, and facilitate all downstream applications like verifications. Different from previous studies that are restricted to mining two-event rules (e.g., $\langle lock \rangle \rightarrow \langle unlock \rangle$), our algorithm discovers rules of arbitrary lengths. Furthermore, in order to facilitate downstream applications, we represent the mined rules as temporal logic, so that existing model checkers or other formal analysis toolkits can readily consume our mining results. We performed case studies on JBoss Application Server (JBoss AS) and a buggy Concurrent Versions System (CVS) application, and the result clearly demonstrates the usefulness of our technique in recovering underlying program designs and detecting bugs.

1 Introduction

Software changes throughout its lifespan. Software maintenance deals with the management of such changes, ensuring that the software remains correct while features are added or removed. Maintenance cost can contribute up to 60%-80% of software cost [4]. A challenge to software maintenance is to keep documented specifications accurate and updated as program changes. Outdated specifications cause difficulties in program comprehension which account for up to 50% of program maintenance cost [4].

In order to ensure software correctness, model checking [5] has been proposed and shown useful in many cases. It accepts a model, often automatically constructed from the code, and a set of formal properties to check. However, the difficulty in formulating a set of formal properties has long been a barrier to its wide-spread application [2].

Addressing the above problems, there is a need for techniques to automatically mine formal specifications from program as it changes over time. Employing these techniques ensures specifications remain updated; also it provides a set of properties to be verified via formal verification tools like model checking. This family of techniques is commonly referred to as "specification mining" [2].

There have been a number of studies on specification mining, which relate to either program comprehension (*e.g.*, [20, 6, 15]) or verification (*e.g.*, [2, 22]). Most specification mining algorithms extract specifications in the form of an automaton (*e.g.*, [15, 2, 20, 6]) or two-event rules (*e.g.*, [22]). While a mined automaton expresses a global picture of a software specification, mined rules break this into smaller parts each expressing a program property which is easily understood. A mined automaton might be too complex to be comprehended manually. Also, methods mining automaton-based specifications from traces use automaton learners which suffer from the issue of *overgeneralization* (see [14, 15]), but this is not the case with mining rules. On the other hand, existing work on mining rules only mines two-event rules (*e.g.*, $\langle lock \rangle \rightarrow \langle unlock \rangle$) which are limited in their ability to express complex temporal properties.

The focus of this study is to automatically discover rules *of arbitrary lengths* from program execution traces. A trace can be viewed as a series of events, with each event corresponding to a method which is called when a program is executed. A multi-event rule is denoted by $ES_{pre} \rightarrow ES_{post}$, where ES_{pre} and ES_{post} are the premise/pre-condition and the consequent/post-condition, respectively. This rule means that "Whenever a series of events ES_{pre} occurs, eventually another series of events ES_{post} also occur."

The above rules can be expressed in temporal logic, and belong to two of the most used families of temporal logic expressions for verification (*i.e.*, response and chain-response) [7]. Examples of such rules include: *(Resource Locking Protocol)* Whenever a lock is acquired, eventually it is released; *(Internet Banking)* Whenever a connection to a bank server is made and an authentication is completed

and one transfers money, eventually money is transferred and a receipt is displayed.

From traces, many rules can be inferred, but not all are important. We therefore utilize the concept of support and confidence from data mining [8] to identify important rules. Rules satisfying user-defined thresholds of minimum support and confidence are referred to as *statistically significant*. A *non-redundant* set (see Section 3) of *statistically significant* rules are the output of our mining algorithm.

As with any program analysis tool, soundness and completeness are desirable goals to have [19]. Our algorithm is *sound* as all mined rules are statistically significant. It is *complete* as all statistically significant rules of the form $ES_{pre} \rightarrow ES_{post}$ are mined or represented.

We performed a case study on the transaction component of JBoss Application Server. The result shows the usefulness of our mining technique in recovering the underlying protocols that the code obeys, thus aiding program comprehension. Also, another case study has been performed on a buggy CVS application built on top of the Jakarta Commons Net [3] adapted from the one studied in [15, 14]. The result highlights the usefulness of our technique in mining bug-revealing properties, thus aiding program verification.

The rest of this paper is organized as follows. We first introduce the semantics of discovered rules in Section 2, and discuss the challenges of rule mining and our proposed solution in Section 3. Section 4 reports on the case studies. With related work discussed in Section 5, Section 7 concludes this study.

2 Semantics of Mined Rules

In this section, we discuss the semantics of mined rules and the computation of support and confidence values.

Temporal Logic Semantic. Our mined rules can be expressed in Linear Temporal Logic (LTL) [5]. There are a number of LTL operators, among which, we are interested in the operators 'G','F' and 'X'. The operator 'G' specifies that *globally* at every point in time a certain property holds. The operator 'F' specifies that either a property holds at that point in time or *finally (eventually)* it holds. The operator 'X' specifies that a property holds at the *next* event. Let us consider two examples of LTL expressions below.

$F(unlock)$

Meaning: *Eventually* unlock is called

$G(lock \rightarrow XF(unlock))$

Meaning: *Globally* whenever lock is called, then from the *next* event onwards, *eventually* unlock is called

Each of our mined rules states: whenever a series of premise events occurs eventually a series of consequent events also occurs. A mined rule denoted as $pre \rightarrow post$, can be mapped to its corresponding LTL expression. Examples of such correspondences are shown in the table below.

Notation	LTL Notation
$a \rightarrow b$	$G(a \rightarrow XFb)$
$\langle a,b \rangle \rightarrow \langle c,d \rangle$	$G(a \rightarrow XG(b \rightarrow XF(c \wedge XFd)))$

The set of LTL rules minable by our technique can be represented in the Backus-Naur Form (BNF) as follows:

$$\begin{aligned} rules &:= G(pre \rightarrow post) \\ pre &:= event | event \rightarrow XG(pre) \\ post &:= XF(event) | XF(event \wedge XF(post)) \end{aligned}$$

By simple transformations, the mined rules can also be expressed in Computational Tree Logic (CTL) [5] and probabilistic CTL [9].

Support and Confidence. There are many possible rules, and we need to identify important ones. Statistics, such as support and confidence, adapted from data mining [8], can be used to distinguish important ones: *(Support)* The *number of traces* exhibiting the premise of the rule; *(Confidence)* The likelihood of the rule's premise being followed by its consequent in the traces.

The meaning of the above is best illustrated by an example. Consider the following set of simplified traces:

Trace 1	*lock,use,use,unlock,lock,use*
Trace 2	*lock,unlock,lock,unlock*

Let us consider the rule $lock \rightarrow unlock$. Its support value is two, as all two traces exhibit the premise of the rule. Its confidence is 0.75, as 75% of the time (3 out of 4) $lock$ is followed by $unlock$. Formal definitions of support and confidence are provided in the technical report [17].

Our technique will only output rules satisfying a user-defined thresholds of minimum support and confidence. Rules satisfying these thresholds are referred to as being *statistically significant*.

3 Mining Algorithm

This section discusses challenges of mining a sound and complete set of multi-event LTL rules, and presents our solution and mining algorithm.

Challenges and Solutions. The complexity of mining multi-event temporal rules is potentially exponential to the length of the longest trace in the trace-set. A naive approach is to check each possible rule of length two up to the maximum length of the traces. This simply does not work because a set of traces with a maximum length of 100 and containing 10 unique events will require 10^{100} checks.

To address the above challenge, we utilize an effective search space pruning strategy. In particular, the following 'apriori' properties are used to prune non statistically-significant rules from the search space:

1. If a rule $evs_P \rightarrow evs_C$ doesn't satisfy the support threshold, neither does any rule $evs_Q \rightarrow evs_C$ where evs_Q is a super-sequence of evs_P.
2. If a rule $evs_P \rightarrow evs_C$ doesn't satisfy the confidence threshold, neither does any rule $evs_P \rightarrow evs_D$ where evs_D is a super-sequence of evs_C.

Furthermore, we notice that many rules are redundant. A rule R_X is redundant if there exists another mined rule R_Y where:

1 The concatenation of R_X's premise and consequent is a proper subsequence of the concatenation of those of R_Y. Otherwise, if the concatenations are the same, R_X has a longer premise.
2 Both R_X and R_Y have the same support and confidence values.

To illustrate redundant rules, consider the following set of rules describing an Automated Teller Machine (ATM):

R1	*accept_card* → *enter_pin,display_goodbye,eject_card*
R2	*accept_card* → *enter_pin*
R3	*accept_card* → *display_goodbye*
R4	*accept_card* → *enter_pin,eject_card*
R5	*accept_card* → *display_goodbye,eject_card*

If the above rules have the same support and confidence values, rules R2-R5 are redundant since they are represented by rule R1. To keep the number of mined rules manageable, we remove redundant rules. This can drastically reduce the number of reported rules. Our performance study on data mining benchmark datasets shows the number of rules is reduced by more than 1,000 times lesser when *redundant* rules are removed – see our technical report [17].

Without the application of the 'apriori' properties and the removal of redundant rules, the case studies considered are infeasible as the naive approach requires an exponential number of checks. A different 'apriori' property and redundancy identification have enabled practical use of pattern mining which would otherwise require exponential running time [8, 21].

Algorithm Sketch. Inputs of our mining algorithm comprise a set of traces and the support and confidence thresholds. The output of our algorithm is a set of rules, each of which expresses: whenever a series of events occurs at a point in time (*i.e.*, temporal point), another series of events will eventually occurs. To generate these rules, we need to identify interesting temporal points, and for such points, note what series of events are likely to occur next. Our mining process is composed of three steps – for full details see [17]:

Step 1 Mine a set of premises where each has a support value greater than the minimum support threshold.

Step 2 For each premise pre, do the following:
- **a** Find all temporal points where the premise pre occurs
- **b** Extract all rules of the form $pre \rightarrow post$, where the consequence $post$ happens with the likelihood greater than or equal to the minimum confidence threshold.

Step 3 Remove remaining redundant rules.

We perform an aggressive pruning strategy to remove redundant rules. Sub search spaces containing redundant rules are identified "early" and pruned. Rather than generating *all* statistically significant rules and removing redundant ones at Step 3 (*i.e.*, "late" pruning), we *avoid* generating redundant rules in the first place (*i.e.*, "early" pruning). At step 3, we only remove the remaining redundant rules not identified by our aggressive pruning strategy. "Early" pruning of redundant rules greatly improves the performance of our algorithm. In our performance study, unless redundant rules are pruned "early", several experiments on a real-life benchmark dataset at various minimum support and confidence thresholds are practically infeasible – see our technical report [17].

At the end of the above process, a complete set of non-redundant multi-event LTL rules of the form $ES_{pre} \rightarrow ES_{post}$ satisfying the support and confidence thresholds are mined.

4 Case Studies

We performed a case study on the transaction component of JBoss AS to show the applicability of our method in providing insight into the protocol that the code obeys – hence aiding program comprehension. Another case study on a buggy CVS (Concurrent Versions System) application adapted from the one previously studied in [15, 14] shows the utility of mined rules in identifying bugs via model checking.

JBoss AS Transaction Component. We instrumented the transaction component of JBoss-AS using JBoss-AOP and generated traces by running the test suite that comes with JBoss-AS distribution. In particular, we ran the transaction manager regression test of JBoss-AS. Twenty-eight traces of a total size of 2603 events, with 57 unique events, were generated. Running the algorithm with the minimum support and confidence thresholds set at twenty-five traces and ninety-percent respectively, 163 non-redundant rules were mined. The algorithm completed in 35.1 seconds.

A sample of the mined rules is shown in Figure 1. The 19-event rule in Figure 1 describes that the series of events ⟨connection to a server instance events, transaction manager and implementation set up event⟩ (event 1-10) at the start of a transaction is always followed by the series of events ⟨transaction completion events and resource release events⟩ (event 11-19) at the end of the transaction. The above rule describing the temporal relationship and constraint between the 19 events is hard to identify manually. The rule sheds light into the *implementation details* of JBoss AS which are implemented at various locations in (*i.e.*, crosscuts) the JBoss AS large code base.

CVS on Jakarta Commons Net. A case study was performed on a buggy CVS application built on top of the FTP library of Jakarta Commons Net to show the usefulness of

Premise	Consequent
TxManLocator.getInstance()	TransactionImpl.instanceDone()
TxManLocator.locate()	TxManager.getInstance()
TxManLocator.tryJNDI()	TxManager.releaseTransImpl()
TxManLocator.usePrivateAPI()	TransactionImpl.getLocalId()
TxManager.getInstance()	XidImpl.getLocalId()
TxManager.begin()	LocalId.hashCode()
XidFactory.newXid()	LocalId.equals()
XidFactory.getNextId()	TransactionImpl.unlock()
XidImpl.getTrulyGlobalId()	XidImpl.hashCode()
LocalId.associateCurrentThread()	
TransactionImpl.lock()	

Figure 1. A Rule from JBoss-Transaction

mined rules in verification and bug detection. The CVS interaction protocol with the underlying FTP library can be represented as a 33-state automaton partially drawn in Figure 2 (see [15, 14] for a more detailed diagram).

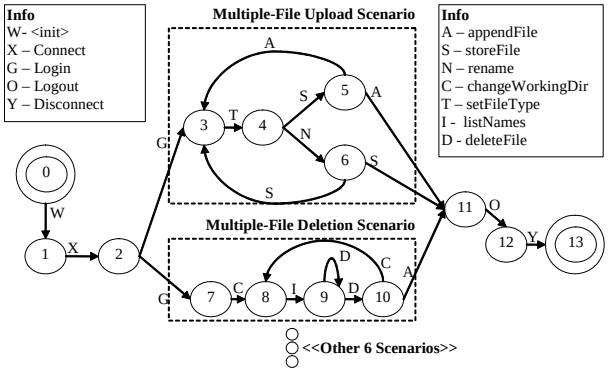


Figure 2. CVS Protocol

We focus on the two scenarios of multiple-file upload and deletion scenario. The scenarios start with connecting and logging into the FTP server and end by logging off and disconnecting from the server. Whenever a new file is added or a file is deleted a record is made to a system log file. Multiple versions of the same file are maintained by adding timestamp to old versions of the file.

The CVS application is buggy, there are 4 bugs that causes *inconsistent system log file*. The system log file describes the state of the CVS repository and should be kept consistent with the stored files. The 4 bugs are illustrated by the error transitions (in dashed lines) shown in Figure 3. Due to the bugs, a file can be added or deleted without a proper log entry being made. Also, an old version of a file can be renamed by appending a time-stamp without the new version being stored to the CVS.

The bugs occur because scenarios are not executed atomically. Each invocation of a method of `FTPClient` of the FTP library may generate exceptions, especially `ConnectionClosed` and `IO` exception. Hence the code accessing `FTPClient` methods need to be enclosed in a `try..catch..finally` block. Every time such an exception happens the program simply logout and disconnect from the FTP server.

To generate traces, we follow the process discussed in [15]. First, we instrument the CVS application byte

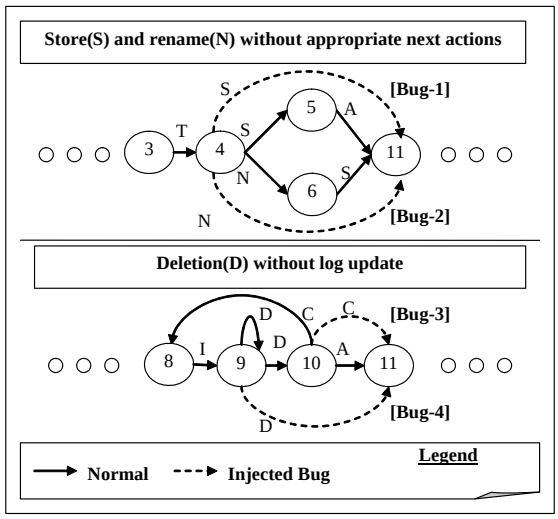


Figure 3. Injected Bug

code using an adapted version of Java Runtime Analysis Toolkit [12]. Next, we ran the instrumented CVS application over a set of test cases to generate traces. Via a trace post-processing step, we then filter events in the traces not corresponding to the interactions between the CVS application and the Jakarta Commons Net FTP library. Thirty-six traces of a total size of 416 events were generated.

We ran our mining algorithm on the generated traces. It ran in less in 1.1 second and mined 5 rules with minimum support and confidence thresholds set at fifteen traces and ninety percent respectively. Among the mined rules, the following two rules are the bug-revealing program properties:

1 Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, file type is set (T) and an old file is renamed(N), *then eventually* a new file is stored(S), followed by a logout (O) and a disconnection from server (Y). This is *denoted as*: $\langle W, X, G, T, N\rangle \rightarrow \langle S, O, Y\rangle$

2 $\langle W, X, G, C, I, D\rangle \rightarrow \langle A, O, Y\rangle$

We used the model checker described in [10]. We converted an abstract model of the CVS application to the format accepted by the model checker and checked against the above two properties. The model checker reported violations of the above properties. These violations correspond to 3 out of the 4 bugs (Bug-2,3,4) in the model.

5 Related Work

In [22], Yang et al. present an interesting work on mining two-event temporal logic rules (i.e., of the form $G(a \rightarrow XF(b))$, where G, X, and F are LTL operators [11]), which are statistically significant with respect to a user-defined 'satisfaction rate'. The algorithm presented, however, does not scale to mine multi-event rules of arbitrary length. To handle longer rules, Yang et al. suggest a partial solution based on concatenation of mined two-event

rules. Yet, the method proposed might miss some multi-event rules or introduce superfluous rules that are not statistically significant – it is neither sound nor complete. In contrast, we mine LTL rules of arbitrary size; scalability is accomplished by utilizing search space pruning strategies adapted from the data mining domain. The method is sound and complete as all mined rules are statistically significant and all statistically significant rules are mined.

There are many other work on mining frequent patterns [1, 21, 16], automaton [2, 20, 6, 15], Live Sequence Charts [18], etc. Technique wise, this work is similar to the family of pattern mining algorithms and has similar worst case complexity. By employing a pruning strategy, typical runtimes of pattern mining algorithms can be much less than the worst case complexity. However, different from the above existing studies, some of which are our own, in this work we mine LTL rules which have a different semantic and require different mining strategy than previous approaches.

6 Discussion and Future Work

Note that the time taken for mining is *much improved* with search space pruning strategy. Without employing a search space pruning strategy, the mining process will require at least E^L check operations, where E is the number of unique events and L is the maximum length of the trace. For traces from JBoss AS considered in Section 4, the mining process (without pruning) will require more than 50^{100} operations. Considering 1 picosecond per operation, it will only complete *in about* $2.501\text{x}10^{148}$ *centuries* whereas we simply need 35.1 seconds. This highlights the power and importance of search space pruning strategies in improving the scalability of the mining process.

In the second case study, Bug-1 cannot be revealed because the bug-revealing property is outside the bound of the LTL expressions minable by our algorithm. The bug-revealing property is: Whenever the application is initialized (W), the connection (X) and login (G) to the server are made, a file type is set (T), *and there is no rename (N) until a new file is stored (S)*, then eventually a log entry is made (A), followed by a logout (O) and a disconnection from server (Y). To mine the property, we need to mine rules containing the LTL operators not($\neg$) and until(U) – this is a future work.

Another open issue is in improving the scalability of our method further. One direction we are investigating is to reduce the size of input traces while retaining the quality of the specification mined. One can do so by throwing away non-important or less important events. In [23], Zaidman *et al.* identify important key classes using a web-mining algorithm. Similar approach to that in [23] might be employed to identify and remove less important events from the traces. In [13], Kuhn and Greevy partition a trace into different phases. Rather than mining specifications for the entire trace, one can separately mine each phase in the trace. This can be more efficient than mining the entire trace. If a test-suite is available, one can also perform a divide-and-conquer strategy by generating traces for each distinct part of the test suite (*i.e.*, group those testing the same component together) and analyzing them separately. Another direction we are investigating is to incorporate latest research in data mining to improve the algorithm further where approximation in the mining algorithm result can improve mining speed (*e.g.*, [24]).

7 Conclusion

In this paper, a novel method to mine a *non-redundant* set of *statistically significant* rules *of arbitrary lengths* of the form: "Whenever a series of events ES_{Pre} occurs, eventually another series of events ES_{Post} also occurs" is proposed. The problems of potentially exponential runtime cost and huge number of reported rules have been effectively mitigated by employing a search space pruning strategy and by eliminating redundant rules. Case studies have been conducted to demonstrate the usability of the proposed technique for program comprehension and verification.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Int. Conf. on Very Large DataBases*, 1994.
[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *SIGPLAN-SIGACT POPL*, 2002.
[3] Apache Software Foundations. Jakarta Commons/Net. *http://jakarta.apache.org/commons/net/*.
[4] G. Canfora and A. Cimitile. Software maintenance. In *Handbook of Software Eng. and Knowledge Eng.*, 2002.
[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
[6] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. on Software Eng. and Methodology*, 1998.
[7] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, 1999.
[8] J. Han and M. Kamber. *Data Mining: Concepts and Techniques, 2nd ed*. Morgan Kaufmann, 2006.
[9] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, (6):512–535, 1994.
[10] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In *TACAS*, 2006.
[11] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge, 2004.
[12] Java Runtime Analysis Toolkit. *jrat.sourceforge.net/*.
[13] A. Kuhn and O. Greevy. Exploiting analogy between traces and signal processing. In *ICSM*, 2006.
[14] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006.
[15] D. Lo and S.-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *SIGSOFT FSE*, 2006.
[16] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proc. of SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2007.
[17] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of strong temporal rules from a sequence database. *Technical report at: www.comp.nus.edu.sg/∼dlo/pcoda07-techrep.pdf*, 2007.
[18] D. Lo, S. Maoz, and S.-C. Khoo. Mining modal scenario-based specifications from execution traces of reactive systems. In *ASE (to appear)*, 2007.
[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.
[20] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, 2001.
[21] X. Yan, J. Han, and R. Afhar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. of SIAM Int. Conf. on Data Mining*, 2003.
[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M.Das. Perracotta: Mining temporal API rules from imperfect traces. In *ICSE*, 2006.
[23] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, 2005.
[24] F. Zhu, X. Yan, J. Han, P. Yu, and H. Cheng. Mining collosal frequent patterns by core pattern fusion. In *Int. Conf. on Data Eng.*, 2007.

Supporting Feature Analysis with Runtime Annotations – Position Paper –

Marcus Denker, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Abstract

The dynamic analysis approach to feature identification describes a technique for capturing feature behavior and mapping it to source code. Major drawbacks of this approach are (1) large amounts of data and (2) lack of support for sub-method elements. In this paper we propose to leverage sub-method reflection to identify and model features. We perform an on-the-fly analysis resulting in annotating the operations participating in a feature's behavior with meta-data. The primary advantage of our annotation approach is that we obtain a fine-grained level of granularity while at the same time eliminating the need to retain and analyze large traces for feature analysis.

Keywords: behavioral reflection, annotations, dynamic analysis, feature analysis, reverse engineering, program comprehension, software maintenance

1 Introduction

Traditionally, reverse engineering techniques focused on analyzing source code of a system [2]. In recent years, researchers have recognized the significance of centering reverse engineering activities around the behavior of a system, in particular, around features [8, 17, 21]. Reasoning about object-oriented systems in terms of features is difficult, as they are not explicitly represented in the source code. The first step therefore is to define what is meant by a feature, establish a feature representation and to locate the relevant parts of the source code that participate in its behavior.

Most existing feature analysis techniques capture traces of method events but they do not capture behavioral data of sub-method elements such as variable assignments [21, 17]. Furthermore, modeling features themselves poses some problems: features are typically modeled as traces of runtime activity resulting in the manipulation and interpretation of large amounts of trace data.

In this paper we address the following issues relevant to dynamic feature analysis:

- Dynamic feature analysis implies a need to manipulate large amounts of trace data.
- Current feature analysis techniques do not consider analysis to the granularity of sub-method elements (*i.e.* variable assignments).

Our goal is to show how *feature annotation* eliminates the need to manipulate large traces and thus makes it possible to collect fine-grained detail about the parts of the code that are involved in the runtime of a feature.

Paper structure. In the next section we briefly describe the current dynamic analysis approach to feature analysis and highlight problems such as the manipulation of large amounts of runtime data and the extraction of fine-grained behavioral information. In Section 3 we briefly introduce *sub-method reflection*, as it serves as a basis for our approach. Subsequently, in Section 4 we introduce our feature annotation approach. We discuss different aspects of our approach in Section 5. Section 6 outlines related work in the fields of dynamic analysis and feature identification. Finally we conclude in Section 7.

2 Dynamic Feature Analysis in a Nutshell

The goal of feature analysis is to reason about a system in terms of its features. A fundamental step of any feature analysis approach is to first apply a *feature identification* technique to locate features in source code. As a basis for feature analysis we use a model which expresses features as first class entities and their relationships to the source entities that implement their behavior [10]. Once the representation of a feature is established, we can reason about a system in terms of its features. Furthermore, we can enrich the static source code perspective with knowledge of the roles of classes and methods in the set of modeled features.

The generally adopted definition of a feature is a unit of observable behavior of a system triggered by a user [1, 8, 16, 24, 25]. Techniques for feature identification through dynamic analysis typically instrument a system, capture traces of feature behavior and establish links to

source code. However, capturing dynamic data to represent features raises many issues that need to be taken into consideration.

Large Amounts of Data. The volume of trace data generated represents a threat to the scalability of any feature analysis approach. As the granularity required for an experiment increases, so too does the volume of information generated.

Dynamic analysis approaches adopt different strategies to deal with large amounts of data. Some of the most popular strategies adopted by researchers to tackle and analyze dynamic data are: (1) summarization through metrics [7], (2) filtering and clustering techniques [14, 26], (3) visualization [3, 12] (4) selective instrumentation and (5) query-based approaches [20]. Many techniques apply a combination of these strategies.

Instead of trying to compress the trace data, we need to question the idea of modeling features as execution traces.

Fine-Grained Analysis. Traditionally, dynamic analysis techniques for feature analysis focused on execution traces consisting of a sequence of method executions [8, 25]. Some dynamic analysis approaches trace additional properties of behavior such as the message receiver and arguments or instance creation events [4, 13]. However very little work in feature analysis has focused on a means to model which sub-method entities are part of a feature.

Fine-grained analysis down to the operation level should be possible.

Before we present our solution to these issues, we briefly introduce the extended reflection mechanism that enables *feature annotation* in Section 4.

3 Sub-Method Reflection

Reflection in programming languages is a paradigm that supports computations about computations, so-called *meta-computations*. Metacomputations and base computations are arranged in two different levels: the *metalevel* and the *base level* [22]. Because these levels are causally connected any modification to the metalevel representation affects any further computations on the base level [19]. Structurally reflective systems contain a first-class, causally connected model of their own structure: classes and methods are objects and changing these objects directly changes the system [9].

Structural reflection stops at the granularity of the method: a method is an object, but the operations the method contains are not modeled as objects. Examples of these operations would be message sends, variable reads or assignments. *Sub-Method Reflection* [5] extends the traditional model of structural reflection to encompass sub-method elements in addition to classes and methods. This is done by associating an extended AST (*Abstract Syntax Tree*) representation with the method.

Before execution, the AST is compiled on demand to a low-level representation that is executable, for example to byte-codes executable by a virtual machine.

Another mechanism provided by sub-method reflection is the annotation. Sub-method reflection provides a framework for annotating any program element with meta-data. An open compiler infrastructure supports the definition of compiler plugins that react to annotations by transforming the generated code.

We have extended Squeak Smalltalk to support sub-method reflection. More in-depth information about this system and its implementation can be found in the paper on sub-method reflection [5].

3.1 Partial Behavioral Reflection

Structural reflection is concerned with modeling the static structure of the systems. *Behavioral reflection* provides a model for execution and a way to intercept and change the execution of a program.

Whereas structural reflection is about classes, methods and the instructions inside the methods, behavioral reflection is concerned with execution events, *i.e.* method execution, message sends, or variable assignments.

One model for behavioral reflection is *Partial Behavioral Reflection* as pioneered by Reflex [23]. We have argued in the past [6] that this model is particularly well suited for dynamic analysis. It supports a very fine-grained, temporal and spatial selection of what exactly to reflect on. Thus it provides control of where and in which context dynamic analysis should be deployed in the system.

The core concept of the Reflex model of partial behavioral reflection is the *link* (see Figure 1). A link invokes messages on a metaobject at occurrences of selected operations. Link attributes enable further control of the exact message sent to the meta-object. One example for a link attribute is the activation condition which controls if the link is really invoked.

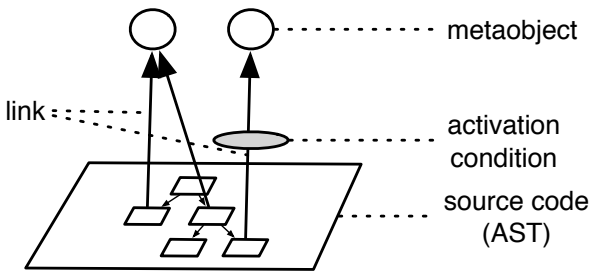


Figure 1. The reflex model

The original implementation of partial behavioral reflection for Java was based on bytecode transformation.

Sub-Method Reflection provides a natural implementation substrate for realizing partial behavioral reflection. Links are annotations on the operations provided by sub-method-reflection. A plugin enables the compiler to take the links into account when generating the bytecode for a method.

4 Feature Annotation

Feature identification (*i.e.* locating which parts of the code implement a feature) by dynamic analysis is done at runtime: the feature is executed and the execution path is recorded. For example, when exercising *Login* feature of an application, we record all methods that are called as a result of triggering this feature. This trace of called methods then encompasses exactly all those methods that are part of the login feature.

4.1 Behavioral Reflection and Features

Trace-based feature analysis can be easily implemented using partial behavioral reflection. In a standard trace-based system, the tracer is the object responsible for recording the feature trace. This tracer is the meta-object (see Figure 2). We define a link that calls this meta-object with the desired information passed as a parameter (e.g. the name and class of the executed method). The link then is installed on the part of the system that we want to analyze. When we then exercise the feature, the trace meta-object will record a trace.

The resulting system is very similar to existing trace-based systems, with one exception: tracing now can easily cover sub-method elements, if required.

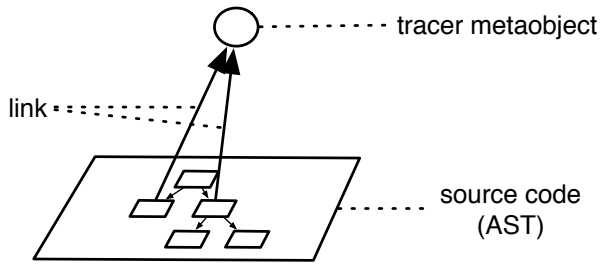


Figure 2. A tracer realized with partial behavioral reflection

4.2 Feature Annotation with Behavioral Reflection

In contrast to traditional dynamic feature analysis approaches, our sub-method reflection based approach does not need to retain a trace. The goal of feature identification is to map features to the source code. With the annotatable representation provided by sub-method reflection, we can annotate every statement that participates in the behavior of a feature. Instead of recording traces, we tag all the AST nodes that are executed as part of a feature with a *feature annotation* at runtime.

The annotation at runtime is realized using partial behavioral reflection. We do not need a dedicated tracer application anymore, instead the meta-object that models an instruction (the AST node) tags itself if it is part of a feature. For this, we define a link that calls a method on the node on which it is installed. This method tags the appropriate node with a feature annotation (see Figure 3).

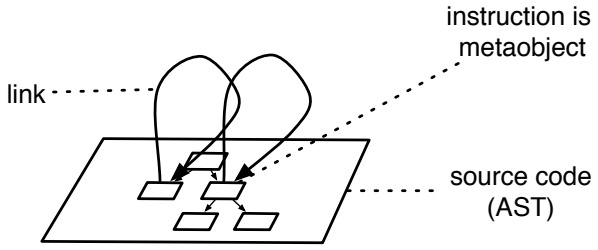


Figure 3. Annotating nodes using partial behavioral reflection

We install the link on all the AST nodes of the system that we plan to analyze. Exercising the feature subsequently annotates all methods or instructions that take part in a feature execution. In this way we do not need to retain traces, resulting in less data to be managed. We have not yet conducted extended case studies using our technique. However preliminary studies with feature traces captured using the traditional approach to tracing show that for the number of methods that are part of a trace, we get 10 times more method execution events. Thus there is a factor of 10 between the number of method execution events and the number of distinct methods in a trace.

The idea of *feature annotation* has many implications, both for how to model and analyze features. This position paper presents the basic idea of feature annoations, the next section discusses some of the possibilities and drawbacks of our approach.

5 Discussion

Our feature annotation approach can easily support many of the existing feature analysis approaches. For example, we could exercise a feature multiple times with different parameters to obtain multiple paths of execution. This can be important, as the traces obtained can very considerably depending on the input data.

For trace-based approaches this results in a many-to-one mapping between features and traces. Using our approach, if the execution path differs over multiple runs, newly executed instructions will be tagged in addition to those already tagged. Thus we can use our approach to iteratively build up the representation of a feature covering multiple paths of execution.

Instead of multiple runs resulting in one feature annotation, the feature annotations can be parametrized with the amount of executions that are the result of exercising the feature. We can, for example, record a metric if a statement is always part of a feature or only in certain contexts similar to the reconnaissance metric of Wilde and Scully [24] or our other feature analysis work [11]. Other information that can be captured is *e.g.*, instance information or feature dependencies as described in the approaches of Salah *et al.* or Lienhard *et al.* [21, 18]. Naturally, the more information gathered at runtime, the more memory would be required. In the worst case, recording everything would result in recording the same amount of information as a complete trace of fine-grained behavioral information.

A downside of the filtering at runtime is that dynamic information is lost. It is crucial to define which information is necessarry for a given feature analysis. A change in a selection strategy implies a need to exercise a feature again. In contrast approaches based on complete traces can perform a variety of postmortem analyses of feature traces, each requiring different level of detail.

6 Related work

We review dynamic analysis approaches to system comprehension and feature identification approaches and discuss these in the context of our work.

Dynamic Analysis for Program Comprehension. Many approaches to dynamic analysis focus on the problem of tackling the large volume of data. Many of these works propose compression and summarization approaches to support the extraction of high level views [26, 11, 15].

Feature Identification through dynamic analysis. Dynamic analysis approaches to feature identification have typically involved executing the features of a system and analyzing the resulting execution trace [24, 25, 8, 1]. Typically, the research effort of these works focuses on the underlying mechanisms used to locate features (*e.g.*, static analysis, dynamic analysis, formal concept analysis, semantic analysis or approaches that combine two or more of these techniques).

Wilde and Scully pioneered the use of dynamic analysis to locate features [24]. They named their technique *Software Reconnaissance*. Their goal was to support programmers when they modify or extend functionality of legacy systems.

Eisenbarth *et al.* described a semi-automatic feature identification technique which used a combination of dynamic analysis, static analysis of dependency graphs, and formal concept analysis to identify which parts of source code contribute to feature behavior [8]. For the dynamic analysis part of their approach, they extended the Software Reconnaissance approach to consider a set of features rather than one feature. They applied formal concept analysis to derive a correspondence between features and code. They used the information gained by formal concept analysis to guide a static analysis technique to identify feature-specific *computational units* (*i.e.*, units of source code).

Wong *et al.* base their analysis on the *Software Reconnaissance* approach and complement the relevancy metric by defining three new metrics to quantify the relationship between a source artefact and a feature [25]. Their focus is on measuring the closeness between a feature and a program component.

All of these feature identification approaches collect traces of method events and use this data to locate the parts of source code that implement a feature. Thus, the feature identification analysis is based on manipulating and analyzing large traces. Furthermore, many of the dynamic analysis approaches do not capture fine-grained details such sub-method execution events. The main limiting factor is the amount of trace data that would result. Our approach eliminates the need to retain execution traces. Thus there is no limitation to annotating all events (methods and sub-methods) involved in a feature's behavior.

Furthermore, a key focus of feature identification techniques is to define measurements to quantify the relevancy of a source entity to a feature and to use the results for further static exploration of the code. Thus these approaches do not explicitly express the relationship between behavioral data and source code entities. Thus to extract high level views of dynamic data, we need to process the large traces. Other works [1, 10] identify the need to extract a model of behavioral data in the context of structural data of the source code. Subsequently feature analysis is performed on the model rather than on the source code itself.

7 Conclusions and Future Work

In this paper we have presented *feature annotation*, a technique that solves some issues found in traditional trace-based dynamic feature analysis systems. Feature Annotation support the analysis on a sub-method level and does not require to store complete trace data.

We have implemented a prototype of feature annotation, future work includes using it on large case-studies. We plan to analyze both performance and memory characteristics and compare our approach to a trace collecting feature analysis system.

Another interesting direction of future work is to experiment with advanced scoping mechanisms, *e.g.* we want to experiment with the idea of scoping dynamic analysis towards a feature instead of static entities like packages and classes.

Acknowledgments.. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366, Los Alamitos CA, Sept. 2005. IEEE Computer Society Press.

[2] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[3] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE Computer Society, 2007.

[4] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[5] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Sub-method reflection. *Journal of Object Technology*, 6(9):231–251, Oct. 2007.

[6] 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.

[7] 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.

[8] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[9] J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, Oct. 1989.

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

[11] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.

[12] O. Greevy, S. Ducasse, and T. Gîrba. Analyzing software evolution through feature views. *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, 18(6):425–456, 2006.

[13] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of CSMR 2003*. IEEE Press, 2003.

[14] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.

[15] 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 International Conference on Program Comprehension (ICPC 2006)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[16] R. Koschke and J. Quante. On dynamic feature location. *International Conference on Automated Software Engineering, 2005*, pages 86–95, 2005.

[17] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, Oct. 2006.

[18] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC 2007)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[19] P. Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, Jan. 1987.

[20] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*, page 34, Los Alamitos CA, Oct. 2002. IEEE Computer Society.

[21] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.

[22] B. C. Smith. Reflection and semantics in a procedural language. Technical Report TR-272, MIT, Cambridge, MA, 1982.

[23] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[24] N. Wilde and M. Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[25] E. Wong, S. Gokhale, and J. Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.

[26] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 329–338, Los Alamitos CA, Mar. 2004. IEEE Computer Society Press.

Identification of Behavioral and Creational Design Patterns through Dynamic Analysis

Janice Ka-Yee Ng and Yann-Gaël Guéhéneuc
Ptidej Team – GEODES
Département d'informatique et de recherche opérationnelle
Université de Montréal – CP 6128 succ. Centre Ville
Montréal, Québec, H3C 3J7 – Canada
ngjanice@iro.umontreal.ca and guehene@iro.umontreal.ca

Abstract

Design patterns are considered to be a simple and elegant way to solve problems in object-oriented software systems, because their application leads to a well-structured object-oriented design, and hence, are considered to ease software comprehension and maintenance. However, after application, design patterns are lost in the source code and are thus of little help during program comprehension and maintenance. In the past few years, the structure and organization among classes were the predominant means of identifying design patterns in object-oriented software systems. In this paper, we show how to describe behavioral and creational design patterns as collaborations among objects and how these representations allow the identification of behavioral and creational design patterns using dynamic analysis and constraint programming.

1 Introduction

Software maintenance is considered a crucial phase of the software development process, as it consumes as much as 90% of the total resources related to the software life cycle [5]. Part of the activities of software maintenance is program comprehension, where developers try to identify the structure and organization of code artifacts, with the help of re-engineering tools, to then perform the maintenance tasks, such as debugging or adding new features.

For more than 10 years now, design patterns [6] have been increasingly used to design and obtain well-structured software systems. However, due to the complexity of large object-oriented software systems nowadays, it is impossible to recover manually the design patterns applied during the design and the implementation of a system, which, in turn, impedes its comprehension [1].

In the past, several approaches have been proposed to detect design patterns in source code using static analysis, for example [14, 15, 16, 19, 22, 24, 25]. The fundamental idea of these approaches consists in analyzing the class structure of a system to identify classes whose structure resembles the most the structure of a design pattern. The dynamic aspect of the system has almost been completely ignored, but it should not be because, on the one hand, behavioral and creational design patterns can hardly be described by their structure, and on the other hand, the dynamic aspect provides data to complement those related to the architecture and design of software systems, as shown in [11, 12].

In this paper, we propose a 3-step approach (as illustrated in Figure 1 Steps 1, 2, and 3) to identify behavioral and creational design patterns in source code using dynamic analysis. First, we describe behavioral and creational design patterns in terms of UML sequence diagrams (really scenario diagrams as explained in Section 2). Second, using dynamic analysis, we reverse engineer a dynamic model—such as UML sequence diagrams—of any given object-oriented software system written in Java. Finally, we perform the `Visitor` pattern identification on one particular scenario of JHotDraw. To this end, we translate the problem of design patterns identification in terms of a constraint satisfaction problem (a.k.a. CSP).

This paper is structured as follow: In Section 2, we provide a metamodel to capture the interactions between objects at runtime. Then, a description of behavioral and creational design patterns in terms of the constructs of this metamodel is provided in Section 3.

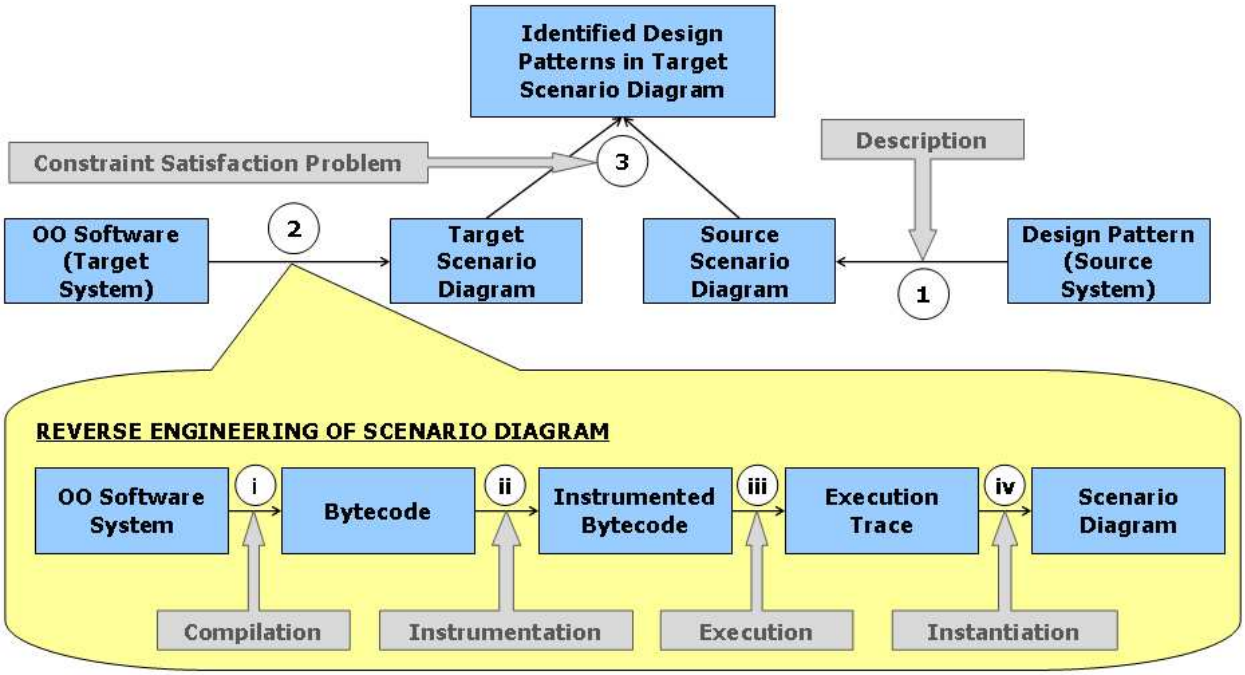


Figure 1. A 3-step approach for the identification of design patterns through dynamic analysis.

In Section 4, we describe our technique to reverse engineer scenario diagram of object-oriented software systems using dynamic analysis. Section 5 elaborates on the technique used to identify behavioral and creational design patterns. We then report the results of one case study in Section 6. Challenges and limitations of the approach are discussed in Section 7. Related work is provided in Section 8. Finally, Section 9 concludes and presents future work.

2 Scenario Diagram Metamodel

In the context of design patterns identification, the reverse engineered UML sequence diagrams are obtained from the execution of some particular use cases. Therefore, these diagrams are referred to as *scenario diagram* [3], as they are only partial UML sequence diagrams describing one specific scenario corresponding to a use case instead of all possible alternatives for the exercised use case.

Following [3] and [17], we implement a metamodel of scenario diagrams to express the data we need to describe the behavior of design patterns and software systems. Figure 2 shows our scenario diagram metamodel. A scenario diagram, class `ScenarioDiagram`, is composed of an ordered list of components, class `Component`, that can either be messages, class `Message`, or combined fragments, class `CombinedFragment`.

Messages can be of three different types: an operation call, class `Operation`, a destruction call, class `Destroy`, or a creation call, class `Create`. Messages have a `sourceClassifier` and a `destinationClassifier` to represent the concept of caller and callee. Caller and callee are of type `Classifier` that can be specialized into an `Instance` or a `Class`, the latter case is applicable if the message in relation to the caller or callee is a class method. If any, messages are composed of arguments, class `Argument`, of different types: either primitive types or object types. The return value of messages is class `ReturnValue`.

Class `CombinedFragment` is inspired by a previous notation [17] to group sets of messages to show conditional flows in sequence diagrams. Although [17] provides eleven interactions types of combined fragments, only the combined fragments loops and alternatives are necessary to behavioral and creational design patterns identification. In this context, combined fragments can be specialized into two types: either loops, class `Loop`, to illustrate repetitions of messages, or alternatives, class `Alt`, to designate mutually exclusive choices between sequence of messages. To model the case where a loop or an alternative is nested into another loop or alternative, we introduce composition links: composition `operand` between classes `Loop` and `Component`, and composition `operands` between classes `Alt` and

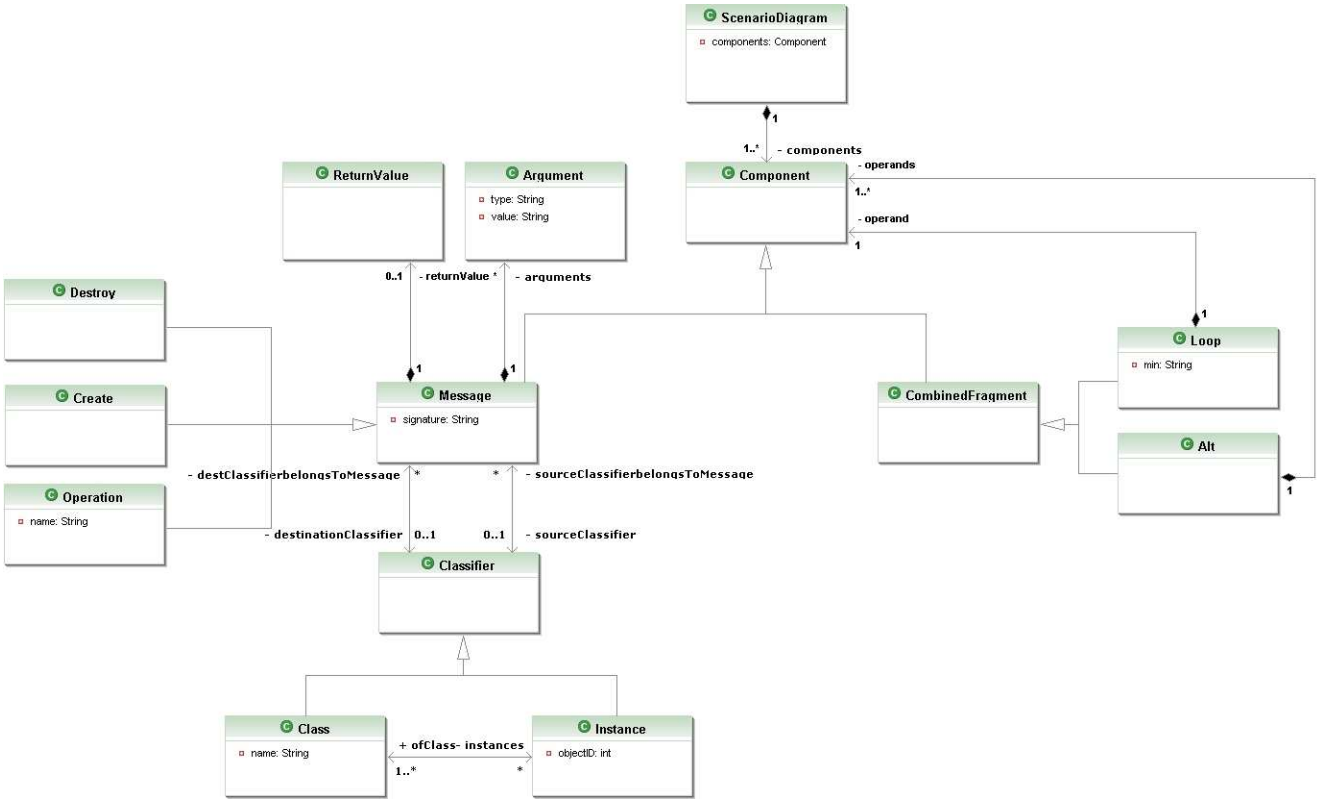


Figure 2. Scenario diagram metamodel.

`Component`. A loop has one and only one operand, while an alternative has one or more operands. For instance, the classic alternative 'if then else' has two operands: operand 'if', and operand 'else'.

3 Description of Design Patterns

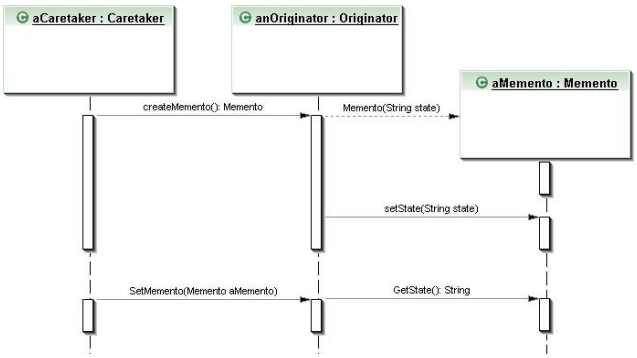


Figure 3. Memento pattern scenario diagram.

In [6], each design pattern is provided with their own description in terms of collaborations between participants. In particular, for some specific patterns, the authors have chosen to use diagrams similar to scenario diagrams to show sequences of messages between objects, i.e., the order in which messages between participants of design patterns are executed. For instance, Figure 3 is a scenario diagram that illustrates how the participants of the `Memento` pattern collaborate.

In our approach, as illustrated in Figure 1 Step 1, we describe behavioral and creational design patterns by transforming the graphical description of collaborations in [6] into an instance of the scenario diagram metamodel. For each design pattern for which a graphical description is available, we describe its participants and its sequence of messages in terms of objects of the scenario diagram metamodel. For instance, for each message involved in the sequence of messages in Figure 3, we instantiate an object `Operation` that is added to the ordered list `components` of an object `ScenarioDiagram` representing the `Memento` pattern. Given message `setMemento(Memento aMemento)` takes 'aMemento' as argument, we instantiate an object `Argument` whose attribute `type` is `Instance`. This object `Argument` is added to the ordered list `arguments` of message `setMemento(Memento aMemento)`. The participants collaborating in the pattern, for instance

`aCaretaker`, `anOriginator`, and `aMemento`, are instantiated as objects `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of the corresponding message. For example, the `sourceClassifier` of `createMemento()` and `setMemento(Memento aMemento)` is `aCaretaker`, while `anOriginator` is their `destinationClassifier`.

4 Reverse Engineering of Scenario Diagram

In the literature, many approaches have been proposed to reverse engineer dynamic models of object-oriented software systems. Based on [3], our approach for the scenario diagram reverse engineering consists in 4 steps (as illustrated in Figure 1 Steps i, ii, iii, and iv). First, we compile the source files of an object-oriented software system to obtain their corresponding class files. Second, we instrument the class files using bytecode instrumentation. Third, we execute the instrumented system following some scenario to produce an execution trace. Finally, we instantiate the scenario diagram that corresponds to the execution trace.

In this section, we describe briefly the mechanism used to produce an execution trace containing dynamic data of an object-oriented software systems in Java (Step ii), and to instantiate a scenario diagram from the execution trace (Step iv).

Instrumentation. In terms of design patterns identification, the type and amount of dynamic data to retrieve are relative to the description of design patterns. In this context, we focus primarily on the control flow data, that is, the sequence of messages actually executed during runtime. We have chosen to instrument Java bytecode with BCEL—the Byte Code Engineering Library [2]. BCEL is a Java library that gives users the possibility to create, analyze, and manipulate easily Java class files.

We need to trace the execution of methods and constructors to instantiate class `Message` in a scenario diagram. To this end, we introduce bytecode instructions to produce dynamic data before and after the execution of the methods and constructors. We indicate in the execution trace when methods and constructors start and end executing in relation to other events. Figure 4 shows an example of execution trace of a toy system implemented in Java.

Instantiation of Scenario Diagram. To obtain the scenario diagram corresponding to an execution trace, the latter is processed. For each execution trace statement such as `operation start`, `constructor start`, or `destructor start`, a message of type `Operation`, `Create`, or `Destroy` is respectively instantiated, while an object `CombinedFragment` of type `Loop` or `Alt` is instantiated for each execution trace statement `loop start` or `alt start`. In both cases, the component corresponding to the line currently analyzed in the execution trace is referred to as the *current component*. If the current component is of type `CombinedFragment`, we add the subsequent objects `Message` or `CombinedFragment` to its ordered list `operands`, until the corresponding end statement is met. Otherwise, they are added to the ordered list `components` of object `ScenarioDiagram`. Each time an object `Message` is instantiated, its corresponding `sourceClassifier` and `destinationClassifier` of type `Classifier` are also determined and instantiated. The set `arguments` of a message is determined by processing the data positioned between the brackets of the corresponding execution statement. Figure 5 is a textual description of the scenario diagram corresponding to the execution trace in Figure 4.

5 Identification of Design Patterns

Using the reverse engineering technique described in the previous section, we instantiate two scenario diagrams. One instance models the sequence of messages of a design pattern, i.e., a *source system*, and the other instance models the sequences of messages of a given source code, i.e., a *target system*. The approach we propose to identify behavioral and creational design patterns in object-oriented software systems consists in identifying the scenario diagrams of a source system in the scenario diagram of a target system.

As illustrated in Figure 1 Step 3, we translate the problem of design patterns identification in terms of a constraint satisfaction problem (CSP, as in previous work [8]). We define the problem of detecting a design pattern in terms of its variables, the constraints among them, and their domains. This CSP represents the problem that the explanation-based constraint solver JCHOCO [13] solves to identify in the target system, sequence of messages that is identical or similar to the one defined by the source system.

Variables. The set of variables `Classifier` and `Message` corresponds respectively to the entities `Classifier` and `Message` modelling the scenario diagram of a design pattern (the source system).

Constraints. The set of constraints among the variables corresponds to the relationships among the en-

```
operation start public static void main (String[] args) callee ModelMementoTest -1
constructor start public void <init>() callee Caretaker 14613018
constructor start public void <init>() callee Originator 12386568
constructor end  public void <init>() callee Originator 12386568
constructor end public void <init>() callee Caretaker 14613018
operation start public void callCreateMemento() callee Caretaker 14613018
operation start public Memento createMemento() callee Originator 12386568
constructor start public void <init>() callee Memento 17237886
constructor end public void <init>() callee Memento 17237886
operation start public void setState(String state) callee Memento 17237886
operation end public void setState(String state) callee Memento 17237886
operation end public Memento createMemento() callee Originator 12386568
operation end public void callCreateMemento() callee Caretaker 14613018
operation start public void undoOperation() callee Caretaker 14613018
operation start public void setMemento(Memento m) callee Originator 12386568
operation start public String getState() callee Memento 17237886
operation end public String getState() callee Memento 17237886
operation end public void setMemento(Memento m) callee Originator 12386568
operation end public void undoOperation() callee Caretaker 14613018
operation end void public static void main (String[] args)  callee ModelMementoTest -1
```

Figure 4. Example of execution trace of a toy system implementing the Memento Pattern.

```
<OPERATION> public static void main (String[] args) <CALLEE> ModelMementoTest <CALLER> inexistant
<CREATE> public void <init>() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
<CREATE> public void <init>() <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
<OPERATION> public void callCreateMemento() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
<OPERATION> public Memento createMemento() <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
<CREATE> public void <init>() <CALLEE> Memento 17237886 <CALLER> Originator 12386568
<OPERATION> public void setState(String state) <CALLEE> Memento 17237886 <CALLER> Originator12386568
<OPERATION> public void undoOperation() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
<OPERATION> public void setMemento(Memento m) <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
<OPERATION> public String getState() <CALLEE> Memento 17237886 <CALLER> Originator 12386568
```

Figure 5. Textual representation of the scenario diagram of Figure 4.

tities of the scenario diagram defined by a design pattern. We use binary constraints, of the form `constraint(variable1, variable2)`, to express the relationships between `variable1` and `variable2`.

Domain. The domain of each variable (`Classifier` or `Message`) corresponds to a set of integers, each corresponding respectively to an entity `Classifier` or `Message` in the scenario diagram of a target system.

For a given set of constraints, the constraint solver JChoco solves the CSP by removing from the domains values that do not satisfy the relationships between `variable1` and `variable2`. If the constraint solver JChoco provides no solution for a CSP, then the corresponding design pattern is considered as not implemented in the target system.

The constraint `caller (classifier1, message2)` (respectively `callee`) defines the relationship '*classifier1 is the sourceClassifer of message2*' (respectively `destinationClassifier`) between `classifier1` and `message2`. The constraint `creator(classifier1, message2)` (respectively `created`) is very similar to constraint `caller(classifier1, message2)`, except that `message2` is an instance of `Create` instead of `Operation` (c.f. Figure 2). Finally, the constraint `follows(message1, message2)` defines the relationship '*message2 is executed after message1*'.

For example, the `Memento` pattern, as shown in Figure 3, is the source system and is modelled by associating a variable with each entity in the scenario diagram (`var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState`, `var_aCaretaker`, `var_anOriginator`, and `var_aMemento`), and by constraining the values of these variables according to the relationships among the entities:

```
follows(var_createMemento, var_newMemento)
follows(var_newMemento,var_setState)
follows(var_setState, var_setMemento)
follows(var_setMemento, var_getState)
caller(var_aCaretaker, var_createMemento)
callee(var_anOriginator, var_createMemento)
creator(var_anOriginator, var_newMemento)
created(var_aMemento, var_newMemento)
caller(var_anOriginator, var_setState)
callee(var_aMemento, var_setState)
caller(var_aCaretaker, var_setMemento)
callee(var_anOriginator, var_setMemento)
```

```
caller(var_anOriginator, var_getState)
callee(var_aMemento, var_getState)
```

The resolution of the CSP modelling the `Memento` pattern returns results of the form:

```
<Sol.#>.var_createMemento = <an entity>
<Sol.#>.var_newMemento    = <an entity>
<Sol.#>.var_setState      = <an entity>
<Sol.#>.var_setMemento    = <an entity>
<Sol.#>.var_getState      = <an entity>
<Sol.#>.var_caretaker     = <an entity>
<Sol.#>.var_originator    = <an entity>
<Sol.#>.var_memento       = <an entity>
```

When applied to the toy system, our approach found one solution:

```
1.var_createMemento = createMemento()
1.var_newMemento    = new Memento()
1.var_setState      = setState(String state)
1.var_setMemento    = setMemento()
1.var_getState      = getState()
1.var_caretaker     = Caretaker [14613018]
1.var_originator    = Originator [12386568]
1.var_memento       = Memento [17237886]
```

6 Case Study

To evaluate our approach, we applied it on JHotDraw v6.0b1 (15 KLOCs), which is a drawing editor with a GUI based on an open source system written in Java. Although it is intentionally designed to have very clear implementations of well-known design patterns, its documentation can eventually help us determine the precision and recall properties of our approach. The scenario used to identify occurrences of the `Visitor` pattern in JHotDraw is to *Cut and Paste a figure in a document*:

```
Create a new document on which figures can be drawn;
Select the 'Draw Rectangle' tool from the menu;
Select the rectangle figure drawn at step 2;
Select the 'Cut' command from the menu;
Select the 'Paste' command from the menu.
```

For this scenario, our approach includes twice in the solution variables `var_accept`, `var_visitConcreteElement`, `var_operation`, `var_objectStructure`, `var_concreteElement`, and `var_concreteVisitor`, to illustrate both the actions '*Cut*' and '*Paste*' executed in the same scenario. By applying our approach to this scenario on a subset of JHotDraw (for performance issues as discussed in Section 7), we obtained two occurrences of the `Visitor` pattern.

Occurrence 1 is:

```
1.var_accept1 = visit(FigureVisitor visitor)
1.var_visitConcreteElement1 = visitFigure(Figure hostFigure)
1.var_operation1 = removeFromContainer(FigureChangeListener c)
1.var_objectStructure1 = FigureTransferCommand [7760420]
1.var_concreteElement1 = AbstractFigure [5489653]
1.var_concreteVisitor1 = DeleteFromDrawingVisitor [12741398]
1.var_accept2 = visit (FigureVisitor visitor)
1.var_visitConcreteElement2 = visitFigure (Figure hostFigure)
1.var_operation2 = setZValue (int z)
1.var_objectStructure2 = FigureTransferCommand [26980954]
1.var_concreteElement2 = AbstractFigure [31746664]
1.var_concreteVisitor2 = InsertIntoDrawingVisitor [2554341]
```

and Occurrence 2 is:

```
1.var_accept1 = visit (FigureVisitor visitor)
1.var_visitConcreteElement1 = visitFigure (Figure hostFigure)
1.var_operation1 = addToContainer (FigureChangeListener c)
1.var_objectStructure1 = FigureTransferCommand [26980954]
1.var_concreteElement1 = AbstractFigure [31746664]
1.var_concreteVisitor1 = InsertIntoDrawingVisitor [2554341]
1.var_accept2 = visit (FigureVisitor visitor)
1.var_visitConcreteElement2 = visitFigure (Figure hostFigure)
1.var_operation1 = removeFromContainer (FigureChangeListener c)
1.var_objectStructure2 = FigureTransferCommand [7760420]
1.var_concreteElement2 = AbstractFigure [5489653]
1.var_concreteVisitor2 = DeleteFromDrawingVisitor [12741398]
```

According to the documentation in JHotDraw, the value of the variables provided in Occurrence 2 correspond to the participants and messages involved in the `Visitor` pattern. In contrast, the value of variable `var_operation2` in Occurrence 1, `public void setZValue (int)`, is not involved in the sequence of messages corresponding to the action '*Paste*'. Therefore, Solution 1 is not an occurrence of the `Visitor` pattern.

7 Challenges and Limitations

In this section, we elaborate on several challenges we faced while reverse engineering scenario diagrams and identifying design patterns, as well as the main problems of the proposed approach.

Dynamic and Static Analysis. In obtaining scenario diagrams, we can choose to capture the behavior of a software system either by static analysis or dynamic analysis. Both strategies have their own drawbacks. On the one hand, even if static analysis can depict a complete picture of what could happen at runtime, it does not show what *actually* happens. Furthermore, using static analysis to retrieve dynamic data requires to analyze source code and determine the dynamic types of object references, which is not conceivable for large, complex systems [7]. On the other hand, reverse engineered scenario diagrams using dynamic analysis represent only part of the system's whole behavior. However, it reports precisely on the interactions between objects.

In the context of design patterns identification, precise data outweighs completeness. Therefore, we favor dynamic analysis over static analysis. To make up the incompleteness of reverse engineered scenario diagrams, we will consider as future work the merging of

several traces, each reporting on one observed behavior according to one scenario (or use case). Also, using test coverage tools can help defining the scenarios that need to be executed to possibly recover all the design patterns applied during the design and implementation of a system.

Target Language and Runtime Environment Specific Approach. In the proposed 3-step approach for the identification of design patterns through dynamic analysis, the process of scenario diagram reverse engineering is specific to the target language: we used bytecode instrumentation to trace a software system's method execution. This instrumentation technique has obvious drawbacks, among which it is specific to the target language, and highly coupled with a particular runtime environment. However, the fundamental principles according to which dynamic data is retrieved should not be affected. Regardless of the language (as long as it is object-oriented) or the runtime environment of the target system, a method execution is traced in such a way that instrumentation bytecode instructions are placed before and after the execution starts and ends.

In contrast, the process of design patterns identification using CSP (Figure 1 Step 3) is not specific to the target language, since its principal actors—variables, constraints, and domain—are described in terms of the constructs of the scenario diagram metamodel only.

Scalability and Performance. One of the key challenges while using dynamic analysis to monitor the behavior of a software system is the large amount of data traced. As the size of the target system grows, the execution trace grows in parallel, and as a result, execution time required to solve the CSP deteriorates.

Among the most commonly used abstraction mechanisms to cope with high volume of data, we used start and end markers to specify respectively the start and end of the action primary to a particular scenario. For instance, in the '*Cut and Paste a figure in a document*' scenario described in Section 6, the two principal actions involved are actions '*Cut*' and '*Paste*'. We thus placed two markers in the execution trace of the corresponding scenario to specify the start and end of action '*Cut*', just before and after the user chooses '*Cut*' in the menu of JHotDraw. In the same manner, two markers are specified respectively for the start and end of action '*Paste*'. In this manner, method executions that are positioned outside each pair of start and end markers can be omitted from the execution trace. Results after applying our identification approach both on the original and the summarized execution trace show identical solutions for the `Visitor` pattern.

The marker mechanism is our first attempt to reduce the volume of dynamic data, and still needs some more refinements to assure that no occurrences of design pattern are omitted because some method executions are eliminated from the original execution trace.

Design Pattern Description. As explained in Section 3, we describe design patterns in terms of collaborations given in [6]. However, as design patterns need not be collaborating precisely as described in the Gang of Four, the design patterns description step could be automated in such a way that users could easily describe the collaborations between participants to characterize their own patterns of interest.

8 Related Work

The identification of design patterns in object-oriented software systems has been the subject of many works. In particular, the identification of structural design patterns has been investigated since as early as 1998 [25]. However, we are not aware of work dedicated to the identification of general non-structural design patterns. Thus, we present work related to the identification of structural design patterns, the use of dynamic data during structural design patterns identification, and the recovery of interaction diagrams.

Structural Pattern Identification. Wuyts [25] published a precursor work on structural design patterns identification. His approach consisted in representing systems as Prolog facts and in describing design pattern as predicates on these facts. Facts were extracted using static analysis. This approach had performance issues, could not deal with variations, and had limited precision and recall. It was followed by many other works to improve on its limits. These works include the use of constraint programming [20], explanation-based constraint programming [10], and, more recently, similarity scoring [24].

Dynamic Data for Identification. To the best of our knowledge, no previous work focused on the identification of behavioral and creational design patterns. Heuzeroth et al. [11, 12] proposed an approach that uses both static and dynamic data to identify so-called interaction patterns and exemplified their approach on the `Observer` pattern using a dedicated detection algorithm. It is unclear how this approach can be generalized to pure-behavioural/creational design patterns.

Shawky et al. [23] proposed a similar approach to improve the precision and recall of a static identification approach.

Some previous works also used dynamic data in addition to structural data to improve precision and recall. In particular, most previous work on the identification of structural design patterns use data related to method calls, which can be considered as dynamic data, for example [1], or [9] used in [10].

Recovery of Interaction Diagrams. The recovery of interaction diagrams has been tackled by several authors. An important contribution to this line of research is the work of De Pauw et al. [18], which describes a model to visualize data about the execution of an object-oriented software system. Briand et al. [4] proposed a method to reverse engineer UML sequence diagrams from execution traces. They used the recovered traces and a metamodel to describe UML v1.x sequence diagrams. Rountev et al. [21] described a first algorithm to reverse engineer UML v2.0 sequence diagrams by control-flow analysis. Their approach did not consider data obtained by dynamic analysis and thus is limited by the accuracy of the control-flow analysis. Briand et al. [3] introduced a complete approach to recover scenario diagrams using execution trace. Their work has inspired our own work.

9 Conclusion

We proposed a 3-step approach to identify behavioral and creational design patterns in source code using dynamic analysis. We described behavioral and creational design patterns in terms of scenario diagrams. Then, we reverse engineered scenario diagrams of a given software systems by means of dynamic analysis through bytecode instrumentation. Finally, we performed design patterns identification using constraint programming by identifying in the scenario diagrams of systems objects and messages conform to (`caller`/`callee`, `follows`, and `create`/`created`) the scenario diagrams of some design patterns. We evaluated our approach on JHotDraw with the `Visitor` design patterns to show its precision and recall.

Future work includes merging scenario diagrams to obtain sequence diagrams; using abstraction mechanisms that can reduce the size of execution trace without loss of data relevant to the identification of design patterns; adding new constraints and improving the CSP of the design patterns to obtain higher precision without impacting recall; evaluating our approach on larger systems; combining this approach with a previous structural approach.

References

[1] Giuliano Antoniol, Gerardo Casazza, Massimiliano di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59:181–196, November 2001.

[2] Apache Jakarta Project. *Byte Code Engineering Library*, June 2006.

[3] Lionel Briand, Yvan Labiche, and Johanne Leduc. Towards the reverse engineering of UML sequence diagrams for distributed Java software. *Transactions on Software Engineering*, 32(9), September 2006.

[4] Lionel Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. *Proceedings of the* 10^{th} *Working Conference on Reverse Engineering*, pages 57–66, November 2003.

[5] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[7] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[8] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *Proceedings of the* 39^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[9] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[10] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *Proceedings of the* 1^{st} *IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[11] Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In Hartmut Ehrig, Bernd J. Krämer, and Atila Ertas, editors, *proceedings the* 6^{th} *world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.

[12] Dirk Heuzeroth, Welf Löwe, and Stefan Mandel. Generating design pattern detectors from pattern specifications. In *18th IEEE International Conference on Automated Software Engineering (ASE) 2003*. IEEE, 2003.

[13] Narendra Jussien and Vincent Barichard. The PaLM system: Explanation-based constraint programming. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.

[14] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[15] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[16] Jörg Niere. Fuzzy logic based interactive recovery of software design. Presented at the ICSE Doctoral Symposium, May 2002.

[17] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004.

[18] Wim De Pauw, Doug Kimelman, and John M. Vlissides. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821, pages 163–182. Springer-Verlag, July 1994.

[19] Niklas Pettersson and Welf Lowe. Efficient and accurate software pattern detection. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 317–326, Washington, DC, USA, 2006. IEEE Computer Society.

[20] Alex Quilici, Quing Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *journal of Automated Software Engineering*, 5(3):347–372, July 1997.

[21] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. *Proceedings of the* 6^{th} *Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, September 2005.

[22] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In Bill Scherlis, editor, *proceedings of* 5^{th} *international symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.

[23] Doaa M. Shawky, Salwa K. Abd-El-Hafiz, and Abdel-Latif El-Sedeek. A dynamic approach for the identification of object-oriented design patterns. *Proceedings of the* 2^{nd} *International Conference on Software Engineering*, pages 138–143, February 2005.

[24] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11), November 2006.

[25] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

Verifying Business Processes Extracted from E-Commerce Systems Using Dynamic Analysis

King Chun Foo[1], Jin Guo[2] and Ying Zou[1]

Department of Electrical and Computer Engineering[1]
Queen's University
Kingston, Ontario, Canada
3kcdf@qlink.queensu.ca, ying.zou@queensu.ca

School of Computing[2]
Queen's University
Kingston, Ontario, Canada
guojin@cs.queensu.ca

Abstract

E-commerce systems must react in real-time to user inputs and business rules. For the purpose of re-documentation, static analysis is often adopted to recover business processes implemented in e-commerce systems. However, static analysis fails to recover the complete tasks in business processes due to the dynamic nature of e-commerce systems. To improve the accuracy of recovered business processes, we devise dynamic analysis techniques which trace the execution of processes. We recover usage scenarios from the execution logs and use them to verify the business processes recovered using static analysis. We verify the effectiveness of our proposed approach through a case study on OFBiz e-commerce applications.

1. Introduction

Most e-commerce systems are constructed using three-tier architecture, which includes user interface (UI) tier, business logic tier, and database tier. E-commerce systems are highly dynamic. They react to users' requests in real-time and generate results according to user's selections, business rules and status data stored in databases. For the example of an on-line bookstore, a book can be placed in the shopping cart only if the book in stock. An e-commerce system implements a collection of business processes that describe the operations provided by an organization. Specifically, a business process consists of tasks, control flows, data and participants. A task is a unit of work, which can be executed either automatically in the business logic tier (i.e., back-end components) or require human interactions through UIs. For example, a book purchase business process can contain "selecting a book" task, "select a payment method" task and "buying a book" task. Control flows describe the task execution paths [4], such as sequential, alternative and parallel paths.

Business processes are often optimized to reduce the cost of business operations and improve the quality of services provided by an organization. Business process optimization requires updating the source code to accommodate the modified part of the business processes. When performing code updates, developers need to identify the portion of the code to change. However, locating the code blocks corresponding to a particular feature or change is a difficult process without up-to-date documentation.

In previous research [2][3], we applied static analysis techniques to examine source code and reason over all possible behaviors that might arise during run-time, in order to recover business processes in e-commerce systems. To reduce the complexity of representing the recovered business processes, our previous work [3] represents the business processes in terms of two abstraction levels: high-level and low business processes. High-level business processes give an overview of business operations and contain tasks which require human interactions or which are executed by components in the business logic tier. Low-level business processes contain the details steps performed in the back-end components. Separating complex system structure can provide a clear view of the overall structure of an ecommerce application while hiding processing details. However, static analysis cannot capture the business tasks executed under the conditions determined at run-time. For the example of an on-line bookstore, administrator can edit or add books to the system. The add or edit operations can be initiated by the "create" link or "edit" link in the book management page. Both links will direct the user to a new page. Depending on which link is pressed, one of the two possible operations will take place. The exact operation executed will be dictated by the run-

time parameter supplied by the user. Using static analysis, we can only extract information that either links would lead to two possible operations. However, the exact operation that will be invoked can only be determined through dynamic analysis where actual value is being used to test the system.

To improve the accuracy of recovered business processes, we aim to integrate static analysis with dynamic analysis for recovering business processes. In this paper, we use dynamic analysis techniques to observe the behaviors of a system at run-time. We also use the result of the dynamic analysis to verify and enhance business processes recovered from static analysis. More specifically, the static analysis identifies business processes from the source code of three tiers of the e-commerce application. As an extension to our previous work [3] we refine our techniques for recovering high-level business processes using UI design patterns, which describe the best practices to implement UI functionalities. The UI design patterns are used to separate multiple business processes implemented in the same UI screens.

The dynamic analysis records how a user would normally interact with the e-commerce applications. Usage scenarios are generated to represent the steps that a user needs to perform in order to complete business processes. We record the tasks performed throughout the three tiers when a user interacts with the systems to fulfill business processes. When a user conducts a particular usage scenario, information regarding each step in the scenario is recorded. Since static analysis may not be capable of recovering business processes correctly when run-time information is needed, therefore, we match the tasks from business processes recovered using static analysis and the usage scenarios traced using dynamic analysis. Depending on the results of the matching, we can determine whether a recovered business process is complete or incomplete.

The rest of the paper is organized as follows. Section 2 presents techniques for recovering business processes using UI design patterns. Section 3 presents the techniques for identifying usage scenarios. Section 4 discusses the case studies. Section 5 gives a brief overview of related work. Finally, Section 6 concludes our work and discusses future work.

2. Recovering Business Processes using Static Analysis

We recover business processes from the three tiers of e-commerce systems using static analysis. Recovering business processes from UIs is a challenging task, since the UI is often developed without referring to the underlying business process specifications. The structure of a UI can be complex due to the hyperlinks and hierarchical structures of the widgets (e.g., buttons, tables, and trees in UIs). The hyperlinks can connect one page to other pages with arbitrary orders. Therefore, it is difficult to identify the control flow constructs (e.g., sequence, alternative and parallel) between the functionality delivered in different pages in an e-commerce application. It is also hard to determine the starting point of a business process from the interwoven page links.

The hierarchical structure of a widget complicates the identification of a task with appropriate granularity. For example as shown in Figure 1, a table widget contains multiple cells in different rows and columns. Each cell can further encapsulate tables, hyperlinks, buttons and selection lists. Many possible tasks with different level of granularity can be identified from the UI widgets. For example, the entire table can be considered as a task that displays the result of a search operation. A button widget that triggers a back-end service could be considered as a task with low-level details. However, it is difficult to understand a business process with excessive low-level detailed tasks since a business process is intended to capture high level business operations.

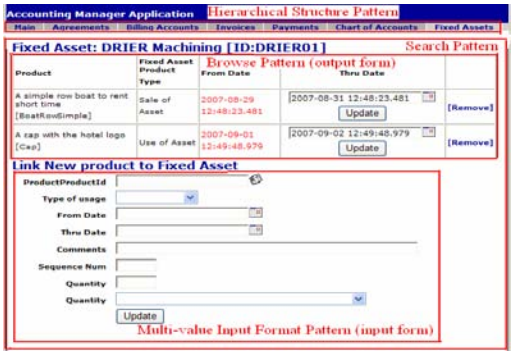


Figure 1. Example UI Page of Sequoia ERP [11]

To overcome the complexity of UI implementation, we use UI design patterns to abstract the structure of the UIs and capture the tasks and controls flows from the structure of pages, and the dependencies among pages. We classify UI design patterns into three categories according to the structure of the UI:

1) Hierarchical structure patterns: are used to organize the overall structure of a UI and group different functionalities of a UI into subsystems. Each subsystem allows a user to work on particular functionality. For example, as shown in Figure 1, the multiple tabs depicted on the top of the screen allow a user to select different subsystems (e.g., agreement, accounting, and catalog) to work on.

This tab structure is an instance of the hierarchical structure pattern. The first screen displayed after selecting a subsystem from the tabs suggests the starting point of a business process.

2) Navigation patterns: describe the navigational structure of UI pages. The navigational structure allows a user to accomplish one business process by navigating through different pages. Generally, a business process can be implemented within one page or can be accomplished across multiple pages. To improve the usability of the UI, different UI navigation patterns are often adopted to guide users through the UI pages. For example, a wizard pattern guides a user to complete tasks in a step by step fashion. We consider the operations fulfilled in the wizard pattern as a business process and we map each operation in the wizard pattern to a task in a business process. The order of each step is converted to a sequential control structure in the business process.
3) Behavioral patterns: characterize functional units delivered in a page. For example as shown in Figure 1, a multi-value input format pattern [10] encapsulates multiple input widgets that take input data from a user; a browse pattern [10] presents a set of ordered data as the output from the back-end components; a search pattern [10] allows a user to specify search criteria, and to review the results of a search. We locate the tasks by identifying the behavior patterns in a UI page.

3. Verifying Business Processes using Dynamic Analysis

The goal of dynamic analysis is to generate a sequence of business tasks which represent the processing steps for fulfilling a business process. A business task, however, can be implemented as code fragments with utility code and other non-business related task in between. Normally, a business task is associated with business data accesses and manipulations. Simply recording all code lines that involve business data would quickly lead to the creation of large log files that is challenging to analyze. Utility classes provide internal basic operations which facilitate the completion of a business process. These non-business logics do not contribute to business processes and hence, should not be logged.

To filter non-business logic, we define criteria to guide the insertion of instrumentation code into the different tiers of an e-commerce application. We define a set of rules to trace the execution of business tasks using code instrumentation in all tiers. Aspect oriented programming can be applied to insert the probes automatically at the location where the rules can be applied. The recorded information from each tier is merged and sorted by access time in order to create a complete usage scenario of a business process.

A potential drawback of dynamic analysis is that it requires a large set of test cases to ensure it covers all execution path of the application. The effort of generating such test case suites makes it impractical for applying dynamic analysis to large software such as e-commerce application. Instead of using dynamic analysis to recover business processes, we apply dynamic analysis to verify the results of static analysis and identify possible mistakes in the business processes generated from static analysis.

3.1 Instrumenting the User Interface Tier

The UI tier is responsible for generating page transitions and collecting user's inputs. This tier is the starting and ending point for all business processes. Upon collecting all necessary information from the user, the UI tier forwards the request of the operation to the business logic tier. We identify two rules for signaling the starting and ending point of a business process.

Rule 1. Each business task is initiated by a user's interaction with the UI. We log each user request as well as the input data for the request.

Rule 2. Upon completion of the user's request of a business task, a new UI screen containing the result of the request is generated. This signifies the end of the business processes and is recorded.

3.2 Instrumenting Business Logic Tier

The business logic tier contains the functional algorithms which process requests from the UI tier using information from the databases. While the other two tiers remain unchanged most of the time, the business logic tier is often updated to reflect changes in the business processes. We identify a preliminary set of rules for instrumenting the business logic tier.

```
1 Public static createPayment(Object paymentType){
2            createPaymentMethod("Credit", null);
3 }
4 Public static createPaymentMethod(Object paymentType,
Object paymentDate){
5            .....
6 }
```

Figure 2. Implementation to support previous releases

Rule 1. To provide support to the previous releases of a system, the implementation of a function may contain code which transfers the program control to

a newer implementation of the function. As shown in Figure 2, the implementation for *createPayment* from line 1 to 3 contains only an invocation of the newer version of the function, namely, *createPaymentMethod*. Since the program control is being transferred to another method, it is assumed that the method being transferred to contain all business logics. Therefore, the instrumentation should start at the method being invoked. For example as illustrated in Figure 2, the probe for tracing the execution of the function (i.e., a business task) is inserted at line 4.

Rule 2. Identify the starting point of a business task which is completed by invoking multiple methods in source code. In most e-commerce systems, business tasks are declared as services in an interface file and implemented in a different file. By analyzing the service declaration, a precise starting point of the implementation of a business task is identified.

Rule 3. Identify business data which govern the purpose of the business tasks. The business data is passed from the UIs and is used to implement business tasks and enforce business rules that specify the conditions for executing business tasks. To generate a meaningful trace, only the code lines that involve business data validation or modification should be recorded. In later cases, data modification is done by populating variables with business data. If the variable is modified multiple times, we only record its last assignment as this is the final value that would be stored in the database.

3.3 Comparing Results from the Static Analysis and Dynamic Analysis

Users often perform multiple business processes in one session. To detect each business process, we analyze the traces produced from all users' requests made during a session. The usage scenarios can be recovered from the traces. Specifically, a usage scenario captures the tasks that are carried out to react to user's requests. Different from recovering a business process, the control flow constructs (e.g., sequential, alternative and parallel execution paths) are not included in a usage scenario.

For each trace records generated by dynamic analysis, we assume that the user completes each operation sequentially without interleaving. To compare the results from both analyses, we match the tasks identified from both techniques in sequential order. There are three possible outcomes through the comparison.

1) If a task is present in the results of both analyses, then the task is verified in the recovered business processes.
2) If a task is present in the dynamic usage scenario of dynamic analysis but not in the recovered business processes from the static analysis, we would treat it as a missed task in which static analysis failed to recover a business task. We complement the static analysis result with the missing task found in dynamic analysis result.
3) If a task is present in static analysis but not in dynamic analysis, we would consider it as a possible misidentified task and we can manually analyze the corresponding code block to determine if it is a business task.

4. Case Study

To demonstrate the effectiveness of our proposed approach, we performed case studies on Sequoia ERP [11], a variant of the Open For Business project (OFBiz). The system is implemented in Java and a proprietary scripting language, called Mini-Language.

We have instrumented the Sequoia by applying the rules discussed in Section 3 on four Sequoia ERP subsystems: catalog, facility, marketing, and workeffort. Using static analysis, we recovered 1) high-level business processes from the scripting code for UI pages and the XML scripting code, 2) low-level business processes from the Java source code. To produce the traces, we recruited an undergraduate student to use the system. The undergraduate student studied the user guides and on-line demos of the system before the experiment.

We developed a prototype tool to recover usage scenarios that have the same starting points as the business processes recovered using static analysis. Such usage scenarios serve as the basis for the verification.

We compare the results by matching the name of tasks and starting points of the process in the results of both analyses. We count the number of missed tasks and misidentified tasks based on the criteria mentioned in section 3.3. We calculate the recall rate and precision using Eqn 1 and Eqn 2.

$$precision = \frac{\#of\ \ identified\ \ tasks - \#of\ \ misidentified\ \ tasks}{\#of\ \ identified\ \ tasks} \quad \text{(Eqn 1)}$$

$$recall = \frac{\#of\ \ identified\ \ tasks - \#of\ \ missed\ \ tasks}{\#of\ \ identified\ \ tasks} \quad \text{(Eqn 2)}$$

Table 1 summarizes the high-level business processes recovered using static analysis from the four studied subsystems in Sequoia ERP. We have recovered 116 business processes and 233 unique tasks

in total. We calculate the precision and recall for misidentified tasks and missed tasks using Eqn 1 and Eqn 2 respectively. The mis-identified tasks should be removed from the recovered business processes. The result of the comparison is shown in Table 2. Comparing the usage scenarios recovered from the dynamic analysis, we locate 4 business tasks that were missed using static analysis. No misidentified tasks were found. The precision and recall are 100% and 98.3% respectively as shown in the Table 2.

Table 1. High-Level Business Processes from Sequoia

Subsystem	# of Workflows	# of Tasks
catalog	66	133
facility	30	62
marketing	8	10
workeffort	12	28
Total	**116**	**233**

Table 2. Precision and Recall for the Static Analysis

	# of Ident. Task	# of Mis-ident. Task	# of Missed Tasks	Precision	Recall
High-Level Process	233	0	4	100%	98.3%
Low-Level Process	195	46	32	76.4%	83.6%

For the low-level business processes, our static analysis techniques missed 32 tasks and mis-identified 46 tasks. We found the precision for recovering low-level process is 76.4% using static analysis. Upon examining the result, we found that the misidentified tasks were mostly related to debugging statements used in the system. We use the results of the dynamic analysis to add the missed tasks and correct the mis-identified tasks. .

5. Related Work

Static analysis is adopted in previous research [1][2][3] to recover business processes. Huang *et al.* [1] propose an approach which depends on variable classification to recover business rules. Zou *et al.* [2] describe a model-driven business process recovery framework using heuristic rules. Hang and Zou [3] produce a collection of complete business processes from the three-tier of e-commerce systems and visualize them using commercial business process modeling tools. Nevertheless, these approaches [1][2][3] require manual verification of the results, and ignore the design structure of UIs when recovering tasks that deliver single unit of work. In this paper we use the dynamic analysis to automatically verify the results from static analysis and enhance the task identification using UI design patterns.

Aalst *et al.* [5][6] recover process models from execution logs. Their research effort is used to generate logs. Feature location is a process to identify the parts of source code corresponding to specific functionalities [7]. Greevy *et al.* [8] map features to software entities using dynamic analysis. Kuhn *et al.* [9] complement dynamic analysis with latent semantic indexing to identify related features in programming language code. In our approach, the detection of UI patterns relies on the static analysis techniques. However, our approach locates tasks using the structures and designs in the UI implementation.

Deursen *et al.* [12] used lexical analysis to track database usage for rapid system understanding purposes. This technique relies on explicit declaration of SQL table or when such information is not available, record definition that is used to read/write entry to database is used instead. The result generated by lexical analysis is only an approximate. In our approach, we based our dynamic analysis on business data access. However, the identification of business data in our approach starts at the UI level and rather than by analyzing the database structure.

6. Conclusion

In this paper, we propose techniques for recovering business processes from e-commerce applications and for verifying the recovered processes using dynamic analysis. To identify tasks with appropriate granularity and separate different business processes, we use UI design patterns. The patterns abstract the UI structures and recover tasks that can deliver a single unit of functionality. The usage scenarios of a user are recovered using dynamic analysis and are compared with the results from static analysis. The case study on an open source e-commerce system demonstrates the feasibility of our techniques.

The limitation with our proposed approach is that manual code instrumentation is required to carry out dynamic analysis which can be difficult when working with large piece of software. Furthermore, manual identification is needed when possible false positive is suggested by dynamic analysis. In the future, we plan to instrument the code automatically.

Reference

[1] H. Huang, W.T. Tsai, S. Bhattacharya, X.P. Chen, Y. Wang, and J. Sun. “Business rule extraction techniques for COBOL programs”, Journal of Software Maintenance 1998, 10(1):3–35.

[2] Y. Zou, T. C. Lau, K. Kontogiannis, T. Tong, and R. McKegney. "Model Driven Business Process Recovery", In Proceedings of Working Conference on Reverse Engineering, 2004.
[3] M.K. Hang and Y. Zou, "Recovering Workflows from Multi Tiered E-commerce Systems", Proceedings of International Conference on Program Comprehension, Banff, July 2007. pp.198-207
[4] H. Schmid and G. Rossi, "Modeling and Designing Processes in E-Commerce Applications", IEEE Internet Computing, January/February 2004.
[5] W.M.P. van der Aalst, T. Weijters, and L. Maruster, "Workflow Mining: Discovering Process Models from Event Logs", IEEE Transactions on Knowledge and Data Engineering, v.16 n.9, pp.1128–1142, Sep. 2004
[6] A.J.M.M. Weijters and W.M.P. van der Aalst, "Process mining: discovering workflow models from event-based data", in: Proceedings of the Belgium-Netherlands Conference on Artificial Intelligence, 2001, pp. 283–290.
[7] D. Poshyvanyk, A. Marcus, and V. Rajlich, "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval", IEEE Transactions on Software Engineering, v.33, n.6, June 2007.
[8] O. Greevy and S. Ducasse, "Correlating features and code using a compact two-sided trace analysis approach", Proceedings of CSMR 2005, 2005, pp.314–323.
[9] A. Kuhn, O. Greevy, and T. Gîrba, "Applying semantic analysis to feature execution traces", Proceedings of PCODA, Nov. 2005, pp. 48–53..
[10] D. Sinnig, "The Complicity of Patterns and Model-Based UI Development", Mater thesis in Department of Computer Science, University of Concordia, Montoreal, Canada, 2004.
[11] http://ofbiz.apache.org/
[12] A. van Deursen, and T. Kuipers, "Rapid System Understanding: Two COBOL Case Studies", IWPC 1998