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

PCODA'06

2nd International Workshop on

Program Comprehension through Dynamic Analysis

co-located with the 13th Working Conference on Reverse Engineering (WCRE'06)

Benevento, Italy
23rd October 2006

Technical report 2006-11
Universiteit Antwerpen
Department of Mathematics & Computer Science
Middelheimlaan 1, 2020 Antwerpen, Belgium

Contents

Patterns & behavior

A Hybrid Analysis Framework to Evaluate Runtime Behavior of OO Systems………. ……………1
Azin Ashkan, Ladan Tahvildari

Summarizing Traces as Signals in Time……………………………………………………………….. 6
Adrian Kuhn, Orla Greevy

An Environment for Pattern based Dynamic Analysis of Software Systems……………………….. 12
Kamran Sartipi, Hossein Safyallah

Reverse engineering

Aiding in the Comprehension of Testsuites…………………………………………………………. 17
Bas Cornelissen, Arie van Deursen, Leon Moonen

A Lightweight Approach to Determining the Adequacy of Tests as Documentation………………..21
Joris Van Geet, Andy Zaidman

High level dynamic analysis views

Combining Reverse Engineering Techniques for Product Lines…………………………………….27
Dharmalingam Ganesan, Isabel John, Jens Knodel

Higher Abstractions for Dynamic Analysis…………………………………………………………. 32
Marcus Denker, Orla Greevy, Michele Lanza

Capturing How Objects Flow At Runtime…………………………………………………………...39
Adrian Lienhard, Stéphane Ducasse, Tudor Gırba and Oscar Nierstrasz

Program Chairs

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

Abdelwahab Hamou-Lhadj
School of Information Technology and Engineering
University of Ottawa
Canada
ahamou@site.uottawa.ca

Andy Zaidman
Software Evolution Research Lab (SWERL)
Delft University of Technology, The Netherlands
&
Lab On Reengineering (LORE)
University of Antwerp, Belgium
a.e.zaidman@tudelft.nl

Program Committee

Gabriela Arevalo
LIRMM, Montpellier, France
Lewis Baumstark
University of West Georgia, USA
Bas Cornelissen
Delft Univ. of Technology
Serge Demeyer
University of Antwerp, Belgium
Stephane Ducasse
University of Savoie, France
Tudor Girba
University of Berne, Switzerland
Orla Greevy
University of Berne, Switzerland
Abdelwahab Hamou-Lhadj
University of Ottawa, Canada
Andrew Malton
University of Waterloo, Canada
Arie van Deursen
Delft Univ. of Technology
Andy Zaidman
Delft Univ. of Technology, The Netherlands &
University of Antwerp, Belgium

A Hybrid Analysis Framework to Evaluate Runtime Behavior of OO Systems

Azin Ashkan and Ladan Tahvildari
Department of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada, N2L 3G1
{aashkan, ltahvild}@uwaterloo.ca

Abstract

Since software is often deployed in safety critical applications, there is a constant need to know whether a system is behaving correctly and reliably in its environment. This research work integrates concepts of static and dynamic analyses to verify the behavioral correctness of a Java software system based on certain safety properties. We also apply the proposed framework on a sample Java application.

1 Introduction

Software plays an important role in our economy, government, and military, and since software is often deployed in safety critical applications, there is a constant need to know whether a system is behaving reliably in its environment. Traditional methods, testing and verification [3], are not enough to guarantee that the current execution of a running system is correct. Testing may scale well and check implementation directly, but it is mostly informal and it does not guarantee completeness. Verification is formal and may guarantee completeness, but it does not scale well and deals mostly with design instead of implementation. An approach of continuously monitoring and checking a running system with respect to particular specifications can be used to fill the gap between these two approaches.

Research has followed three directions for runtime monitoring and analysis. The first direction modifies byte code of a target program (byte code instrumentation). [7] uses a script-driven automated byte code instrumentation of Java programs and sends the monitored information to an observer while Java-Mac [9] provides a full automated process and modifies the byte code too. The second direction of approaches instruments source code. [10] uses a specification logic to define security policies and instruments the source code. [4] embeds temporal logic assertions to the source code. DynaMICs [6] specifies constraints as event-condition-action rules and checks them anytime a change is made to the values of constraint variables. All these approaches need to change the source code and therefore they need a special compiler for every language they use. The third direction builds a monitoring module and configures it to generate events at desirable points in the code. The program code is not modified during the process. JassDA [2] is a framework which generates events during the program runtime and analyzes them using a CSP-like specification.

Our research work follows the third direction. It proposes a framework that integrates concepts of reverse engineering and runtime monitoring. The framework is a synergy between static and dynamic analyses. It confirms that a target Java system is running correctly with respect to the OCL [12] based specification of certain safety properties.

The remainder of this paper is organized as follows. Section 2 gives an overview of the proposed framework. Some concepts, which are used in the paper, are defined in Section 3. In Section 4, we introduce our approach and, in Section 5, we present the results obtained by applying the framework on a Java case study. Finally, Section 6 summarizes the contributions of this work and outlines directions for further research.

2 The Proposed Framework

As illustrated in Figure 1, the proposed framework is comprised of four stages:

- **Extracting Architecture:** Reverse engineering techniques are used to extract a meta-model of a Java legacy system by utilizing graph representation of the system model.
- **Seeding Objectives:** Safety properties are defined in OCL (Object Constraint Language) as specifications for the behavioral analysis. Information required for filtering runtime events is also obtained in this stage.
- **Monitoring Runtime:** Runtime behavior of the system is monitored according to the output of the previous stages.
- **Analysis:** Information obtained from the last stage is analyzed and verified based on safety properties.

The first and the second stages are static while the third and the fourth stages are dynamic. The following sections elaborate further on this process.

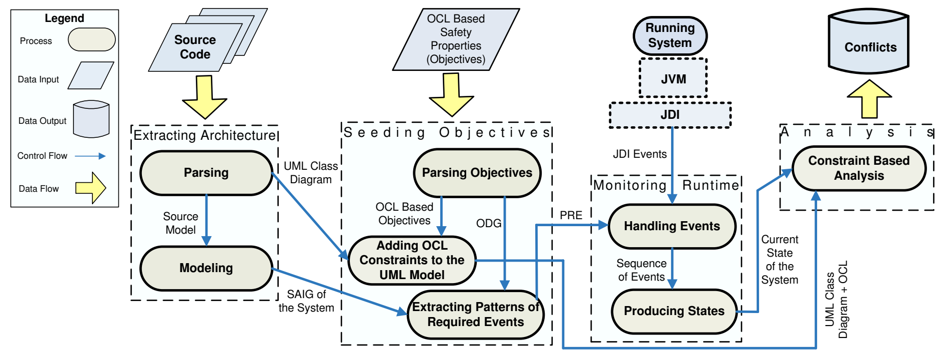


Figure 1. The Framework Architecture

3 Terminology and Concepts

The following definitions and terminologies will help readers understand the concepts in details.

Definition 3.1 *An **Attributed Graph** is a directed graph $AG(V, E)$ in which:*

- $V = V_m \cup V_d$ *that* V_m *and* V_d *are called **main** and **data** nodes respectively*
- $E = E_l \cup E_a$ *that* E_l *and* E_a *are called **link** and **attribute** edges respectively such that:*
  - ⋆ $E_l = \{(u, v) \mid u, v \in V_m\}$
  - ⋆ $E_a = \{(u, v) \mid (u \in V_m) \wedge (v \in V_d)\}$

Terminology 1 *Consider set of all languages in the world as L and set of all data types of these languages as T_L. An object-oriented software system S is implemented in a language l where $l \in L$. We denote any class type and class attribute of the language l with c_t and c_a classifiers respectively. T_l is the set of all data types of l where $T_l \subset T_L$. For example, T_{Java} = {int, short, long, double, float, boolean, byte, char, string}.*

Definition 3.2 *A set of relations R over the system S in the language l is defined on a set D, where $D = \{c_t, c_a\} \cup T_l$. Considering t_i is a data type of language l where $t_i \in T_l$ and $i \in \{1, ..., |T_l|\}$, R includes:*

- `depends-on`*: the dependency relation (such as association and aggregation according to the UML 2.0 specification) between any two class entities of S. It is denoted as c_t* `depends-on` *c_t*
- `has-attr`*: the attribute ownership of a class. It is denoted as c_t* `has-attr` *c_a*
- `has-type`*: the relation between a data type and an attribute defined over that type. It is denoted as c_a* `has-type` *t_l*

Definition 3.3 *A **Schema Attributed Abstract Graph** of a language l, namely $SAAG(V, E)$, is an AG over the set of data types T_l such that:*

- $V_m = \{c_t, c_a\}$
- $V_d = T_l$
- $E_l = \{(c_t, v) \mid [(c_t \texttt{ depends-on } v) \textit{ iff } (v = c_t)] \vee [(c_t \texttt{ has-attr } v) \textit{ iff } (v = c_a)]\}$
- $E_a = \{(c_a, t) \mid (t \in T_l) \wedge (c_a \texttt{ has-type } t)\}$

Definition 3.4 *A **Schema Attributed Instance Graph** of the system S, namely $SAIG(V, E)$, is defined over the $SAAG$ where S is implemented in the same language l. The $SAIG(V, E)$ is equipped with a pair of morphism functions $\mu_V : V^{SAIG} \to V^{SAAG}$ and $\mu_E : E^{SAIG} \to E^{SAAG}$ such that:*

- *V_m is the set of all classes or the attributes of the classes*
- $V_d \subseteq {V_d}^{SAAG}$
- $E_l \subseteq {E_l}^{SAAG}$
- $E_a \subseteq {E_a}^{SAAG}$
- *We know that for an AG(V, E), $V = V_m \cup V_d$ and $E = E_l \cup E_a$. While any $SAIG$ is considered as AG, morphism functions μ_V and μ_E can be defined as follows:*
  - ⋆ $\forall\, v \in V$:
    - ◇ *if $v \in V_m$ and v is a class of the system S, then $\mu_V(v) = c_t$ where $c_t \in V^{SAAG}$*
    - ◇ *if $v \in V_m$ and v is a class attribute of the system S, then $\mu_V(v) = c_a$ where $c_a \in V^{SAAG}$*
    - ◇ *if $v \in V_d$, then $\mu_V(v) = v$ where $V_d \subseteq {V_d}^{SAAG}$ and $v \in V^{SAAG}$*
  - ⋆ $\forall\, e \in E : \mu_E(e) = e$ *where* $E \subseteq E^{SAAG}$ *and* $e \in E^{SAAG}$

The SAIG has a lower level of abstraction than the SAAG and is actually instantiated from it.

Definition 3.5 *An **Adjacency Function** of a directed graph $G(V, E)$, namely $Adj^G : V \to P(V)$, returns the set of out-edge nodes for each node of G where $P(V)$ or 2^V, namely power set of V, is the set of all subsets of V.*

Terminology 2 *A **Safety Property** constrains the permitted actions, and, therefore, the permitted state changes of a system [11].*

4 Hybrid Analysis

Runtime monitoring and analysis require certain information about a target system. Such information is obtained through the static analysis in a way that does not modify the source code. For that reason, we have combined static and dynamic analyses in our framework described below.

4.1 Static Analysis

Traditional reverse engineering techniques can extract the structure of the target system while OCL can define some safety properties as the objectives in behavioral analysis.

4.1.1 Extracting Architecture

We have two goals in this stage: i) modelling the structure of the system, and ii) describing the model in a graph representation that could be extendible for future extensions. There are two main processes in this stage:

- **Parsing :** A Java program is parsed to extract abstract source code models which describe the Java class, package, and source file. These models are essential for representing the system at the source code level. We form them as a UML-compliant model on which the OCL constraints will be applied later.
- **Modeling :** This process generates a SAIG (Schema Attributed Instance Graph) of the system based on the obtained source code models. This graph is our extension to attributed graphs [5] as defined in definition 3.4. We export the SAIG as XML documents that can be easily queried and traversed in subsequent stages.

4.1.2 Seeding Objectives

The runtime information we receive, can be very large for the analysis. We must reduce this by narrowing down to the required information. Our monitoring technique has access to all objects, their attributes, methods entrances, methods exits, and so on. We reduce them to those objects and attributes which are involved in OCL-based safety properties. Therefore, in this stage, we have two main goals: i) identifying safety properties as OCL rules on the UML model of the system, and ii) extracting patterns of required events to reduce and filter generated events during monitoring the runtime. There are three main processes in this stage:

- **Adding OCL constraints to the UML model :** Here we specify the objectives of the behavioral analysis. These are the safety properties on the behavior of the system which are described as OCL rules.
- **Parsing objectives :** The OCL-based safety properties are parsed in this stage to extract objects and attributes on which they depend. The extracted information is represented in a simple dependency graph called the Objective Dependency Graph (ODG).
- **Extracting patterns of required events :** Once the ODG is constructed, a model of required attributes for dynamic analysis is available. It is still required to populate this model on the structure of the system to construct another model called the *Pattern of Required Events (PRE)* through algorithm 4.1. The PRE helps to filter generated event traces during monitoring the runtime and reduce them to what is needed by the analysis component. As is shown below, the SAIG is traversed in depth-first order based on the ODG to build the attributes of the PRE. For each common node v between the ODG and the SAIG, any out-edge neighbor (v') of v in the SAIG and the edge between them (including v) are copied to the PRE ((v, v') is added to the PRE). If v' is a *main* node, it will be also considered for further traverse. Otherwise, if it is a *data* node, there is no need to do so and the $flag2VisitingNode$ for this node is set as true since it is only a data node, and has been already copied to the PRE as the data type of v.

Algorithm 4.1: Extracting Pattern of Required Events

Input: $SAIG$, Adj^{SAIG}, and ODG of an OO system S
Output: PRE to monitor the behavior of S
$PRE \leftarrow \emptyset$
foreach $v \in V^{SAIG}$ **do**
    $flag2VisitingNode(v) \leftarrow false$
while ($\exists u \in {V_m}^{SAIG}$) *and* ($\sim flag2VisitingNode(u)$)
do
    $traverse(u)$

Procedure $traverse(v)$
begin
    $flag2VisitingNode(v) \leftarrow true$
    if $v \in V^{ODG}$ **then**
        foreach $v' \in Adj^{SAIG}(v)$ **do**
            if $\sim flagVisitingNode(v')$ **then**
                if $v' \in {V_d}^{SAIG}$ **then**
                    $PRE \leftarrow PRE \cup \{(v, v')\}$
                    $flag2VisitingNode(v') \leftarrow true$
                else
                    if $v' \in V^{ODG}$ **then**
                        $PRE \leftarrow PRE \cup \{(v, v')\}$
                        $traverse(v')$
                    else
                        $flag2VisitingNode(v') \leftarrow true$
end

4.2 Dynamic Analysis

Our monitoring component generates runtime event traces of a system without the need to do any type of instrumentation. Afterwards, a back-end analysis component carries out the processing of these traces with respect to the safety properties. At this time, the monitoring component works with Java based systems, however the modular architecture of the framework allows to replace or equip this component in a way that other OO languages can be supported.

4.2.1 Monitoring

JPDA [13] is a multi-tiered debugging architecture contained first within Sun Microsystem Java 2 SDK version 1.4.0. It consists of a Java Virtual Machine Debug Interface (JVMDI), and a Java Debug Interface (JDI), as well as a protocol, Java Debug Wire Protocol (JDWP). We have used the JDI, which defines and requests information at the user code level, in order to implement our monitoring component. There are two main processes in this stage:

- **Handling events:** A Java thread has been implemented in this stage which uses the JDI specifications. It invokes the main class of a Java target program that has been passed to it. The JDI helps us to generate a trace of the program's execution by making access to dynamic information. For the time being, our framework monitors the values of attributes and the local variables of primitive types. The reason is that the state of a program is ultimately contained in the attributes and variables of primitive types. The PRE from the previous stage is utilized by this process to filter monitored event traces. Therefore, only those event traces, which match with the specifications of the PRE, are passed to the next process.
- **Producing states:** In this stage, the sequence of generated traces is formed in a format which is acceptable by the analysis component. Each event represents a state of the system at the time that it was extracted from the running program.

4.2.2 Analysis

There is one process in this stage which uses OCL-based constraint analysis on the generated events from the previous stage. We have used an open source tool called USE (UML-based Specification Environment) developed in Bremen University [1]. The main components of this tool are an animator for simulating UML models and an OCL interpreter for constraint checking. System states are snapshots of a running system. We have adopted some parts of the source code of USE to be incorporated into our framework as the analysis component shown in Figure 1. We get the real data coming from the monitoring component to make the snapshots of the running system.

During the running time of our framework, the analysis component runs with the specifications (UML model of the system with the OCL-based safety properties). It starts with an empty system state where no objects and association links exist. The monitoring component provides snapshots of system states for the analysis component. OCL rules are evaluated on those states. If a constraint fails so that a system state is found to be invalid, it will be reported as a *conflict* in the behavior of the system at that time.

5 An Example

This section contains results produced by applying our framework on a small Java application with 300 lines of code. It is a trading system borrowed from [8] and extended based on some requirements that are elaborated further in this section. Two major classes of the system are *Stock* and *Trader*. Some traders instantiated from the *Trader* subscribe in a stock instantiated from the *Stock*. The stock has some items to sell and provides traders with information about each item, including its price, whenever it is ready for market. Each trader decides whether it wants to buy the item or not, based on its trading policy. If yes, it lets the stock know about its decision so that nobody else can buy that item.

5.1 Hybrid Analysis

First of all, the SAIG of our Trading application is extracted and exported as an XML document. The SAIG characteristics for one of the classes of the system, *Trader*, are as follows:

- $V_m = \{Trader, Stock, name, basePrice, tradePrice, itemID\}$
- $V_d = \{int, long, string\}$
- $E_l = \{\texttt{depends-on}, \texttt{has-attr}\}$
- $E_a = \{\texttt{has-type}\}$
- $\mu_V = \{(Trader, c_t), (Stock, c_t), (name, c_a), (basePrice, c_a), (tradePrice, c_a), (itemID, c_a), (int, int), (long, long), (string, string)\}$
- $\mu_E = \{(\texttt{depends-on}, \texttt{depends-on}), (\texttt{has-attr}, \texttt{has-attr}), (\texttt{has-type}, \texttt{has-type})\}$

The objective of our analysis is twofold. First, each trader should buy an item which is under its base price. Second, no two traders can buy the same item. Each objective is defined as an OCL-based safety property. Both properties are applied on the context of the *Trader* class as they only constraint the behavioral property of this class. We name them *TradingPolicy* and *Transaction* respectively, as shown below.

- inv **Transaction**: Trader.allInstances $\rightarrow$ forAll (t_1, t_2: Trader | (t_1 $<>$ t_2) implies $t_1.itemID$ $<>$ $t_2.itemID$)
- inv **TradingPolicy**: self.tradePrice $<=$ self.basePrice

The next stage is to parse the two OCL-based properties to obtain the Objective Dependency Graph (ODG) and export it as an XML document. In such a document, one or more *objective* tags (each corresponding to a safety property) give the dependency information of safety properties. These tags contain one or more *class* tags that contain the names of classes on which the corresponding properties depend. There is a tag, named *dependencies*, after each *class*

tag. It includes the names of attributes (in one or more *attribute* tags) which construct the OCL representation of the safety property corresponding to the parent *objective* tag.

After applying Algorithm 4.1 on the SAIG and the ODG of the system, the PRE is obtained and exported as an XML document. There are one or more *className* tags in this document indicating names of the classes whose particular attributes need to be monitored. Those attributes are shown by *classAttr* tags. Names of these classes and their attributes come from the ODG to the PRE model. Tracing through the SAIG, based on the ODG, helps to extract the package name of each class and the type of each required attribute. Having names of required classes, their package names, required attributes of those classes, and types of those attributes, help the monitoring component to filter and reduce runtime events.

Safety Property	Traders	States	Conflicts
Transaction	2	174	64
	4	223	97
TradingPolicy	2	174	0
	4	223	0

Table 1. Results of the Analysis

The monitoring component invokes the main class of the trading system and generates event traces from the system based on the PRE. Produced states are given to the analysis component. We have done two experiments for a period of time that 50 items are sold to the traders, and they only differ in terms of the the number of traders instantiated in the system. The results of analysis on the two safety properties, presented in Table 1, shows the total number of produced states in each experiment and the number of states with conflicts for each safety property.

5.2 Discussion on the Obtained Results

According to Table 1, there is no conflict reported for the *TradingPolicy* safety property. This is reasonable due to the way the trading policy was implemented for each trader. It was a simple selection algorithm based on a threshold price (called *basePrice*). However, the other property, *Transaction*, has caused conflicts in some states. This means that, there are states during the runtime where at least two traders have bought the same item due to a transactional problem. The reason for such a transactional problem is the interference caused by making access to critical sections mutually exclusive [11]. As Table 1 summarizes, number of transaction conflicts is greater in the experiment with four traders than the experiment with two traders. Due to the mutually exclusive problem, the greater the number of processes (traders), the greater the number of interferences.

6 Conclusions and Future Work

We proposed a framework utilizing a synergy between static and dynamic analyses to confirm that a target Java system is running correctly with respect to the specification of certain safety properties. Two key features of the proposed framework are: i) the source code of a legacy system is not modified, and ii) the most parts of the process are automated. Using the help of reverse engineering techniques through static analysis (generating the PRE), we have reduced the large size of runtime information in dynamic analysis. We narrowed down the information to what is required for the analysis. At this time, the analysis component detects if there is any conflict with a safety property. One possible extension to this research is to report the cause of each conflict in the analysis component. The trading system is a small example, we plan to apply our framework on large Java-based applications to obtain more concrete results. Another future direction is to define an evaluation criterion for our monitoring technique while the evaluation is done manually at this time.

References

[1] A UML-based Specification Environment (USE), 2006. http://www.db.informatik.uni-bremen.de/projects/USE/.

[2] M. Brorkens and M. Moller. JassDA trace assertions, runtime checking the dynamic of Java programs. In *Proceedings of the International Conference on Testing of Communicating Systems*, pages 39–48, 2002.

[3] I. Crnkovic and M. Larsoon. *Building Reliable Component-Based Systems*. Artech House, 2002.

[4] D. Drusinsky. Monitoring temporal rules combined with time series. In *Proceedings of Computer Aided Verification Conference (CAV)*, pages 114–117, 2003.

[5] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In *Proceedings of the 2nd International Conference on Graph Transformation (ICGT)*, pages 161–177, 2004.

[6] A. Q. Gates, S. Roach, O. Mondragon, and N. Delgado. DynaMICs: Comprehensive support for run-time monitoring. *Electronic Notes in Theoretical Computer Science*, 55:1–17, 2001.

[7] K. Havelund and G. Rosu. An overview of the runtime verification tool: Java PathExplorer. *Journal of Formal Methods in System Design*, 24:189–215, 2004.

[8] Immagined cities, 2006. http://cities.lk.net/approco.html.

[9] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-Mac: A run-time assurance approach for Java programs. *Journal of Formal Methods in System Design*, 24:129–155, 2004.

[10] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[11] J. Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3:239–272, 1995.

[12] B. B. Schmid, T. Clark, and J. Warmer. *Object Modeling With the OCL: The Rationale Behind the Object Constraint Language*. Springer-Verlag Berlin Heidelberg, 2002.

[13] Java Platform Debugger Architecture (JPDA), 2006. http://java.sun.com/products/jpda/.

Summarizing Traces as Signals in Time

Adrian Kuhn and Orla Greevy
Software Composition Group
University of Bern, Switzerland
{akuhn, greevy}@iam.unibe.ch

Abstract

One of the key challenges of dynamic analysis approaches is that they imply a huge volume of data, thus making it difficult to extract high level views. In this paper we describe a novel approach to trace summarization by visually representing entire traces as signals in time. Our technique produces a visualization of the complete feature space of a system that fits on one page. The focus of our work is to visually represent individual traces feature behavior. We assume a one-to-one mapping between features and traces. We apply the approach on a case study, and discuss how our visualization supports the reverse engineer to identify patterns in traces of features. Moreover, we show how the visual analysis of our trace signals reveals that assumed one-to-one mappings between features and traces may be flawed.

Keywords: reverse engineering, dynamic analysis, trace summarization, features, visualization.

1. Introduction

Reverse engineering usually implies the abstraction of high level views that represent different aspects of a software system. Object-oriented systems are difficult to understand by browsing the source code due to language features such as inheritance, dynamic binding and polymorphism. The behavior of the system can only be completely determined at runtime. The dynamics of the program in terms of object interactions, associations and collaborations enhance system comprehension [11]. Typically dynamic analysis involves instrumenting a program under investigation to record its runtime events. The context of our dynamic analysis is feature-centric reverse engineering (i.e. we exercise a system's features on an instrumented software system and capture traces of their runtime behavior).

Interpretation of execution traces is difficult due to their sheer size, thus filtering or compressing the data is a crucial step in the construction of high level views. The main challenge of trace summarization is to reduce the volume of data without loss of information that is relevant for a particular analysis goal [9]. Dynamic analysis together with program visualization may be used in debugging, evaluating and improving program performance and in understanding program behavior.

Because of the accuracy and speed with which the human visual system works, graphic representations make it possible for large amounts of information to be displayed in a small space. By making a visual representation for the millions of events that make up the feature traces of an application, quickly discernible relationships and patterns can be obtained.

In this paper, we describe a novel visualization approach for dynamic analysis that draws an analogy between execution traces and signals in time. We use the nesting level to visualize traces as time plots, and provide a visualization that allows up to two dozen feature traces to be displayed simultaneously on a single screen. We show evidence of the visualization's usefulness and how it supports the reverse engineer to interpret and reason about the dynamic information. In particular, we address the following reverse engineering questions:

- *How do we fit a visualization of many traces on one screen?*
- *Can we detect patterns of activity in the traces?*
- *Do our traces reveal flaws in our definition of features?*

We use SmallWiki [5] as an example case study. The same SmallWiki case study has been analyzed in a previous work by Greevy *et al.* with a metrics-based approach [8]. For this paper we traced a total of 18 feature traces.

Structure of the paper. In Section 2 we draw the analogy between traces and times series, and visualize trace signals as time plots. Based on that in Section 3 we introduce a new trace summarization technique. Section 4

Figure 1. An execution trace as signal, showing on the y-axis the nesting level and along the x-axis the sequence of execution; summarized with Monotone Subsequence Summarization using gap size 0.

dives into the details of a case study, while Section 5 discusses the pros and cons of the time series analogy and the findings of our analysis. In Section 6 we provide a brief overview of related work in the fields of dynamic analysis, visualization and summarization of execution traces and time series related work. Finally we outline our conclusions in Section 7.

2. Traces are Signals in Time

As any chronological sequence of partially ordered data points qualifies as a *signal*, we can draw analogy between traces and signal processing. This done, we treat traces as if they were signals in time, which in turn provides us with a rich toolkit of well-established and ready-to-use algorithms from the field of signal processing and time series.

A key benefit of treating traces as signals is that we get time plots for free, as shown in Figure 1. Time plots are well-known from a broad range of applications, such as from the field of meteorology or stock markets. They show the change of a signal over time.

A *feature trace* is a record of the steps a program takes during the execution of a feature. We adopt the definition of a feature as a user-triggerable functionality of a software system [6]. In the case of object-oriented applications, a trace records method calls, whereas for systems implemented in procedural programming languages, it records function calls. In this paper we adopt the object-oriented terminology: we consider each execution step as a message sent from the sender to the receiver, whereupon the receiver executes the method selected by the message.

As a formal definition of a *trace*, we use a chronological sequence T of one or more execution events. The call hierarchy imposes a tree structure on the sequence, each event e_n has zero or more child events, with e_{n+1} as its first child, if any. With this definition, the execution sequence is equivalent to a depth-first traversal of the call hierarchy. Further, to equip the events with a partial order, we define the *nesting level* $\mathrm{Level}(e_n)$ of an event e_n as its depth in the call hierarchy.

Typically, an execution event includes information about execution time. However, in this paper we omit execution times and retain only the order of events. As the rise and fall of the signal is preserved even if all events are spaced equally apart in time, we can ignore execution time of events without loss of generality. The outline of the signal remains the same independent of the interval between its data points. Therefore the summarization technique we present in Section 3 retains its results.

3. How to Summarize Traces

In this section we introduce a trace summarization technique, which is based on the representation of traces as time signals. We introduce a technique called *Monotone Subsequence Summarization*, which makes use of the fact that a trace signal is composed of monotone subsequences separated by pointwise discontinuities.

The structure of a trace signal as defined in the previous section is plain simple: beginning at the starting node the nesting level either

- increases step by step as each event calls its first child or
- stays constant as subsequent children of the same event are called, until
- we reach an event without children, in which case the nesting level suddenly drops as execution continues with the latest sibling of the previous events.

The first two cases are monotonally increasing subsequences, whereas the latter is a pointwise continuity.

To summarize a trace signal, we cut the signal at its pointwise discontinuities into monotone subsequences, and compress each such subsequence into a summarized event chain. Thus the summarization is considerably shorter than the raw trace signal, see Figure 2, and consists of method-call-chains instead of single method calls.

The *Monotone Subsequence Summarization* cuts a trace signal between each two consecutive events where the nesting level does not increase

$$\mathrm{Level}(e_n) \leqslant \mathrm{Level}(e_{n+1})$$

into pieces $c_1 \dots c_m$ and these pieces become the events of the summarized trace. Furthermore, we define $\mathrm{Level}(c_n)$ as the minimal nesting level within the chain c_n.

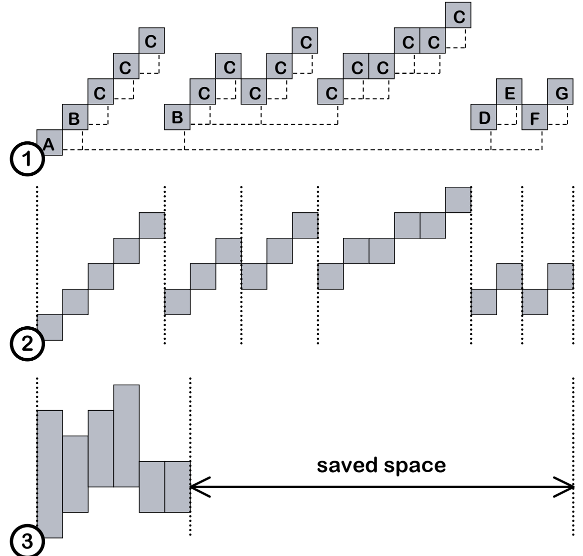


Figure 2. From top to bottom: 1) The outline of a trace, rotated by one quadrant such that time is on the X-axis. 2) We remove method names and retain the nesting level only, each discontinuity is marked with a dotted line. 3) Each monotone subsequence is compressed into one method-call-chain, saving a considerable amount of space.

Using this technique, we reduce the length of a trace signal on average by about 50%. However, we can further improve this by taking into account that not each discontinuity has the same gap size: often the signal drops only by one or two nesting levels, that is the execution stays close to the current chain of method calls, whereas other discontinuities span dozens of nesting levels and thus mark a real break in the execution. Therefore, we allow small gaps within a chain of method calls and refine the above expression as

$$\mathrm{Level}(e_n) - \mathrm{Level}(e_{n+1}) \leqslant \mathrm{gap_size}$$

Using gap_size = 3 it is possible to save up to 90% of the length of a trace signal. In other words even a trace with ten thousand events will fit on one screen.

4. Case study: SmallWiki

For our experiments with the techniques described in the previous sections, we chose *SmallWiki* [13], an open-source, fully object-oriented and extensible Wiki framework. A Wiki is a collaborative web application that allows users to add content, but also allows anyone to edit content. Thus SmallWiki provides features to create, edit and manage hypertext pages on the web.

We identify features of SmalWiki by associating features with the links and entry forms of the SmallWiki pages. We make the assumption that each link or button on a page triggers a distinct feature. We selected 18 distinct user interactions with the SmallWiki application and exercised them on an instrumented system to capture 18 distinct execution traces. The features we chose represent typical user interactions with the application such as login, editing a page or searching a web site. Then we apply trace summarization as described in Section 3, and we represent each feature trace visually as a time plot on one screen.

4.1. Analyzing the time plots of feature traces

In the following paragraphs we describe how we analyzed the time plots of the SmallWiki feature traces and reasoned about these views of feature behavior. As we have access to the developers of SmallWiki, we are able to check the findings of our signal processing analysis techniques with them.

Considering Figure 4, we observe a couple of phenomena exhibited by most or all trace signals of this feature spaces. We recognize that further research in the form of more case studies is required before we can conclude that these phenomena are common to any feature space, but we assume that these observations hold true for most trace signals.

All features share a common introduction. We observe that all features start with the same introduction, see Figure 3 annotation 1. This introduction corresponds to the time plot of the resolveURL feature. This makes sense due to the nature of SmallWiki as a web-based application. Resolving a given URL is the first step to be performed in order to execute a user-initiated feature. This phenomenon is most probably common to any feature space, as most architecture includes some top layer which does some preprocessing before executing the actual feature. This observation reveals that the traces could be further summarized by removing or *factoring out* the common introduction part of the trace.

Shared parts may include variations. Although the same introduction is shared by all trace signals, our analysis reveals variation points. In Figure 4 we see that the introduction sequences of the features copychild, addfolderchild, addpagechild and removechild include a variation point, that is they contain a distinct sequence which is not present in the other traces, see Figure 3 annotation 2.

Specific behavior is restricted to small hot-spots. Our analysis reveals that only a small amount of the over-

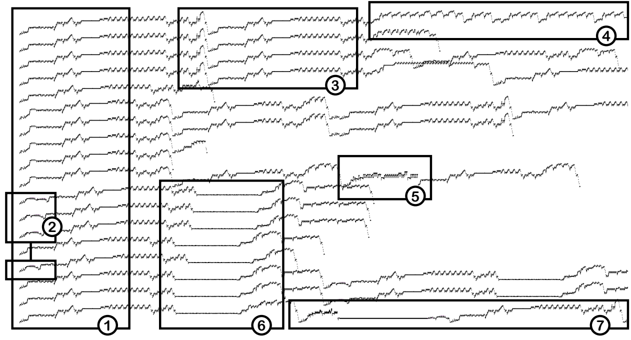


Figure 3. Trace patterns of Figure 4, there is a shared introduction (1) with slight variations (2); recurring patterns (3,6) as well as unique sequences (4,5,7).

all behavior of a feature is specific to one sole feature (that is characterized as *single-feature*[8]). This might be due to the generic nature of the SmallWiki application. However we expect to observe this in other case studies as well, since most applications include sequences of common set-up and common tear-down enclosing the actual functionality of a feature. For example, as we exercised user-triggerable actions of SmallWiki, all involve exercising common functionality to handle the http request-response dialog of a web application. Recent work of de Pauw *et al.* in detection in patterns in traces reveals that the actual number of distinct patterns in execution traces was small. The results of their work revealed only 10 distinct patterns in a 40MB trace [3].

Some features are very similar. Our time series representation of feature traces as shown in Figure 4 reveals which features are closely related (that is they exhibit common patterns of behavior). The features copychild, addfolderchild, addpagechild and removechild are all invoked from the same page in SmallWiki, see Figure 3 annotation 6. We verified our findings with the developers and they confirmed that these features actually exercise the same code.

The features comps, props, stylesheets and edittemplate reveal similar time plots on Figure 4. Once again the developers confirmed our findings, as all these features are concerned with *look-and-feel* aspects of the system.

Not all features are equally similar. Figure 4 reveals that the features addFolder, addPage are similar. The feature *editPage* appears to be similar to the previous two features but then exhibits a strong variation, see Figure 3 annotation 7. The similar parts of these features indicate to the reverse engineer that these features may be conceptually related. The developers confirmed our findings.

5. Discussion

In this section we discuss some open issues and limitations of the applied techniques and the signal analogy itself.

On the choice of the nesting level as Y-axis. To represent traces as signal in time, we plot the nesting level of the execution events on the Y-axis. Even though this yields natural looking time plots, such as those familiar from meteorology or stock markets, we have not yet investigated if the nesting level is the most useful property to discriminate the differences between feature traces. Other information such as method names or arguments may prove to be more useful or better discriminators.

On the one-to-one mapping between features and traces. For this analysis, we assume a one-to-one mapping between feature-traces and features. However, the visualization of the feature space revealed that there is no a one-to-one mapping between features and traces. We need to consider a feature, not an execution trace, as the smallest unit of behavior: traces such as, for example the set of copychild, addfolderchild, addpagechild and removechild seem to implement variations of the same feature, while traces such as for example editpage seem to implement multiple features in sequence. It is an open question, how to best model this graph of relations between and among features and traces. It is by performing feature analysis in the first place that we discover such relationships. Thus, obtaining the best feature definition for an analysis is based on the analysis itself. This clearly suggests an iterative approach to feature definition based on the findings of feature analysis. We plan to investigate this more in the future.

6. Related Work

The basis of our work is directly related to the field of dynamic analysis [1], in particular in the context of reverse engineering[14], visualization of runtime information [2] and trace summarization techniques [10, 9].

Many approaches to dynamic analysis focus on the problem of tackling the large volume of data. Many compression and summarization approaches have been proposed to support the extraction of high level views to support system comprehension [8, 9, 14]. This research is directly related to our work.

In the context of reverse engineering and system comprehension, Zaidman and Demeyer [14] propose an approach to managing trace volume through a heuristical clustering process based on event execution fre-

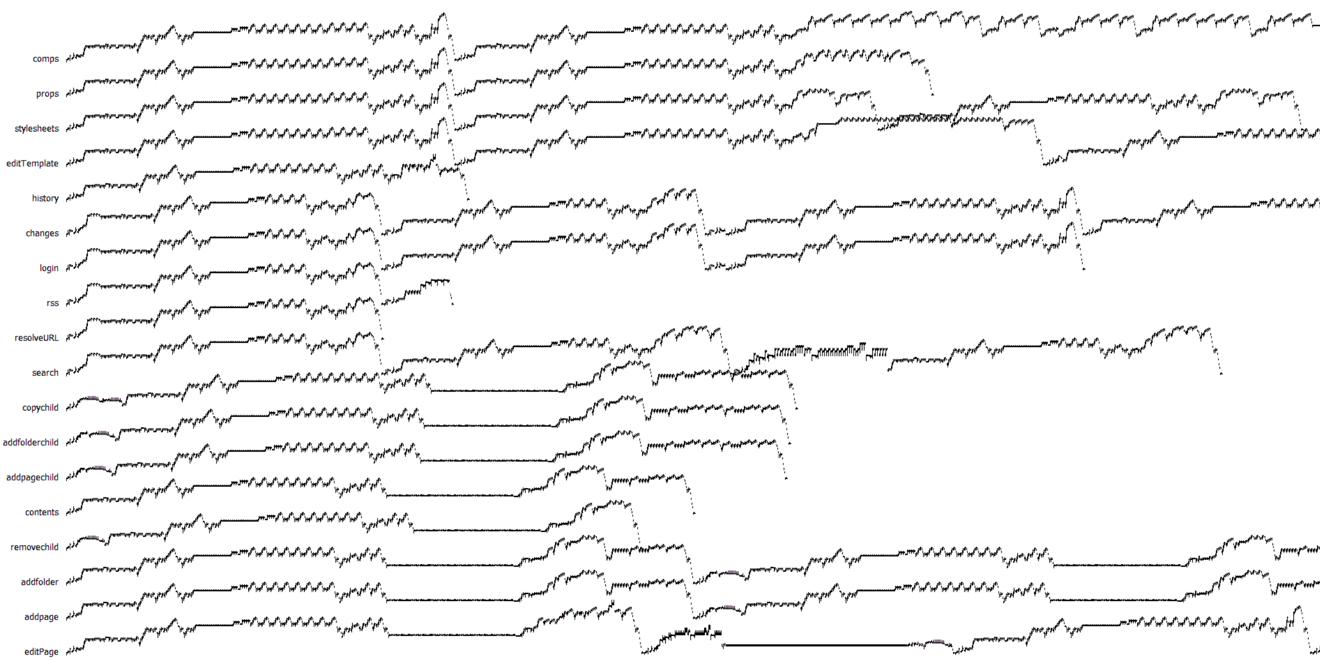


Figure 4. The complete feature space of the SmallWiki case study on one page.

quency. They use a heuristic that divides a trace into recurring event clusters. They argue that these recurring event clusters represent interesting starting points for understanding the dynamic behavior of a system. Their goal is to obtain an architectural insight into a program using dynamic analysis. The context of our work is reverse engineering and system comprehension. We extend this work by exploiting a range of analysis techniques from the domain of signal processing.

The trace summarization techniques such as that proposed by Hamou-Lhadj are directly related to our summarization approach [9]. He describes a trace summarization based on the ideas of text summarization and proposes that the trace summarization take an entire trace as input and return a summary of the main executed events as output. Summarization is based on selection and generalization techniques of text summarization.

A primary contribution of our approach is the ability to represent entire traces on one screen. Other researchers have addressed this. Jerding *et al.* propose an approach to visualizing execution traces as Information Murals [11]. They define a *Execution Mural* as a graphical depiction an entire execution trace of the messages sent during a program's execution. These murals provide a global overview of the behavior, They also define a *Pattern Mural* which visually represents a summary of a trace in terms of recurring execution patterns. Both views are interdependent. Our signal views have the advantage that they reflect the time and sequence of dynamic data.

Pattern detection in dynamic behavior is a research question that has been addressed by many researchers. Hamou-Lhadj and Lethbridge describe an algorithm that extracts patterns in execution traces. They present a set of matching criteria that the use to decide when two patterns are considered equivalent [10]. De Pauw *et al.* apply pattern extraction algorithms to detect recurring exection behavior in traces [3]. Recent work of Nagkpurkar and Krintz [12] describe a technique whereby they characterize the behavior of programs as *phases*. These phases represent repeating patterns in the trace. They decompose a program into fixed-sized intervals of events and combine these according to how similar the intervals are.

7. Conclusions and Future Work

In this paper we drew an analogy between dynamic analysis and signal processing and we described how to transform traces into time series. We visualized traces as time plots, and presented a summarization technique that reduces the length of the trace signal by 50% to 90%, while preserving information relevant to our research goals.

We implemented our signal visualization in DynaMoose, a dynamic analysis tool integrated with the Moose reengineering framework [4]. Using time plot visualization and Monotone Subsequence Summarization we have been able to fit the complete visualization of

18 traces containing over 200'000 events on one single screen. Furthermore, due to the capacity of the human visual system in detecting pattern, this visualization made possible to discern patterns both within and between the traces that would other have been obfuscated by the vast amount of raw data. We plan to further investigate on these promising results, using pattern matching and data mining algorithms.

Analysis of our SmallWiki case study reveals patterns in traces. We recognize that further research in the form of more case studies is required before we can conclude that all phenomena observed in this paper are common to any feature space. However we assume that at least some of the observed patterns are generalizable on most case studies. In our case study, we observed that all traces share a common introduction sequence, which however shows slight variations in some traces. Also we observed that there are large sequences shared by multiple traces, and that there are very few patterns which occur in one sole trace only.

This leads us to our initial question whether our definition of features is flawed, and in fact, a many-to-many relationship between traces and features is much more probable than a simple one-to-one relationship. For example in our case study, traces such as copychild, addfolderchild, addpagechild and removechild seem to implement variations of the same feature, while traces such as for example editpage seem to implement multiple features in sequence. It is an open question, how to best model this graph of relations between and among features and traces. We plan to investigate this more in the future.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[2] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[3] W. De Pauw, S. Krasikov, and J. Morar. Execution patterns for visualizing web services. In *Proceedings ACM International Conference on Software Visualization (SoftVis 2006)*, New York NY, Sept. 2006. ACM Press.

[4] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[5] S. Ducasse, L. Renggli, and R. Wuyts. Smallwiki— a meta-described collaborative content management system. In *International Symposium on Wikis (WikiSym'05)*, pages 75–82, New York, NY, USA, 2005. ACM Computer Society.

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

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

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

[9] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

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

[11] D. Jerding, J. Stasko, and T. Ball. Visualizing message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996.

[12] P. Nagpurkar and C. Krintz. Phase-based visualization and analysis of java programs. In *Elsevier Science of Computer Programming, Special issue on Princples of programming in Java*, volume 59,Number 1-2, pages 131–164, Jan. 2006.

[13] L. Renggli. SmallWiki: Collaborative content management. Informatikprojekt, University of Bern, 2003. http://smallwiki.unibe.ch/smallwiki.

[14] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 329–338, Mar. 2004.

An Environment for Pattern based Dynamic Analysis of Software Systems

Kamran Sartipi and Hossein Safyallah
Dept. Computing and Software, McMaster University
Hamilton, ON, L8S 4K1, Canada
{*sartipi, safyalh*}@*mcmaster.ca*

Abstract

Dynamic analysis of software systems incorporates several challenging issues and also requires supporting tools and techniques to be qualified as a usable and adoptable approach by the research community. This paper presents the issues in designing a trace-based dynamic analysis of software systems and provides a pattern-based dynamic analysis environment to address these issues. The proposed approach identifies the implementation of different software features without any prior knowledge about the implementation of the subject system. The supporting environment employs techniques to tackle the large sizes of the execution traces by using a novel redundant-trace filtering mechanism and the application of data mining techniques. The proposed approach targets specific-features of the software in order to generate frequent traces as execution patterns corresponding to the targeted features. Further on, scattering the patterns over a concept lattice allows us to separate patterns that correspond to common features from patterns corresponding to specific features. The proposed environment is supported by a toolkit called Dynamic Alborz.

KEYWORDS: Dynamic Analysis; Execution Pattern Mining; Concept Lattice; Feature; Trace; Scenario, Alborz.

1. Introduction

Study of the static properties of the software systems has been the focus of research in the reverse engineering community for more than two decades. Techniques such as software clustering, pattern-matching, and structure visualization have been comprehensively studied. The researchers now seek novel approaches to push the state of the practice into more sophisticated techniques that consider dynamic properties of software to open additional views for analyzing and understanding legacy systems [11, 12]. This rather new research avenue is increasingly important as it potentially adds behavioral semantics to the static analysis. On the other hand the dynamic analysis as a standalone technique plays a critical role in program comprehension through techniques for visualizing the behavior of the system [10], clustering the execution traces [20], summarizing the content of large execution traces [7], interaction among GUI components [5], feature to code assignment [4, 6], software structure evaluation [14], and web mining [19].

In this paper, first we elaborate on some important issues that are common in most typical dynamic analysis techniques, and then propose an environment for dynamic analysis that provides solution techniques to address these issues. We also argue that utilizing the discovery nature of techniques such as: data mining to extract the hidden patterns of execution; concept lattice analysis to visualize the structure of the relations among execution traces and their patterns; and string matching algorithms to find redundant traces; would be of significant aid in better analyzing the behavior of a software system.

2 Issues in dynamic analysis

The requirements for research in dynamic aspects of software systems as an activity in software reverse engineering include: devising a proper technique; providing supporting environment; and defining the procedures to achieve and evaluate the results. In this section we elaborate on a number of issues that need to be tackled in such approaches and then provide sample solutions.

Dealing with large execution traces
A major challenge in the trace-based dynamic analysis approaches would occur right at the beginning of the analysis, that is managing very large traces [8, 14, 20]. Execution of typical task scenarios on a medium size software system can produce very large traces ranging to thousands or tens of thousands of function calls which would be an obstacle in proceeding with the analysis. The effective trace of function calls for the intended scenario would be cluttered by a large number of function calls from the operating system, initialization or termination operations, utilities, repetition of sequences caused by the loops, and also noise functions

that are interleaved within the main sequence.

We propose three different techniques to deal with large execution traces. The first technique is based on eliminating the loop-based repetitions in a trace by representing the initial sequence of function entry/exit pairs as *dynamic call tree* that allows us to perform a top-down program-loop elimination operation [14]. In a dynamic call tree a function *f* can appear at different tree nodes with different IDs provided that their sub-trees are different. However, the same functions (as tree-nodes) with identical sub-trees will take on the same IDs. This allows a top-down analysis for loop elimination operation by removing tree-nodes with identical IDs as the children of a tree-node and keep one of them. The second technique is based on the data mining algorithm *sequential pattern discovery* [1] that extracts frequently occurring traces that allows to locate redundant traces such as program-loops or common software operations (e.g., mouse movement, user-interface interactions, and utilities) that occur in different execution traces. Data mining operations by nature generate a large number of patterns, most of which are sub-strings of a larger pattern, hence, a further operation for sub-trace elimination is required to obtained unique traces. The third technique is based on *string manipulation* algorithms [2] that allow to identify repetitive patterns in an string of elements.

In the above techniques, we can either eliminate the patterns to reduce the size, or make a record of the identified patterns as a means for locating close patterns (not exact patterns), where a pattern sequence is interleaved by other function calls. In this way, we can identify approximate matches.

Dynamic analysis to assist static analysis

As mentioned earlier the amalgamation of dynamic analysis techniques with static analysis of software system (that include a rather comprehensive collection of techniques) is considered as a very promising approach [11, 16, 12]. The existing multi-view approaches mostly attempt to extract (or visualize) the static and dynamic views of a software system to allow better understanding of the software properties. These techniques usually produce independent views, not an integration of both views. We propose a technique that allows the dynamic analysis to inject additional information into static analysis based on the external behavior of the system (task scenarios or use cases) in order to adjust the static analysis to produce more sensible results. This approach uses feature to source code assignment to localize the core functions that implement specific software operations (features) [14] and then uses the core functions as the seeds in a software clustering technique [18] to collect functions into software modules or components that correspond to specific operations of the software system [12]. Also, dynamic analysis can be used to assess the structural merits of a software system. This is done by mapping between the group of functions that implement different software features onto the structure of the system (files or modules) in order to assess the impact of a feature on the structure of the system [14].

Discovery of patterns

An important aspect of any analysis task (static or dynamic) is to identify the relevant hidden patterns that assist the engineer to limit the scope of analysis to particular parts of the system as opposed to performing a global search for the desired properties. In this context, the application of techniques such as data mining and concept lattice analysis would reveal patterns of relations among the software entities [3]. Specific data mining techniques such as association rules discovery has direct application in static analysis [13] and sequential pattern discovery has been applied on dynamic analysis for locating the implementation of software features in source code [14]. The application of other data mining techniques on software analysis is yet to be studied. Mathematical concept lattice analysis is an excellent tool for visualizing the structure of relations among software entities and has been well adopted in software reverse engineering for modularization of a software's structure [9, 17, 15]. Very few approaches in dynamic analysis utilize the visualization power of concept lattice techniques. These techniques must handle the inherent characteristic of the lattice in the sense that the lattice easily becomes overwhelmed by the number of generated concepts. One remedy for such a problem is to raise the granularity level of the objects and attributes in defining the context table. We have suggested an approach that separates the common execution patterns from specific patterns of a targeted operation [12].

Usability and extendibility

A common problem for most software analysis approaches is the lack of adequate tool support to warrant durability of the approach and adoptation by other researchers. Most software analysis techniques require an environment including external applications in order to be operational, and they possess multi-step processes. Inadequate tool support would cause interesting approaches to become obsolete or being used only by their developers. Much effort is needed to make an interesting approach usable by others. The characteristics of the supporting environment include: i) short learning curve, usually each step needs to be guided by wizards with clear explanation of their tasks; ii) extendible and interoperable with relevant tools to set a working environment. The interoperability of tools is a key issue in the success of a technique. There are standard and open source technologies available for developing platform independent environments (e.g., Java and XML), as well as tool inte-

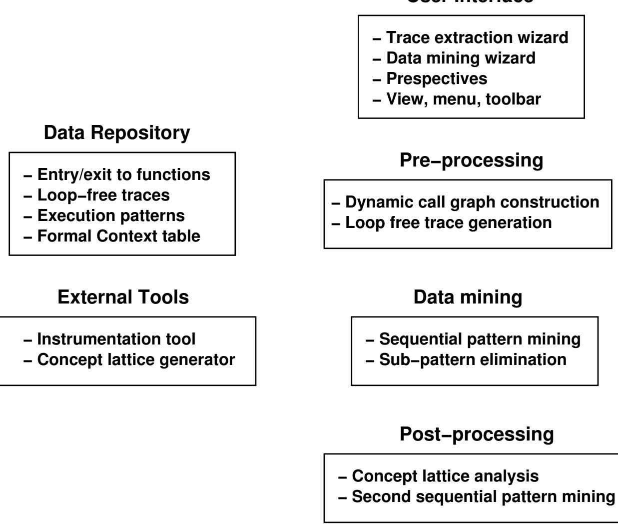


Figure 1. Components and employed techniques for the proposed dynamic analysis environment with the features such as trace filtering and pattern discovery.

gration platforms (e.g., Eclipse) that are based on plug-in technology. Other aspects for success of an approach include: proper and easy to use documentation and manuals and on-line help; as well as open architecture to add more features in future.

In the rest of this paper, first we briefly introduce *Dynamic Alborz* which is a multi-view, interactive, and wizard-based dynamic analysis toolkit that takes advantage of the Eclipse plug-in technology to provide feature extensibility, and uses GXL format to interoperate with other reverse engineering tools. Next, the snapshots of the Dynamic Alborz are presented.

3 Dynamic analysis using Alborz Toolkit

Figure 1 illustrates the set of components and their services that collectively implement the environment for the Dynamic Alborz toolkit. The environment takes advantage of pattern discovery of data mining and concept lattice analysis and is built as an Eclipse plug-in to be used by the research community. The stages of the toolkit operation include: trace extraction; pattern mining; pattern analysis; and structural evaluation. In the rest of this section these stages are briefly described.

Trace extraction: important features of a software system are identified by investigating the system's user manual, on-line help, similar systems in the corresponding application domain, and also user's familiarity with the system. A set of relevant task scenarios are selected that share a single software feature. We call this set of scenarios as *feature-specific scenario set*. For example, in the case of a drawing tool software system, a group of scenarios that share the "move" operation to relocate a figure on the computer screen would constitute such a feature-specific scenario set. In the next step, the software under study is instrumented to generate function names at the entrance and exit of a function execution. By running each feature-specific scenario against the instrumented software system a sequence of function invocations are generated in the form of *entry/exit pairs*. To make the large sizes of the generated traces manageable, in a preprocessing step we transform the extracted entry/exit pairs into a sequence of function invocations and also remove all redundant function calls caused by the cycles of the program loops. The trimmed execution traces are then fed into the execution pattern mining engine in the next stage.

Pattern mining: in this stage, we reveal the common sequences of function invocations that exist within the different executions of a program that correspond to a set of task scenarios. We apply a sequential pattern mining algorithm [1] on the execution traces to discover such hidden execution patterns and store them in the Data Repository for further analysis.

Pattern analysis: each execution pattern is a candidate group of functions that implement a common feature within a scenario set. We employ a strategy to locate functions in execution patterns corresponding to specific features within a group of scenario sets. This is performed by identifying those patterns that are specific to a single software feature within one scenario set (namely *feature-specific patterns*). Similarly, we identify the patterns that are common among all sets of scenarios (namely *omnipresent patterns*). Even for a specific feature, a large group of execution patterns are generated that must be organized (and some must be filtered out) to identify core functions of a feature. We employ two different mechanisms for this purpose: concept lattice analysis and second sequential pattern mining techniques. Concept lattice is an ideal tool for such a task, hence we use the visualization power of concept lattice to generate clusters of functions within feature-specific functions and omnipresent functions. Alternatively, we apply the sequential pattern mining for the second time on the extracted execution patterns of the previous steps to separate feature-specific pat-

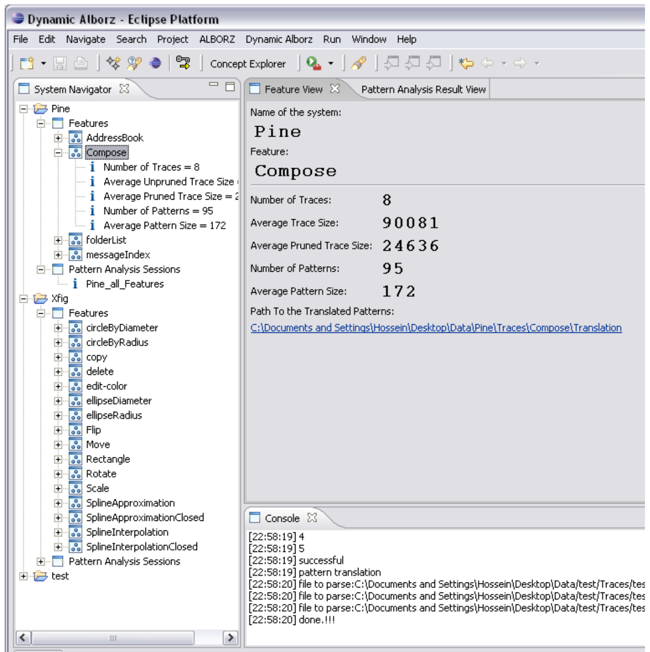


Figure 2. The Dynamic Alborz plug-in is installed in the Eclipse platform; where the stored-data organization of the analyzed systems (left), the statistical data on the analyzed traces (right top), and the execution dialogue (right bottom) are shown.

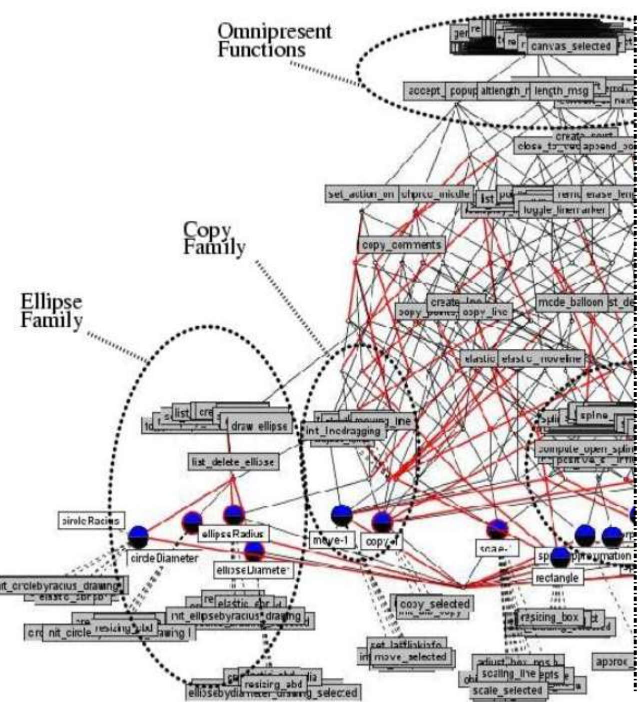


Figure 3. Part of the concept lattice illustrating the groups of features as family of features and corresponding functions.

terns from omnipresent patterns.

Structural evaluation: in a further operation, by associating the group of functions that implement specific feature to the system's structural modules, i.e., files of the system, two metrics for measuring *module cohesion* and *feature functional scattering* are obtained that together provide a means for measuring the impact of individual features on the structure of the software system.

Figure 2 provides a comprehensive overview of the Dynamic Alborz plug-in within the Eclipse platform. On the left side the structures of stored data in the *Data Repository* for two analyzed systems *Pine* email system client and *Xfig* drawing tool are shown. Each *feature* has a set of associated task scenarios that share the feature and generate the traces. For each feature the following statistics are provided: i) number of traces; ii) average trace size with loop-based subtraces (entry/exit pairs); iii) average trace size after pruning the loop-based subtraces; iv) number of the unique execution patterns; and v) average size of the execution patterns. These are typical data that can be obtained for the other features as well. It can be clearly observed that eliminating loop-based subtraces trimmed almost 45% of the function call traces (entry/exit pairs). Also, extracting patterns drastically reduces the size of the traces to be analyzed (i.e., from the range of tens of thousands to the range of hundreds).

Figure 3 illustrates a part of the concept lattice generated from Xfig data, where "Xfig features" represent "lattice objects" and the functions in the patterns corresponding to the features represent "attributes of objects" in the lattice. The use of lattice allows us to separate the patterns that are common to the most of scenarios (top of the lattice) from the patterns that correspond to specific features (bottom of the lattice). Also, the group of features that are close to each other in the lattice and share many attributes represents families of features, as illustrated in Figure 3.

4. Conclusions

In this paper, we discussed important problems with regard to the trace-based dynamic analysis of software systems and proposed possible solutions. These problems include: handling very large sizes of the generated execution

traces that are overwhelmed by redundant loop-based subtraces; extracting execution trace patterns that allow us to restrict the scope of analysis to traces that occur frequently either naturally or by user's focus on specific features; incorporate dynamic analysis with static analysis as a general theme to enhance the recovery power of the static techniques; and finally paying more attention to the usability and extendibility of the analysis environment. We also proposed solutions such as: top-down loop-trace elimination using dynamic call trees; application of sequential pattern mining and concept lattice analysis to extract patterns; injecting execution-based semantic information into structure recovery processes in order to achieve more sensible results. Finally, we provided an environment and a toolkit (Dynamic Alborz) for identifying the implementation of both specific and common operations of a software system. As the future extension, we intend to explore new techniques for amalgamation of dynamic and static views of software systems.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[2] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Process Letters*, 12(5):244–250, 1981.

[3] C. M. de Oca and D. L. Carver. A visual representation model for software subsystem decomposition. In *Proceedings of the Working Conference on Reverse Engineering*, pages 231–240, 1998.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Feature-driven program understanding using concept analysis of execution traces. In *Proceedings of the IWPC'01*, pages 300–309, 2001.

[5] M. El-Ramly, E. Stroulia, and P. Sorenson. Mining system-user interaction traces for use case models. In *Proceedings of IWPC'02*, pages 21–29, 2002.

[6] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR'05*, pages 314–323, 2005.

[7] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC'06*, pages 181–190, June 2006.

[8] A. Hamou-Lhadj, T. Lethbridge, and F. Lianjiang. Challenges and requirements for an effective trace exploration tool. In *IWPC'04*, pages 70–78, 2005.

[9] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359, 1997.

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

[11] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proceedings of the IEEE CSMR*, pages 47–55, 2002.

[12] K. Sartipi, N. Dezhkam, and H. Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *Proceedings of WCRE'06*, page 10 pages, October 2006.

[13] K. Sartipi and K. Kontogiannis. A user-assisted approach to component clustering. *Journal of Software Maintenance: Research and Practice (JSM)*, 15(4):265–295, July/August 2003.

[14] K. Sartipi and H. Safyallah. Application of execution pattern mining and concept lattice analysis on software structure evaluation. In *Proceedings of the SEKE'06*, pages 302–308, June 2006.

[15] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, Nov./Dec. 1999.

[16] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. View-driven software architecture reconstruction. In *Proceedings of the IEEE Working Conference on Software Architecture (WICSA'04)*, pages 122–132, 2004.

[17] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the ICSE 1999*, pages 246–255, 1999.

[18] T. A. Wiggerts. Using clustering algorithms in legacy systems modularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43, 1997.

[19] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR'05*, pages 134–142, 2005.

[20] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR'04*, pages 329–338, 2004.

Aiding in the Comprehension of Testsuites

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

Arie van Deursen
Delft University of Technology and CWI
The Netherlands
Arie.van.Deursen@cwi.nl

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

Abstract

An integral part of test-driven software development is utilizing testcases to ensure the software's quality. However, as testsuites grow larger, they tend to grow beyond control and are no longer easily comprehended. In this position paper, we propose to employ dynamic analysis and abstractions in reconstructing scenario diagrams from such testsuites. We discuss several challenges and suggest solutions to tackle these issues.

1. Introduction

When implementing and maintaining software systems, *testing* is of vital importance to help increase the quality and correctness of code. Test-driven development [1] implies creating and maintaining an extensive testsuite in order to guarantee that the various components work correctly, both individually (by means of unit tests) and as a whole (through use of testcases).

A testing framework for Java software that is commonly used is *JUnit* [2]. JUnit allows for the specification of both unit tests and full testcases and is easy to use. A JUnit testcase consists of several steps: the creation of a fixture, exercising the method under test, comparing the results, and the teardown. It can be run as part of a complete testsuite.

Our goal is to help developers in the course of understanding and maintaining testsuites, and perhaps even discover errors or mistakes [3]. To make such testcases easy to understand, one must come up with a visualization that is both detailed and human readable. This involves *analyzing* or *tracing* the testcases, applying certain *abstractions* and, finally, *presenting* the results.

UML sequence diagrams [4] are a potentially useful means to visualize a system's behavior. A *scenario diagram* is a somewhat simplified version of a sequence diagram that is derived from a specific scenario, i.e., containing one particular control flow. Scenario diagrams provide detailed information on interactions at either the class level or the object level, and are very readable because the chronological order is intuitive. However, if no abstractions are applied, scenario diagrams tend to become too large: the complete execution of a sizeable software system would result in a scenario diagram that contains more information than the reader can handle.

In this position paper, we propose to use reconstructed scenario diagrams for the purpose of making JUnit testsuites easier to comprehend. We obtain these diagrams by tracing the execution of a testsuite.

The next section outlines the issues and design choices that we will encounter. Section 3 discusses several solutions that we are proposing, and Section 4 describes related work. Finally, we draw conclusions and indicate future directions in Section 5.

2. Challenges

In the course of converting testsuites to scenario diagrams, we face several challenges. This section addresses the most prominent issues and design choices.

Dynamic vs. static In obtaining scenario diagrams from testcases, we can choose whether to capture the system's behavior by *static* analysis (i.e., analyzing the code) or through *dynamic* analysis (i.e., tracing the execution). The benefits of a static approach are the genericity and compactness, whereas a dynamic technique offers more detailed information on important aspects such as *late binding*. This is illustrated in Figure 1: this example of *dynamic dispatch* would not have been very readable in a static context because of the lack of object identities therein. A well-known drawback of scenario diagram reconstruction using dynamic analysis is that one needs specific scenarios and that the diagrams thus represent only part of a whole system's behavior; however, since testcases *are* basically scenarios we feel that, in this particular context, more accurate information outweighs genericity.

Test stage separation The second issue that arises is how to trace the *various phases* of the execution of a testcase,

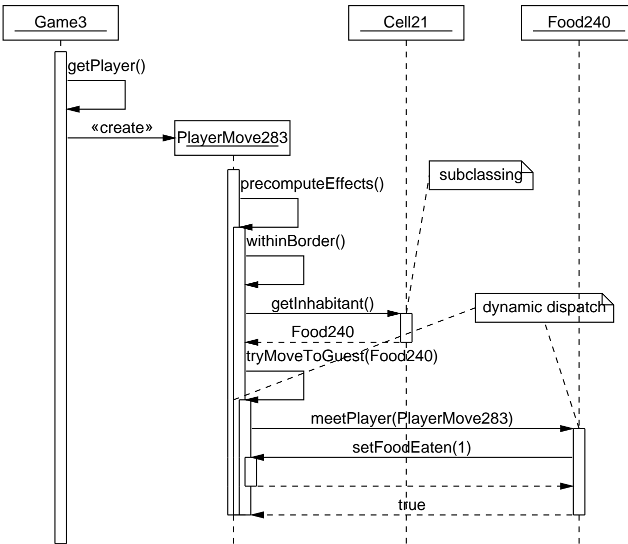


Figure 1. Dynamically reconstructed scenario diagram featuring an example of dynamic dispatch.

i.e., how we can distinguish between the test initialization, the test execution and the result validation. Traditional tools and techniques generally do not deal with this testcase-specific problem. It is an important issue, as the initialization and validation phases are potentially less relevant and should therefore be treated separately (i.e., be put in a separate diagram or even skipped).

Class vs. object level Another design choice concerns whether we want to trace the interactions on the *class* level or on the *object* level. The former is easier, whereas the latter provides more detailed information that is especially suitable for display in scenario diagrams.

Scalability However, despite the fact that in general the execution of a unit testcase is relatively short, *scalability* problems are inevitable. Most simple unit tests will probably fit within a single scenario diagram, whereas more complex testcases induce too many interactions to simply put in a diagram without applying any form of abstraction. Therefore, we will need abstractions that are both efficient and *useful*, i.e., we must determine which interactions are presumably irrelevant and can consequently be omitted. One way would be for the tool to suggest several abstractions, while ensuring the viewer remains in control of the level of detail.

3. Techniques

We have come up with several design choices and solutions to the issues discussed earlier. We are in the process of implementing these in a prototype tool that we are using for analyzing a range of existing test suites.

3.1. Tracing testcases

There exist several methods to obtain traces from software systems, among which the most commonly used are manually instrumenting code (e.g., [5]), using a debugger or profiler, and instrumentation through *aspects* [6]. The shortcomings of each of these techniques are mostly well known and are not discussed here. We feel that using aspects in our framework is the most flexible solution in our context, since it enables us to specify very accurately which parts of the execution are to be considered, where tracing must start and stop, and which steps need be taken afterwards.

Aspects can trace the execution of a testcase and produce detailed information on all interactions, such as the unique objects that are involved, the current thread, and the (runtime) arguments in case of method and constructor calls. Being able to distinguish between objects has certain advantages, as it provides detailed information on object interactions and exposes occurrences of polymorphism and late binding. Figure 1 shows an example[1].

In addition, aspects allow for the precise definition of which objects and interactions are to be traced, and enable us to make a distinction between the various stages in a testcase. Among other things, this distinction offers us the opportunity to filter the assertions in the comparison stages, in case the viewer considers them unnecessary. Moreover, we have a means to create *separate* scenario diagrams of the various phases for the viewer to browse through.

3.2. Abstractions

In order to make large scenario diagrams easier to read, we need several types of abstractions to reduce the amount of information. In the context of scenario diagrams, one intuitively thinks of omitting objects and classes and hiding interactions to shrink the dimensions of the diagram. But which messages and objects can be omitted while maintaining the general idea of the testcase?

One technique that we propose is to limit the *stack depth* of the execution. By use of a maximum stack depth, we can hide all interactions above a certain threshold, thus omitting messages and (potentially) the objects involved. Intuitively, this filters low level messages that tend to be too detailed, at least for an initial viewing. This is illustrated by Figures 2 and 3: the former diagram depicts the testcase in full detail, whereas the latter shows only the essence. A similar

[1]The scenario diagrams in this research were created using UMLGraph [7].

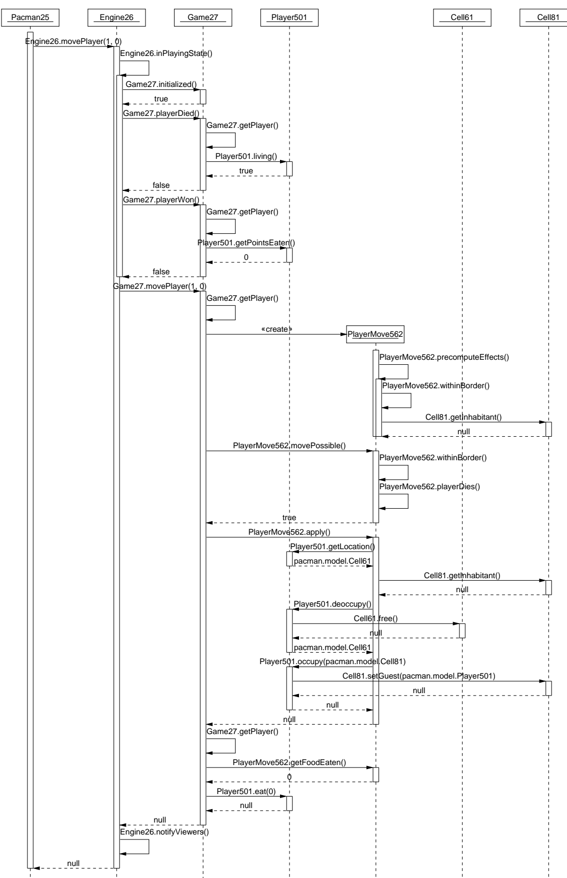


Figure 2. Reconstructed scenario diagram at a low abstraction level.

abstraction was applied in [8], in which (in a static context) the length of a call chain is considered.

Another abstraction method is to *hide constructors* if there are too many. This is especially applicable in the initialization stages of complex testcases. Additionally, it would be even more useful to hide *irrelevant* constructors and the associated objects, i.e., to filter them in case they are never used later on. That way, we will presumably have reduced the dimensions without loss of important information.

As was briefly mentioned in Section 2, we feel that the viewer must ultimately decide which abstractions are to be applied and which are not. We therefore plan to conclude the processing of all testcases with *recommended* scenario diagram specifications: based on several testcase statistics such as the amount of objects, constructors, and stack depth frequencies, it is automatically determined which of the abstractions are highly advised. Such specifications must be subject to adjustments at the viewer's discretion.

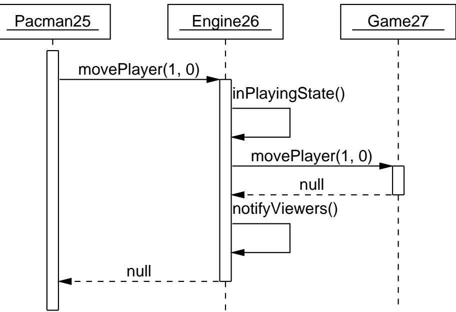


Figure 3. Reconstructed scenario diagram at a high abstraction level (with a maximum stack depth of 2).

4. Related work

There is a great deal of ongoing research regarding the visualization of dynamic information. Among this research is a comparison of dynamic visualization techniques by Pacione et al. [9] and of trace summarization tools in particular by Hamou-Lhadj et al. [10].

On the subject of testing, an interesting research was conducted by Gälli et al. [11], in which the authors focus on ordering (unit) tests based on the number of methods that are called by these tests.

With respect to reverse engineering UML scenario diagrams, several attempts have been made in the past. They include both static and dynamic approaches and, as was discussed in section 2, each of these techniques has its advantages and drawbacks.

Various approaches reconstruct scenario and interaction diagrams based on static analysis of program code [8, 12, 13, 14]. The techniques that are used vary from mapping of control flow graphs [13] to interprocedural dataflow analysis [8]. A comparison of various approaches is presented in [14].

There have been various reports of dynamic approaches in the literature as well, of which some are discussed by Briand et al. [15]. In the same paper they present a strategy that is aimed at capturing the objects, the messages that are exchanged, conditions, and repetitions when executing a scenario. This is accomplished through manual intrumentation of the source code. The paper does not describe how the scenarios are obtained, i.e., how they are distilled from a use case or filtered from the reconstructed scenario diagram.

Systä et al. [16] aid in the understanding of Java systems in an environment called Shimba, which uses both static analysis and dynamic information. They reason at the level of classes and obtain the required information from a system's bytecode and by using a customized debugger. Their focus is primarily on maintaining consistent views on both

structural (static) and behavioral (dynamic) aspects of the system.

Riva et al. [17] combine static and dynamic analysis to reconstruct message scenario charts. In their trace-based approach, they provide an abstraction mechanism based on the decomposition hierarchy that is extracted from the system's source code. It is not described how the scenarios are defined, and in dealing with large diagrams, they only offer manual abstraction techniques.

5. Conclusions

We have proposed to employ dynamic analysis and scenario diagrams with the goal of aiding in the comprehension of testsuites. We have discussed the issues and design choices that we will encounter and, through several examples, elaborated on our choices for these techniques. Dynamic analysis is the most logical choice for us as it provides the most details; and scenario diagrams, as long as they are not too large, are an excellent means of showing this detailed behavior. By means of certain metrics, a set of recommended abstractions is determined that aims towards presenting a scenario diagram (for each testcase stage) that is both readable and contains the desired amount of detail.

Our next step is to implement these techniques in a framework and to examine whether we can obtain meaningful results for both interested viewers and experienced developers. To this end, we are planning to perform at least two case studies.

JPACMAN is a simple game consisting of 25 Java classes and is mainly used for educational purposes. Though being a small system, it is complicated enough to give us an indication as to the the usefulness of our approach, since JPACMAN involves polymorphic methods and large traces. It also has a testsuite comprising over 50 testcases.

Another case that we are currently investigating is *CroMod*, an industrial Java system featuring both simple unit tests and complex testcases. By means of extensive feedback from the developers, we want to discover which abstractions are generally required and hope to improve our techniques.

References

[1] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[3] J.A. Jones and M.J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 273–282, 2005.

[4] OMG. UML 2.0 infrastructure specification. Object Management Group, http://www.omg.org/, 2003.

[5] InsectJ: A generic instrumentation framework for collecting dynamic information within Eclipse, http://insectj.sourceforge.net/.

[6] AspectJ: The AspectJ project at Eclipse.org, http://www.eclipse.org/aspectj/.

[7] D. Spinellis. On the declarative specification of models. *IEEE Software*, 20(2):94–96, march/april 2003.

[8] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineerd sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 254–263, 2005.

[9] M.J. Pacione, M. Roper, and M. Wood. Comparative evaluation of dynamic visualisation tools. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 80–89, 2003.

[10] A. Hamou-Lhadj and T.C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON'04)*, pages 42–55, 2004.

[11] M. Gälli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pages 114–123, 2004.

[12] R. Kollmann and M. Gogolla. Capturing dynamic program behaviour with UML collaboration diagrams. In *Proceedings of the 5th Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 58–67, 2001.

[13] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 96–102, 2005.

[14] R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 22–32, 2002.

[15] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 57–66, 2003.

[16] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software - Practice and Experience*, 31(4):371–394, 2001.

[17] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proc. 6th Conf. on Software Maintenance and Reengineering (CSMR'02)*, pages 47–55, 2002.

A Lightweight Approach to Determining the Adequacy of Tests as Documentation

Joris Van Geet* and Andy Zaidman**

*University of Antwerp, Belgium
Joris.VanGeet@ua.ac.be
**Delft University of Technology, The Netherlands
& University of Antwerp, Belgium
Andy.Zaidman@ua.ac.be

Abstract

Programming process paradigms such as the Agile process and eXtreme Programming (XP) tend to minimise ceremony, favouring working code over documentation. They do, however, advocate the use of tests as a form of "living documentation". This research tries to make an initial assessment of whether these unit tests can indeed serve as a form of full-fledged documentation. The lightweight approach we propose is mainly based on the number of units that is covered by each unit test. This paper discusses the approach, the corresponding tool and the results of a first case study.

1 Introduction

The program comprehension process a user goes through when studying a piece of software can benefit greatly from having up to date documentation available. However, often the documentation of a software project is either out-dated or non-existent. Programming practices such as the Agile process or the XP process even have a tendency to minimise documentation, as these processes value working code over comprehensive documentation [3].

Both Agile programming and XP emphasise testing and even advocate the use of a test-driven approach when writing a new piece of software [2]. Because the tests are written first, they completely define what the code should do. As such, the tests can be considered as a form of "living" documentation that can be consulted when one wants to learn what the code is supposed to do [4].

Our research aim then is to make an initial assessment of the quality of the unit tests with regard to their adequacy as documentation. To determine their adequacy, we will look at two criteria of tests, namely:

1. The test coverage of the system, i.e. how much of the system is actually tested.
2. Whether each test is focused on a single unit of the system or whether each test covers a number of units.

This second criterion forms the basis for our hypothesis: *unit tests are possibly not adequate enough for documentation purposes when they cover a number of units*. This basic idea stems from the fact that when a unit test covers multiple units of production code, the unit test will be harder to understand because of an increase in coupling and complexity. A similar observation has been made by Selby and Basili when it comes to understanding "regular" production code [5]. This is one of the reasons for the pursuit of low levels of coupling.

To determine these "test dependencies" we rely on dynamic analysis, which, in the presence of polymorphism, allows us to circumvent expensive slicing operations.

As a case study to determine the test coverage and extract the test dependencies, we used Apache Ant[1], a widely used Java build tool. We determined its test coverage for a number of versions and extracted the test dependencies from the latest available version.

[1] Form more information, see: http://ant.apache.org

2 Unit Tests as Documentation

Testing comes in many forms and can be classified in various ways. The "Guide to the Software Engineering Body of Knowledge" (SWEBOK) [1] provides some interesting classifications. One of them is based on the *granularity of testing*:

- **Component/Unit testing** is concerned with verifying functionality of small and (clearly) separable components.
- **Integration testing** aims at verifying the interaction between components. Usually these components have already been tested by the previous strategy.
- **System testing** tests the system as a whole. This strategy is considered useful for testing non-functional requirements, as the functional requirements should have been tested by the previous two strategies.

It should be noted, however, that the boundary between component testing and integration testing is blurred for object oriented systems as objects are used at all stages of the software process [6]. This observation by Sommerville is interesting because it conflicts with the criteria for the adequacy of tests as documentation, which we set out in Section 1.

It is our opinion that to have optimal documentation, i.e. to be able to understand each unit present in the system, each unit should be documented. As a consequence we expect each unit to be tested, which we can evaluate by determining the test coverage, but we also expect each unit to be tested *in isolation*, to have a clear and unrestricted view of how the unit works. Furthermore, we acknowledge the fact that when units are not tested in isolation, their complexity tends to increase, which can also hinder understandability. We are aware of the fact that certain units cannot be tested in complete isolation, but the usage of stubs can be beneficial to the understandability because they are often less complex than their actual implementations.

3 Tool

When trying to determine whether each test command tests only a single unit of production code (criterion 2 from Section 1), we need to extract test dependencies. A test dependency being *the relation between a unit of code and its invoking test command*. Since we focus on the JUnit testing framework, we define a unit of code as a (production) method and a test command as a unit test method.

For extracting the test dependencies we created a tool with a pipe and filter architecture. The tool starts by tracing the execution of the test scenario(s), followed by an analysis of that trace data to eventually result in two xml files that both contain the same test dependency information, albeit in a different form. The first file contains for each unit of production code the test commands which invoke the unit, while the second XML file contains the inverse relations, namely for each test command, the units of code that are invoked by it.

To trace the different execution scenarios, we used a *profiler agent* implemented with the Java Virtual Machine Profiler Interface (JVMPI) [7]. This agent provides a two way communication path with the virtual machine. We are interested in various events that the virtual machine emits during execution, especially the `method_entry` and `method_exit` events. Our agent specifically listens for these two events as they provide the crucial information for a dynamic call graph. Whenever such an entry or exit occurs, some identification information is written to the trace file containing the fully qualified name of the method, its formal parameters and its return type[2].

Because the virtual machine sends out these events for all methods, including the ones from system classes and third party libraries, we performed a basic form of filtering to only trace packages or classes that are of interest. Note that at this stage we merely store the trace data for further analysis (offline analysis), instead of analysing the trace data on the fly (online analysis).

The trace file from the profiler agent provides us with the necessary raw data to extract test dependencies as it lists the entry and exit of all calls in chronological order. The dependency extractor takes a regular expression to identify the test packages or classes. Methods of such a test class are identified as a test method if they take no arguments and their name starts with the string '`test`', as this is the convention in the JUnit testing framework.

Once we have identified the test methods we can easily deduce all methods that are *tested* by a certain test method, as they appear between entry and exit of that test method. To obtain all the test methods that test a particular method, we inverse this relationship. Finally, we store this information in a proprietary XML format, thereby making the test dependencies explicit in both directions. Furthermore, method calls that appear more than once within the same test method are only listed once, as this tool provides a *flattened call graph* resulting in a *set* of methods for each test method and vice versa.

4 Results

As we mentioned before, we used Ant, the well-known build tool, as an initial case study for our experiment. We chose Ant because of its relative simplicity and also because it is widely used, both in the open source community and the

[2]The return type of a method is not necessary to uniquely identify a method. However, the Java Virtual Machine provides this data together with the parameters, we keep it for human readability.

	method coverage	
ant version	percentage	bare count
1.6.3	61%	3247/5351
1.6.4	63%	3399/5363
1.6.5	65%	3739/5745

Table 1. Method coverage as generated by Emma.

ant version	methods	tests	calls
1.6.3	4467	1330	286499
1.6.4	4472	1337	288363
1.6.5	4767	1407	324250

Table 2. Total count of methods, tests methods and method calls.

version	mean	σ
1.6.3	68.02	190.35
1.6.4	64.48	183.49
1.6.5	64.14	182.40

Table 3. Average number of test methods for an arbitrary method.

version	mean	σ
1.6.3	230.45	146.10
1.6.4	215.68	136.68
1.6.5	215.41	137.01

Table 4. Average number of methods that an arbitrary test method runs through.

closed source community, as evidenced by the integration of Ant in many commercial IDEs.

The results of our experiment can be divided into four parts.

1. The first part determines the test coverage, for which we used already available tools.
2. The second part deals with numerical data that we retrieved from the Ant distribution.
3. The third part presents anecdotal evidence that we retrieved when studying code fragments for evidence of our findings from the numerical data.
4. The fourth part presents an historical perspective, capturing the evolution of the testing strategy.

4.1 Test Coverage

Table 1 gives an overview of the test coverage, more specifically the methods that are covered by the tests. As can be seen, the coverage varies from 61% to 65% percent, depending on the version of the Ant project.

Potentially, this also means that only about 2/3 of the methods are documented, although this standpoint could be considered a little harsh, as, just as with regular documentation, not every part of a system needs to be thoroughly documented.

It is our opinion that a coverage level of about 65% should be sufficient for documentation purposes, although we acknowledge that a higher level of test coverage can – logically – only improve understandability.

4.2 Numerical data

Initially, we calculated the *number of unique methods* tested, the *number of test methods* executed and the *total number of method calls* present in our flattened call graph to get a quick feel of the application's test infrastructure. The results of this operation are listed in Table 2.

Based on this information we performed two calculations, namely:

- the average number of test methods that test an arbitrary method (Table 3)
- the average number of methods an arbitrary test method runs through (Table 4)

We can see that, on average, a method is tested by approximately 64 test methods and a test method tests approximately 215 methods. These numbers are shocking in contrast with the *ideal* one to one relation between method and test method. However, the enormous standard deviation[3] of the averages we calculated, suggests that the actual values are highly variable, indicating that further investigation is needed.

To get a better view on the distribution of tested methods and test methods, we represented them in a box plot[4] which uses more robust measurements such as the *median* and other *quartiles* instead of the unstable mean.

Figure 1 illustrates the distribution of the number of methods that are tested by a test method. For version 1.6.5, for example, you can see that half of the test methods test more (and the other half tests less) than 212 methods, since 212 is the median (= the second quartile `Q2`). For the same version you can see that half of the test methods test no more than 331 (the third quartile) and no less than 149 (the first quartile) methods. The distribution is almost symmetric around the median, leaving us with similar results as the

[3] According to [9] standard deviation is the most common measure of *statistical dispersion*. Simply put, standard deviation measures how *spread out* the values in a data set are. Traditionally this measure is represented as σ.

[4] Please note that we use a stripped version of the traditional box plot [8]. Whereas a standard box plot has a parametrised *acceptable range* to define what an outlier is (usually 3/2 times the *inter quartile range*), our range simply extends to the minimum and the maximum values, thus not explicitly specifying outliers.

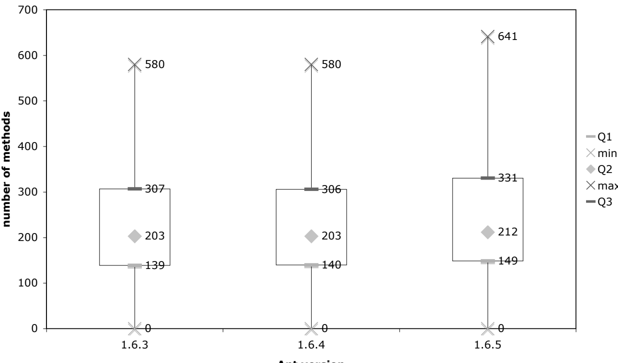


Figure 1. Box plot of the distribution of the number of methods tested by an arbitrary test method.

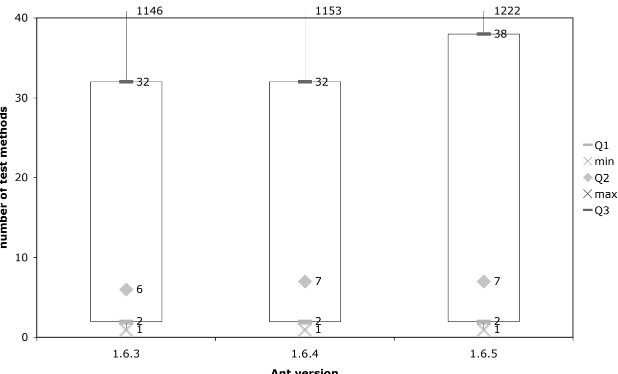


Figure 2. Box plot of the distribution of the number of test methods that test an arbitrary method.

```java
public class RenameTest extends BuildFileTest {
  public void setUp() {
    configureProject(
      "src/etc/testcases/taskdefs/rename.xml"); }
  public void test1() {
    expectBuildException("test1",
        "required argument missing"); }
  public void test2() {
    expectBuildException("test2",
        "required argument missing"); }
  public void test3() {
    expectBuildException("test3",
         "required argument missing"); }
  public void test4() {
    expectBuildException("test4",
        "source and destination the same"); }
  public void test5() {
    executeTarget("test5"); }
  public void test6() {
    executeTarget("test6"); }
}
```

JUnit Test Case

```xml
<project name="xxx-test" basedir="." default="test1">
   <target name="test1">
      <rename/>
   </target>
   <target name="test2">
      <rename src=""/>
   </target>
   <target name="test3">
      <rename dest=""/>
   </target>
   <target name="test4">
      <rename src="testdir" dest="testdir"/>
   </target>
   <target name="test5">
      <rename src="template.xml" dest="."/>
   </target>
   <target name="test6">
      <rename src="template.xml" dest="template.tmp"/>
      <rename src="template.tmp" dest="template.xml"/>
   </target>
</project>
```

Test Build File 'rename.xml'

Figure 3. Test structure for the `rename` Task

ones we obtained from the averages.

However, for the number of test methods per method, we do get a different perspective on the distribution. As you can see in figure 2 the box plot is stretched towards the top, meaning that there are only few outliers. For version 1.6.5, for example, half of the methods are tested by no more than seven test methods. A quarter of the methods are even tested by no more than two test methods. As opposed to the average of 64, we can see that 75% of all methods are tested by no more than 38 test methods.

4.3 Anecdotal Evidence

Let us dig a little deeper, based on these statistical observations. In Table 2 we can see that Ant 1.6.5 runs approximately 1400 test methods and the box plot in Figure 2 shows us at least one method that is tested by *more than 1200* of these test methods. Based on the rather high average (Table 4) of methods that an arbitrary test method runs through, we suspect even more of these methods that are tested by almost all test methods. Two possible explanations come to mind:

1. Some form of *generic setup code* is executed at every run. This code would have to be located in the test methods themselves[5], since the dependencies of the `setUp()` and `tearDown()` methods are not extracted from the original trace.
2. Some form of *generic test code* provides an execution scenario that is similar for all tests. This would indicate an integration testing strategy or at least a lack of stub usage.

[5] Setting up test data in the test method is not uncommon as it is the only way to initialise different test data for test methods in the same class.

	bare count			percentage	
ant version	yes	no	total	yes	no
v1	153	53	206	74.27%	25.73%
v2	259	81	340	76.18%	23.82%
v3	328	107	435	75.40%	24.60%
v4	414	150	564	73.40%	26.60%

Table 5. Second Experiment: Test methods based on `BuildFileTest`**.**

Further investigation of the source code revealed the latter option to be true. We queried our dependencies for classes containing those *often called* methods and briefly navigated through the source code with a code browser. Figure 3 nicely illustrates our findings. The top of Figure 3 is a source code extract of the unit test of the Ant `rename` task. As you can see in line 1, this test extends `BuildFileTest`, an abstraction of a unit test that uses a build file as test data. Line 3 shows that, for each test run, a project is configured based on `rename.xml` (bottom of Figure 3), a build file specifically designed for testing the `rename` task. As you can see, each test method has its own *target* in the build file. Executing a test method is nothing more than calling its corresponding target on the newly created project and checking whether or not the task produces the correct exception. When searching for that specific build file, we found similar build files for almost all other tests.

4.4 Historical Perspective

We performed similar experiments for four other phases of Ant's evolution, to verify whether production code and tests evolve (more or less) simultaneously. Our observations here are that the test base, and with it the amount of unique methods that are tested, grows consistently with each version. This suggests that newly created test methods test priorly untested functionality. Also, we see that the number of tested methods increases more rapidly than the number of test methods. Furthermore, on average, a test method runs through *more* methods with each subsequent version. This indicates that the integration testing strategy is gaining popularity as the development of Ant evolves.

To investigate this further, we queried our dependencies for the `BuildFileTest` class, as it is the basis of the testing framework in version 1.6.x. The dependencies revealed the presence of this class in all versions except for `v1`. Closer investigation showed that in `v1` similar functionality was available in the `TaskdefsTest` class. As the name indicates, this was only used to test Ant's `taskdef` constructs. In the transition to `v2` this class was renamed to `BuildFileTest` to be used by all Ant constructs. To investigate the evolution of this testing strategy we queried the dependencies for all test methods that call at least one of these framework methods and for all test methods that call none of them. The fourth column of table 5 shows the percentages of test methods that rely on the `BuildFileTest` (or the `TaskdefsTest` for `v1`). A remarkable result at first sight, as we might have expected this percentage to grow in subsequent versions. However, this merely indicates that the integration testing framework was already in place in `v1` (in the form of the `TaskdefsTest`) and that the increased usage of that framework has been consistent over the different versions: for every four new test methods, three were based on the `BuildFileTest`.

4.5 Discussion

From the statistical data, the anecdotal evidence and the evolutionary trends that we have discussed in the previous sections, we can conclude that the development team of Ant does not follow a strict unit testing strategy, but rather, follows a strategy that can be classified as an integration testing strategy.

As we have mentioned previously, this kind of testing process can lead to tests that are less suitable for documentation purposes. The main indicator for this reduced adequacy is the fact that a single unit of code cannot be easily understood without also understanding other modules.

This tight coupling might have its consequences when trying to understand a single unit or a small set of units within the system separately. Furthermore, it has been shown that tightly coupled systems are more difficult to understand [5], and there is no reason to assume that this is any different for tests.

5 Conclusion

In this paper we have presented a lightweight approach to determine the adequacy of (unit) tests as a form of documentation. Such a form of documentation is actually advocated by the Agile process and eXtreme Programming (XP). To determine the adequacy, we set out two criteria, namely (1) the level of test coverage and (2) whether the tests work in isolation, i.e. how many units of production code are involved in one test command.

With regard to the test coverage we witnessed a method test coverage of around 65%, which is a quite good level, but can be improved for documentation purposes.

With regard to the isolation factor, we witnessed an integration testing strategy in our Ant case study. This integration testing strategy stands opposed to the isolation criterion that we set out and as such, we have to express our concerns with regard to the understandability of these pieces of (test) code, as involving multiple units of code within one test

command increases coupling and complexity, two closely related factors that can influence understandability.

References

[1] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. Accessed on July 17th 2006, http://agilemanifesto.org/.

[4] E. Hieatt and R. Mee. Going faster: testing the web application. *IEEE Software*, 19(2):60–65, 2002.

[5] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, 2 1991.

[6] I. Sommerville. *Software engineering (6th ed.)*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[7] Sun. The Java Virtual Machine Profiler Interface documentation, 2004. Retrieved May 7th 2006.

[8] J. W. Tukey. *Exploratory data analysis*. MA: Addison-Wesley, 1977.

[9] Wikipedia. Standard deviation. Retrieved May 14th 2006.

Combining Reverse Engineering Techniques for Product Lines

Dharmalingam Ganesan, Isabel John, Jens Knodel

Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany

{dharmalingam.ganesan, isabel.john, jens.knodel}@iese.fraunhofer.de

ABSTRACT

In a product line context, the migration strategy to exploit functionality embodied in existing components having a high reuse potential can be fourfold: reuse as is, reuse and adapt, recover and reconstruct, or (re-)implement. This position paper focuses on the recover and reconstruct strategy and presents an integrated architecture reconstruction approach that aims at migrating core functionality, major variants and key features of existing systems into a product line infrastructure. We combine dynamic and static reverse engineering techniques within a systematic reconstruction approach, whereby one of the main goals is to be compliant to the product line architecture. In this paper, we present the reconstruction approach and illustrate the technique interactions techniques in a case study.

Keywords
software architecture, product line engineering, asset recovery, dynamic analysis.

1. INTRODUCTION
Product line engineering is a development paradigm that stands for pro-active, strategic, and successful reuse [1]. Product line engineering aims at sharing more than just the development effort, the goal is it, to improve the quality, reduce time-to-market, and increase the number of derived products. Typically, product lines are built on top of existing, related software systems whereby the common artifacts are migrated in a product line asset base. In order to meet the quality requirements of the asset base high, the product line architects have to decide whether or not an existing asset becomes part of the asset base, in particular to identify the needs for adaptation to product line needs and to reason about its suitability for the product line [4] [7]. This paper shows how architects can be supported in this task by a combination of different reverse engineering techniques taking into account different assets like code, documentation or the running system.

We propose an incremental migration towards product line engineering that employs an integrated architecture reconstruction approach to identify components with a high product line potential and leading to a prioritization in the migration plan. The approach applies a combination of different reverse engineering techniques and we show how the techniques can benefit from each other.

Related work concerns in general architecture recovery and feature location techniques used to extract and determine assets for building an asset base. Regarding architecture recovery a number of tools have been developed that can be used to extract higher-level views on the implementation of software systems. Tools are, for example Bookshelf [3] or Rigi [10]. They follow the Extract-Abstract-View Metaphor but they do not deal with runtime traces.

Discovering the architecture from run time traces was proposed in [11]. They use state machine as a mapping language to raise the level of abstraction of the collected traces to architecture level. An interesting approach for visualizing message sequence charts is introduced in [2]. Various colors and linked views are used to display the interaction program entities. Regarding feature location a number of approaches exist [8], [9]. Parts of these techniques can be integrated into our asset recovery and incorporation process.

The remainder of the paper is structured as follows: Section 2 presents the elements of the integrated architecture reconstruction approach, while Section 3 instantiates and deepens the approach by combining static, dynamic and documentation reconstruction techniques demonstrating the combination benefits in a case study centered around Eclipse plug-in development. Section 4 draws some conclusions.

2. APPROACH
The reconstruction of legacy systems usually aims at recovering the legacy architecture or the as-built architecture as currently implemented in one of the existing legacy products. To achieve these goals, the existing legacy documentation (e.g., user manuals, architectural descriptions, or requirements documentation) even if outdated and the source code are analyzed. The analyses can either be static (i.e., the systems are not executed) or dynamic (i.e., runtime traces of the execution of the systems are collected) and the results are processed and stored into a single system asset base or a fact base. A fact thereby is exactly one piece of information about an existing system. Synergy effects by the combination of different analysis types may be achieved.

The integrated reconstruction approach we propose in this paper uses the same analyses techniques and operates as well on the single system asset bases as described in the last paragraph but the approach is a product line reconstruction approach as the goal is to populate the product line asset base. Therefore the integrated architecture reconstruction approach considers assets from all existing systems. The assets may even be similar, overlapping, or contradictory. Thus, the main differences to most other approaches (i.e., the novelty of our approach) are the goals that we address:

- We are only interested in the parts of the existing systems that provide product line relevant assets. These parts

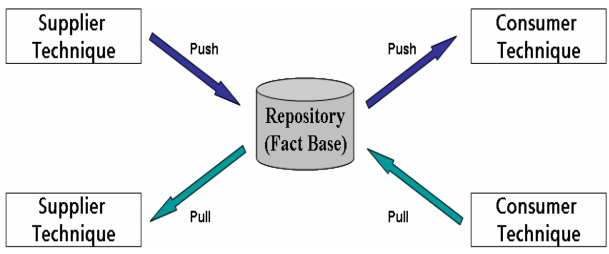


Figure 1: Push and Pull Model between Supplier and Consumer Techniques

comprise core functionality, key features, major variants, or architectural subsets relevant to the envisioned product line architecture and appealing for reuse. The legacy architecture of the existing system is of no interest so that we are able to work within a narrow scope (i.e., only the product line important assets or parts of the assets) and that leads to efficient way of conducting the analyses. By this distinction we focus the analyses only on relevant parts of the legacy systems.

- Having a prioritization of product line relevant assets, we combine requirements recovery techniques with static and dynamic architecture recovery techniques. The combination enables a better understanding of the asset's purpose and implementation and accelerates the reconstruction process since results produced by one technique are processed further by another technique. The existing assets are recovered and their adequacy with respect to the product line architecture is analyzed by identifying the amount of adaptation to make an asset usable within the product line.

The means to control and monitor the migration towards product line engineering is the migration plan. The migration plan defines which parts of the existing systems are of interest, which variants have to be analyzed, or which components have to be reconstructed. The migration plan is thus the steering vehicle of the product line architects, the reverse engineers, and the experts of the existing systems to coordinate and to schedule the different activities. Data extracted and information gained during analyses is stored in the fact base to serve as basis for further or detailed analyses.

The reconstruction yields product line component candidates that have to be assessed with respect to their adequacy. In presence of several legacy systems, this allows the identification of more than one component candidates completely or partially fulfilling the given requirements. To decide about reusing such existing components, the component's internal quality and suitability for the product line have to be evaluated to ensure that the component is able to serve the product line needs [7].

Technique Interaction Model

Static, dynamic and documentation analysis techniques are inherently beneficial techniques serving various purposes when viewing them as individual techniques. The same holds for different techniques from just one area (i.e., there are various techniques from all three areas that could interact and influence each other). Each technique contributes to a specific aspect when mining existing artifacts for information about the underlying architecture, realization of functionality, implementation of features, and rationale behind the decisions made.

Central to every technique is the fact base that stores, manages and enables fast access of the information gathered so far about the existing systems. Analysis results coming from one analysis techniques presented as views are handled and stored in the fact base as well. Often elements get annotated by comments of the system experts or interpretations of results when reviewing them. In particular, the facts about the system are basis for further documentation derived on top of the analysis or the facts. In addition it is worthwhile to note the analyses techniques where a fact has its origin and to include this information in the fact base as well in order to trace back the analysis results to the artifacts which contributed to the results.

The fact base can be regarded as some kind of repository that enables the population, processing, modification of the facts and the fact base. Fact bases usually contain different levels of data ranging from low level source code data (e.g., structural information like classes, methods, and variables, data and control flow information like call trees, inheritance trees, or abstract syntax trees) to abstract information (e.g., features, functionality and uses cases, architectural components and design rationale, etc.).

In order to enable an integrated architecture reconstruction with respect to mutual influences of techniques, the analyst applying one particular technique has to be made aware of other techniques. Since the techniques are often applied independent from each other, the fact base enables the sharing of the results.

Figure 1 presents the interaction of techniques by means of a supplier-consumer pattern where different techniques supply information about a software system to the fact base, or they consume information from it:

- **Supplier Technique**: The supplier techniques produce items of information about existing systems. Advanced supplier techniques combine already existing information from the fact base, to achieve new results or new viewpoints on the available information by selecting only particular aspects. Supplier techniques are filling the fact base with information that then is used for further analysis.
- **Consumer Technique**: The consumer techniques consume the item of the fact base in order to be able to perform the analysis technique. The consumer benefits from the existing information already available in the fact base. Consumer techniques typically aggregate information from lower level to higher levels of abstraction. Consumer techniques exploit the fact base for their purposes and assume certain information to be pre base in order to be conducted by the reverse engineer.

Depending on the context, in which a particular technique is applied it can be both, consumer and supplier. Next to the classification into supplier and consumer techniques, Figure 1 presents two modes of operation for the supplier-consumer pattern:

- **Push Model**: In the push model, a particular technique pushes the information it produces into the fact base, so that the analyst can be made aware of it at a later point of time, and that the results can then be pushed further into the consuming technique. The consumer uses all the information that is already contained in the fact base. A typical example for a supplier technique following the push model is the fact extraction, since the reverse engineer is doing this in almost every case, and the results of the fact

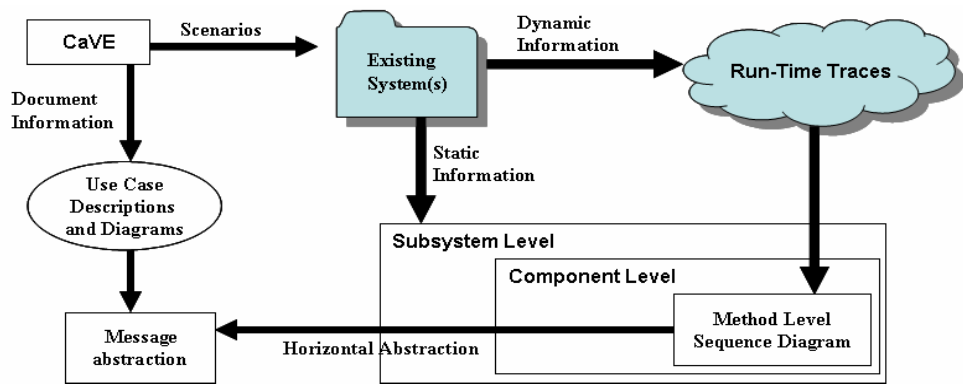


Figure 2: Combination of Techniques

extraction activity are then pushed as input to other analysis techniques (e.g., architectural evaluation, metrics computation), which then process the inputs further or combine different inputs together.

- **Pull Model**: In the pull model mode of operation the analyst (or reverse engineer) explicitly triggers other analysis techniques that contribute to the consumer technique. In this case, the analyses are delayed until the supplier techniques have been conducted, and the results have been stored in the fact base. Examples for the pull model mode of operation are advanced clustering techniques that take special similarity metrics into account in order to cluster the structural entities of a system into groups. These similarity metrics are required in order to apply the clustering technique, so they have to be computed first. Since these metrics are of limited interest for other analyses, the reverse engineers will trigger the computation on demand and wait until the results have been supplied.

In our approach, we use all techniques either as push or as pull techniques. Depending on the situation, the order of the techniques is different and so none of the techniques can rely on information that is already in the fact base. But with this model we can make sure that we always can build up the facts that are needed for the different techniques to proceed.

3. TECHNIQUE COMBINATION

We now describe different constellations of types of reverse engineering techniques from an abstract viewpoint and present where the techniques have overlaps, where a result flow is possible, and what the potential benefits of specific combinations are. We illustrate the combination with partial results of a case study. The case study is centered on Eclipse. Two programming language plug-ins of the open source tool platform Eclipse, namely JDT (Java Development Tools, a Java IDE supporting the development of any Java application) and CDT (C/C++ Development Tools), are the subject of the case study. The technique combination in the case study is shown in Figure 2. Our purpose was to learn and eventually to reuse key features of JDT and CDT (e.g., model management, persistency, file system synchronization) for our own product line of Eclipse plug-ins, so we wanted to find out about common and variable features, code and behavior of JDT and CDT. In particular, we reused (and adapted) concepts and components to develop a common core that became part of three plug-ins centered on the analysis of product line architectures. An overview on the technique combination can be found in Table 1. In this section, we will focus on the dynamic aspects.

Our approach for reconstructing the architectural views follows the extract-abstract-present paradigm. In the extraction phase,

	Requirements Recovery	Static Architecture Recovery	Dynamic Architecture Recovery
Requirements Recovery		Check Completeness and Consistency Input for Architecture Scenarios Give Start Architecture Variations in Start Architecture Rationale Extraction	Check Completeness Input for Scenarios Variation in Scenarios Identification of user visible test cases and scenarios
Static Architecture Recovery	Input for document selection feature selection requirements structuring		Horizontal Abstraction (Collapsing traces) Static Architectural Views (e.g. structural view) Comparison of static and dynamic behavior like coupling and cohesion
Dynamic Architecture Recovery	Input for document selection Documentation of non-functional requirements	Dynamic/Behavioral Architectural Views Dynamic Metrics Input for test coverage estimations	

Table 1: Overview Static and Dynamic Architecture Recovery and Requirements Recovery

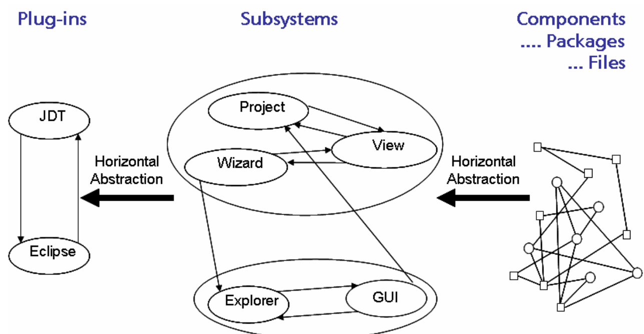


Figure 4: Static Abstractions for the JDT

both the source code and runtime traces are used to build a fact base. Usually this fact base contains low level information (for example, function calls). In the abstraction phase, using the collected static and dynamic information, high level views are reconstructed. Finally, in the presentation phase, the extracted views are presented for analysis. We use hierarchical graphs for visualizing the static structure of the software and the message sequence charts for visualizing the runtime behavior of the software. Table 1 depicts an overview on the three different recovery areas (static architecture recovery, dynamic architecture recovery, and requirements recovery). Our case study involved the following techniques in all three areas (namely static architecture recovery: SAVE [6] and interface analysis, dynamic architecture recovery: dynamic analysis with JRat [5], and requirements recovery: CaVE [4]).

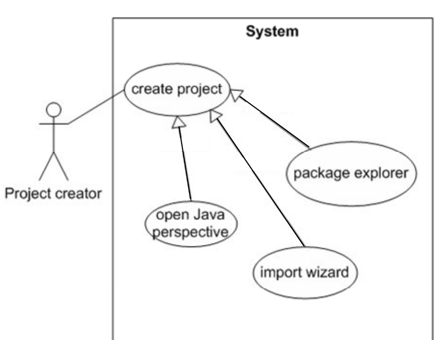
Document Analysis: As input for the document analysis, for JDT and CDT exists a large amount of additional resources, like FAQs, Tutorials, user guides and designer documentation. A selection of these documents was analyzed by non-domain experts with the CaVE approach. The emphasis thereby was on the JDT and CDT project management, for which we extracted concepts, features and use cases, relationships. Each plug-in was analyzed independently by a non domain expert resulting in a conceptual architectural view. Defining conceptual components and assigning entities and domain concepts to components was left to the follow up analysis.

Static analysis: Static architecture evaluation in this case study was done for the JDT plug-in using the SAVE method. SAVE can be used to compare the architects "intended" architecture (i.e., their mental model of the architecture) with the systems "as is" or architecture (i.e., the architecture as it is implemented in the source code). However if such an architectural description is missing at all, it can be used in similar way for reconstructing architecture.

The SAVE tool requires an initial architectural description as entry point. We used the conceptual model produced during document analysis and a source code model produced by parsing as input. The document analysis results provided insights about the architectural key entities, which are not directly visible from extracted facts from the source code. During several iterations the initial architectural view and the source code model were aligned in a semi automatic mode. The number of lower-level components was reduced through merging them into to conceptual components on a higher level of abstraction. This was obviously necessary, as the initial source code model was too close to the implementation for an architectural description. The analysis revealed a model-based abstraction layer for Java language concepts and a component managing the Java model. The concept of a buffer for source code was found to effectively propagate changes from Java model elements (managed by the *JavaModelManager*) to the underlying file system and vice versa. All physical elements (.java files, packages) were represented as model elements of JDT and managed efficiently by a last-recently-used (LRU) cache. This strategy is effective in model element access and consumes less memory. The rationales help the product line architects in understanding the main concepts for managing JDT projects.

Dynamic Analysis: In the case study we identified hot spots in a system implemented in Java using dynamic analysis. With hot spots we mean the locations in the source code that have performance bottlenecks. On top of already existing tool JRat [5] we developed a framework which allows performance measurement of asynchronously communicating components to make requests traceable. Because instrumentation is an overhead, our environment uses system experts to identify certain key locations in it. Even for a simple runtime measurement (measuring the time between issuing a service request and getting the response) it was necessary to change the existing code of the product to be measured in order to make time usage of a single request traceable. Asynchronous communication did not allow the mapping of a method call to a specific user request. Each message had to be stamped with that information.

The main purpose of combining the low level information is to gain information on how to abstract the source code elements. Figure 4 depicts an example on how low-level source code elements can be abstracted to architectural entities, or on the highest level in the Eclipse context, to the plug-in level




```
Required Routines of
Class   org.eclipse.jdt.core.JavaCore
Method
create(org.eclipse.core.resources.IProject)

        org.eclipse.jdt.internal.core.
         JavaModel.getJavaProject
         (org.eclipse.core.resources.IResource)
        org.eclipse.jdt.internal.core.
         JavaModelManager.getJavaModelManager()
        org.eclipse.jdt.internal.core.
        JavaModelManager.getJavaModel()
```

Figure 3: Extracted Information from JDT Documentation and Code

capturing how plug-ins interact with the Eclipse platform. Such abstractions are needed in order to cope with the complexity of the source code elements. The details capture how the source code entities interact, but the big picture is crucial to enable the main understanding and to facilitate the architects finding their way in understanding the software systems.

In the behavioral view, the interaction starts with a user triggering the creation of a new Java Project. The framework delegates the call to the JDT-GUI component where the Java project wizard is located. After the user entered the information necessary for project creation and pressed finish, the wizard creates a generic project in the Eclipse workspace. Moreover the wizard adds a Java nature to the project, so it can be distinguished from other projects in the workspace. The framework opens the perspective associated with Java projects (if not already open), i.e. several tree viewers and editors associated with Java development are opened. The creation of the generic project in the Eclipse workspace is triggered to all listeners interested in these events. In that case it is the Java Model (more concrete: the JavaModelManager) who has subscribed to these events. The model manager figures out that the newly created project has the Java nature. Consequently he translates the Eclipse workspace changes into Java model events. In the case the creation of a generic project that has a Java nature leads to the creation of a new JavaProject in the Java model. Internally the JDT-model now triggers the model changes to the JDT-GUI component, so all the Java specific viewers and editors that are registered with the Java model get notified and can update their views.

The runtime analysis process was performed and a behavioral view could be extracted as it is shown in Figure 5. The horizontal abstractions were provided by static analyses as described above, while the component interaction was collected in run-time traces, the messages sent were derived from use case scenario abstractions. This view completes the views already created by static and document analysis.

The analysis of the model management in JDT and CDT enabled us to reuse the same mechanisms and the source code components (with adaptations) for a product line of architecture

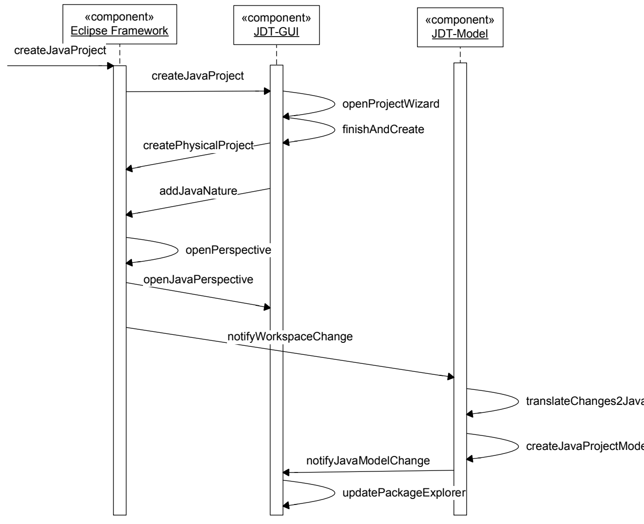


Figure 5: Behavioral view of Use Case

analyses tools. The reused parts have been merged into a common core that, up to now, has been instantiated for three of our own plug-ins. We are confident that our analysis effort was well-invested and now pays off in a mature and field-tested model management. The reuse goals drove the reconstruction activities and due to the integrated approach we were able to focus the activities only on those parts relevant to the product line architecture analysis plug-ins we developed.

4. CONCLUSION

This position paper described an integrated architecture reconstruction approach that explicitly combines analyses techniques from different areas, namely static architecture recovery, dynamic architecture recovery, and requirements recovery based on documentation. The integrated architecture reconstruction approach combines architecture recovery and requirements recovery to exploit existing artifacts of software systems. It is a framework for the systematical reconstruction of existing systems, especially in a product line context. In particular, the focus is set on the interaction of different analysis technique that process artifacts or asset, and that cover the whole range of available artifacts (i.e., not only the pure source code).

5. REFERENCES

[1] P. Clements, L. Northrop: Software Product Lines. Addison Wesley, 2001.

[2] S. Eick: An Interactive Visualization for Message Sequence Charts. Proceedings of 4th ICPC, 1996.

[3] P. Finnigan, R. Holt et al.: The software bookshelf. IBM Systems Journal, 36(4), November 1997

[4] I. John. Capturing Product Line Information from Legacy User Documentation. In: Käkölä, Timo; Dueñas, Juan Carlos (Eds.) Software Product Lines, Research Issues in Engineering and Management. Springer 2006

[5] JRat, http://jrat.sourceforge.net , June 2006.

[6] J. Knodel et al.: Static Evaluation of Software Architectures. 10th CSMR, Bari, Italy.

[7] J. Knodel, et al.: Asset Recovery and Their Incorporation into Product Lines. Proc. of 12th WCRE, Pittsburgh, 2005.

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

[9] W. Wong et al: Locating Program Features Using Execution Slices, In Proc. of the IEEE Symp. on Application-Specific Systems and Software Engeering and Technology, pp. 194-203, Mar. 1999.

[10] K. Wong: The Rigi User's Manual, 1998.

[11] H. Yan et al.: DiscoTect: A System for Discovering Architectures from Running Systems. Proceedings of ICSE, 2004.

Higher Abstractions for Dynamic Analysis

Marcus Denker, Orla Greevy
Software Composition Group
University of Berne
Switzerland

Michele Lanza
Faculty of Informatics
University of Lugano
Switzerland

ABSTRACT

The developers of tools for dynamic analysis are faced with choosing from the many approaches to gathering runtime data. Typically, dynamic analysis involves instrumenting the program under investigation to record its runtime behavior. Current approaches for byte-code based systems like Java and Smalltalk rely often on inserting byte-code into the program under analysis. However, detailed knowledge of the target programming language or virtual machine is required to implement dynamic analysis tools. Obtaining and exploiting this knowledge to build better analysis tools is cumbersome and often distracts the tool builder from the actual goal, which is the analysis of the runtime behavior of a system.

In this paper, we argue that we need to adopt a higher level view of a software system when considering the task of abstracting runtime information. We focus on object-oriented virtual machine based languages. We want to be able to deal with the runtime system as a collection of reified first-class entities. We propose to achieve this by introducing a layer of abstraction, *i.e.*, a behavioral middle layer. This has the advantage that the task of collecting dynamic information is not concerned with low level details of a specific language or virtual machine. The positive effect of such a behavioral middle layer is twofold: on the one hand it provides us with a standard API for all dynamic analysis based tools to use, on the other hand it allows the tool developer to abstract from the actual implementation technique.

Keywords

Dynamic Analysis, Behavioral Reflection, Meta Programming, Tracing

1. INTRODUCTION

In recent years there has been a revival of interest in dynamic analysis [16]. System analysis of runtime behavior is vital for performance analysis to detect hotspots of activity and bottlenecks of execution or memory allocation problems such as unnecessary object retention. In a reverse engineering context, dynamic analysis is used is to extract high-level views about the behavior of low-level components to facilitate the comprehension of the system [15, 17, 31]. The focus of analysis determines the relevance and level of detail of the information captured during dynamic analysis. In the field of reverse engineering, there is no consensus on the type or level of granularity of runtime information that is necessary for a particular analysis. An inherent requirement of such tools is that they be easily extensible as the requirements and the research focus evolves.

Dynamic analysis yields precise information about the runtime behavior of systems [2]. However, the task of writing tools to abstract runtime data is not trivial. Developers of tools are faced with many options as there are numerous techniques that address the task of collecting runtime data. The approach to tool development and the abstraction of dynamic data is therefore not standardized. Each individual tool adopts a specific technique and focuses on low-level details of the chosen technique to achieve its goals.

This leads to recurrent problems of all approaches and techniques:

- all clients need to re-implement large parts of the code responsible for gathering the runtime data, and
- the abstraction level is too low in the sense that clients need to know too much about the implementation side.

In this paper we propose the introduction of an abstraction layer based on *behavioral reflection* to facilitate the design and development of tools for dynamic analysis. We introduce our reflection framework and identify its strengths and shortcomings.

Structure of the paper. In the next section we identify some typical applications of dynamic analysis. In this context we outline the state of the art in gathering dynamic data at runtime. Section 3 then shows problems and shortcomings associated with the current approaches. In Section 4 we give an overview of reflection frameworks. In Section 5 we introduce our reflection framework and identify how it can be used to solve the problems shown, and we identify what is missing from our implementation with the intention of initiating a discussion and obtaining feedback. Section 7 outlines our conclusions and future work.

2. DYNAMIC ANALYSIS TECHNIQUES

Dynamic analysis typically involves instrumenting the program under investigation to examine or record certain aspects of its runtime behavior. Code instrumentation is a mechanism that allows insertion of code at runtime to monitor and track the runtime behavior of a system. In this section we review the techniques currently available for abstracting the runtime behavior of a system. The underlying concepts behind dynamic analysis tools are currently limited to using these techniques for extracting dynamic information [17].

The granularity and amount of behavioral data extracted during the execution of a system varies depending on the specific focus of a particular analysis tool. Dynamic analysis implies vast amounts of data. A simple execution of a system's functionality can result in a large number of events [10]. Typically, dynamic analysis tools employ filtering and compression techniques to limit the amount of dynamic data collected depending on a specific focus of the analysis. For example, if the goal of the analysis is feature location [11], this requires that a relationship between the external functionalities of a system and the parts of the code that implement this functionality is established. Therefore, it is usually sufficient to extract trace events representing the method calls performed during the execution of a specific functionality [1, 15]. An example of trace representation is given by Richner and Ducasse [27]. Each line records the class of the sender, the identity of the sender, the class of the receiver, the identity of the receiver and the method invoked. The order of the calls is maintained.

Analysis techniques that focus on monitoring the state of objects at runtime require a more detailed analysis to extract information about variable access. This level of granularity is required if the focus of the analysis is to infer program invariants [12]. Performance analysis tools generally focus on object creation and deletion and the correct memory allocation details. Thus the requirements of dynamic analysis tools vary depending on their specific focus. This is a drawback, because the analysis goals should not restrict the type of information to be collected. We want to extract as much dynamic data as possible and then depending on the requirements of a particular analysis, extract and use a appropriate subset of the dynamic data.

There are various approaches and language-specific frameworks that support the extraction of dynamic information. We describe the details of the underlying mechanisms used by dynamic analysis tools in the following paragraphs.

Source code modification. One way to control message passing is to instrument the code via source code modification and recompilation. The main drawback of this technique is that all controlled methods have to be reparsed and recompiled. Furthermore, another recompilation is needed to reinstall the original methods.

Bytecode modification. Another way to control message passing is to directly insert byte-code into the byte-code of the compiled methods. The introduced byte-code controls the message passing. However, this technique relies heavily on profound knowledge of the bytecode instructions used by the virtual machines. Another potential danger of this technique is that these codes are not standardized and can change.

Instrumenting the Virtual Machine. A technique for collecting runtime information is instrumenting the runtime environment in which the system runs. For example, the Java Virtual Machine can be instrumented to generate events of interest. The advantage of this technique is that it does not require modification of the source code.

The Java Virtual Machine Profiling interface (JVMPI) [20] provides a set of hooks to the JVM to signal interesting events, such as thread starts or object allocations. JVMTI [21] is the sucessor to JVMPI and provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine. It provides additional facilities for bytecode instrumentation. Profilers based on JVMPI or JVMTI interfaces implement profiling agents to intercept various events, such as method invocations. Unfortunately these profiling agents have to be written in platform native code, contradicting the Java motto of "write once run anywhere". Binder developed Komorium, a novel sampling profiler written purely in Java that directly instruments the bytecode of Java programs [4]. Another pure Java solution is the Java Interactive Profiler (JIP) is based on JVMTI and provides control to turn on and off profiling at runtime (see `http://jiprof.sourceforge.net/`).

Method Wrappers. Brant *et al*, describe a code instrumenting technique for Smalltalk based on method wrappers [5]. Wrappers are mechanisms for introducing new behavior that is executed before and/or after, and perhaps instead of, an existing method. Rather than changing method lookup, they modify the method objects that the processes return. Wrappers are easy to build for Smalltalk as it was designed with reflective facilities that allow programmers to intervene in the lookup process, while the same is not true for Java which only supports introspection.

Debuggers. It is possible to run a system under the control of the debugger. In this case, breakpoints are set at locations of interest (*e.g.*, entry and exit of a method). This technique has the advantage of not modifiying the source code and the environment. However it slows down the execution of a system considerably, and the instrumentation itself can be cumbersome. This can be done on the source level before compilations, or on bytecode at load time (Java) or runtime (Smalltalk). The abstraction layers we deal with are either those of the program text or those of bytecode.

Logging Services. Logging can be used to track the state of a running system at various points in time. A good framework for doing this with Java is provided by log4j (see `http://logging.apache.org/`). The drawback is again that this implies modifying the source code.

3. CHALLENGES

As we have seen, there are multiple implementation techniques for gathering runtime data. The key problem is that every new client application has to re-implement large parts of the runtime data gathering code depending on the language and runtime environment. Furthermore, the abstraction is too low level, resulting in clients that are concerned with manipulating too many implementation details.

In the following sections we elaborate on the main problems of the current approaches.

3.1 Instrumentation re-implemented

Most projects that require to access runtime data typically re-implement the data gathering mechanism. Instrumentation code is inserted at all places of interest in the code

(*e.g.*, at message sends). In the case of bytecode manipulation techniques, the actual modification of the bytecode is achieved using libraries (*e.g.*, Javassist [8, 7] or Bytesurgeon [9]). Bytecode manipulation provides a very low level of abstraction and requires detailed knowledge of the bytecode of the programming language. Moreover, it is prone to language evolution, *i.e.*, the specification of the VM may change.

The positive effect of the low level approach is of course that we build a very specific implementation, tailored exactly towards the information needed for a specific task. The drawback is that the instrumentation logic is tightly coupled with the application that requires the dynamic data, thus in most cases we will have to re-implement the instrumentation logic for each new task. Figure 1 shows an example: We have two projects that are interested in gathering runtime data (Tracer1 and Tracer2). Although both run on a standard virtual machine, both independently implement the code for bytecode instrumentation. We have seen this happen often in our research, for example both the trace debugger Unstuck [19] and a test coverage tool both utilized the same byte-code manipulation library (ByteSurgeon), but they did not share any instrumentation code.

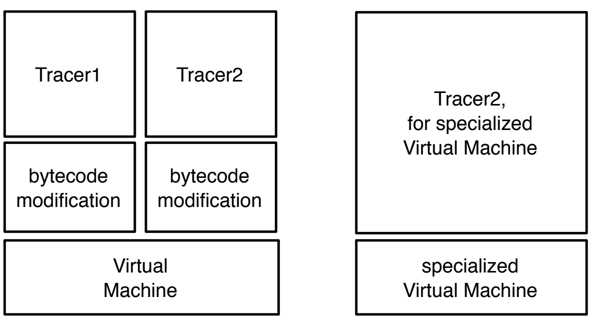


Figure 1: Trace tool today

3.2 Implementation Specific Designs

Implementors of tools that gather runtime data are faced with the decision on which implementation technique to adopt. Subsequently, they design a tool based on specific knowledge of the target language and runtime environment. The resulting tool inevitably is forced to encode implementation issues into the design of the tool. Thus, the result is a tool that is tightly coupled with a particular runtime environment.

This approach has obvious drawbacks. It is very difficult to change the adopted implementation technique: *e.g.*, bytecode manipulation is portable, but a specialized virtual machine might be faster. When the implementation technique is closely tied to a particular virtual machine, we are bound to this one implementation, we cannot switch to a byte-code based implementation on a case-to-case basis. Figure 1 shows that for a special virtual machine, we need to re-implement our system.

4. BEHAVIORAL REFLECTION

Systems like Smalltalk and CLOS model their own structures (classes, methods) as first class objects.

The term *introspection* defines the ability to query the system about this information, whereas we talk about *intercession* when changes to these structures directly change the structure of the system itself. Systems that provide both are called *reflective*.

Structural reflection is concerned with reification of the program and its abstract types. *Behavioral reflection*, on the other hand, is concerned with the ability of the language to provide complete reification of its own semantics and implementation as well as complete reification of the data and implementation of the runtime system.

Popular object oriented languages provide different degrees of introspection or reflective capabilities. Smalltalk is, to some extent, a reflective system [13, 3]: Classes and methods are objects, we can change those objects to change the structure of the system. Java and .NET on the other hand, have only introspective features (*i.e.*, allows for querying an object for its class, a class for its methods and constructors, query the details of those methods and constructors, and tell those methods to execute), and partial intercession (intercession is limited to method invocation and attribute manipulation) [6].

Languages that facilitate the description of methods as first class objects inherently support dynamic analysis. The method wrappers technique exploits the first class nature of methods in Smalltalk for providing a way to intercept method execution [5]. Examples of dynamic analysis tools built on the method wrapper technique are Greevy and Ducasse's TraceScraper tool for feature analysis [15] and John Brant's Interaction Diagram and Coverage Tools [5]. However method execution is just an aspect of runtime information. For a complete dynamic analysis we need to focus on other runtime events such as *e.g.*, *message sends between object instances* or *instance variable access*. Thus, we recognize the need to define a reflective meta representation that describes all behavioral aspects of systems. We want a system that can reify those events on demand, providing a system with full behavioral reflection.

In both Java and Smalltalk, the reflection mechanisms provided are concerned mostly with structure. They do not provide an easy way to change the semantics of the runtime model: Message sends, instance variable access are not modeled with objects. A true behavioral reflective system models behavior in a way that it is first class and changes are possible: *e.g.*, we are able to define our own version of what a message send is.

Looking back into the history of object oriented systems, we can find that there has been extensive research on behavioral reflective systems, *e.g.*, the work done around Meta Object Protocols [22] for CLOS. The meta object protocol provides all operations (*e.g.*, method activation or variable access) to be reified and re-defined.

In systems like Java and Smalltalk, behavioral reflection can be realised via special virtual machines or bytecode manipulation, with the latter being portable. Examples for the virtual machine approach are Iguana/J[26], Metaxa [14], or Guarana [25]. The prime example for a bytecode modification based meta object protocol is Reflex [30]. Reflex provides a model for behavioral reflection that allows for a very fine grained selection of when and what to reify.

5. THE BEHAVIORAL FRAMEWORK

The drawbacks we have identified with current approaches

lead us to suggest that the solution would be to introduce an additional layer of abstraction to our system, which we refer to as a *behavioral framework*.

We now analyze how a behavioral reflection framework could be used to tackle and solve the problems of previous approaches to gathering runtime information.

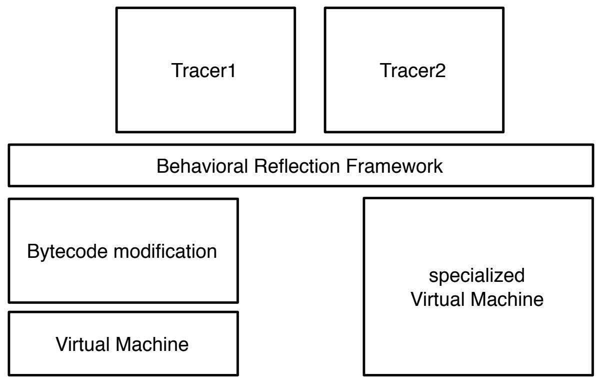


Figure 2: A common abstraction layer

5.1 A Shared API

With the existence of a behavioral layer, all tools could use it to hook into runtime events. The individual tools are no longer concerned with a specific code insertion implementation. Instead, they just leverage the abstractions provided by the behavioral layer framework.

In Figure 2 we see again our two tools that are interested in dynamic information. Now both tools just use the behavioral layer, thus they do not need to implement the byte-code modification code themselves, but share it.

5.2 A Pluggable Implementation

Another important requirement of an abstraction layer is to provide a high degree of flexibility, but at the same time retain the same interface for clients. The proposed behavioral framework should make it possible to have a pluggable implementation (the backend): it can be realized via editing byte-code, a changed virtual machine or other means.

Figure 2 shows how we now can use both programs on the modified virtual machine, without having to implement the logic ourselves: All tools using the abstraction layer will work on both the standard virtual machine and any specialized virtual machine that the abstraction layer supports.

5.3 Requirements

In the following we identify a list of requirements for a behavioral framework to tackle the challenges we identified previously.

Runtime installation: We need to introduce behavior dynamically at runtime. When we are done with the analysis, it should be possible to revert to the original state of the system.

Unanticipated use: The behavioral change should be possible at any time in the deployed system, without the need to prepare the system in any way at startup.

Fine-Grained Selection: The operation occurrences we are interested in are different depending on what we analyze. We want to be able to select the entities up to the level of the single occurrence in the code.

Implementation Hiding: From a dynamic analysis perspective, we are not interested in the underlying mechanisms of obtaining runtime information. The fundamental goal of a behavioral layer is to allow us to abstract from the details of a specific implementation technique (*e.g.*, VM change, byte-code extension, byte-code modification) used to extract behavioral information from an application at runtime.

Performance: To make the framework usable for analyzing real work applications, we need a framework with low overhead. The best case would be a system where we pay exactly the same overhead as if we were to annotate the code with profiling calls by hand.

5.4 Implementation

We have realized a framework for partial behavioral reflection for Squeak (a dialect of Smalltalk) called Geppetto[28]. Geppetto uses the runtime byte-code transformation framework ByteSurgeon[9] and follows the model of partial behavioral reflection as pioneered by Reflex[30]. Unlike Reflex, which is constrained by the underlying model of the Java language, our Geppetto implementation can be used completely unanticipated: code does not need to be prepared at load or compile time, reflection can be enabled at runtime and completely retracted when not needed.

Geppetto allows for reifying message sending, method execution and variable access (read and write) for both instance variables and temporary variables. Selection is very fine-grained: per package, class, object, method, operation and operation occurrence. Geppetto can be used in any Squeak program, without the need to adapt it at load or start time. Installation happens transparently at runtime.

Geppetto uses ByteSurgeon to insert small peaces of code, so called *hooks* into the bytecode where a selected operation (e.g. message send) occurs. Figure 3 shows the model in detail. Hooks are grouped to *hooksets*, which are bound to a *metaobject* by a *link*. The *link* defines the protocol between the base and the meta layer. Links can be enabled or disabled based on an *activation condition*.

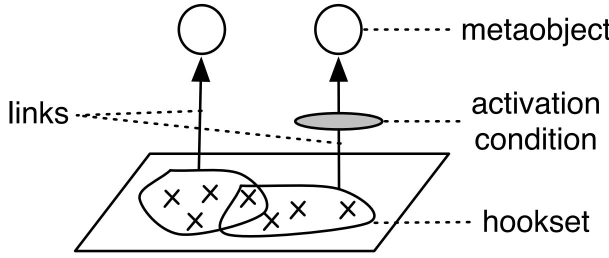


Figure 3: Hooksets, Links and Metaobjects in Geppetto

For a complete description of the Gepetto behavioral reflection framework, see [28].

5.5 Usage

The behavioral reflection framework provides a general API: the reification of runtime events triggers calls to meta objects, which are instances of normal classes. The tool developer thus is free to use the framework as needed by specifying which concepts to reify and which information to pass on to the meta object. The framework does not provide a model of the data obtained (*e.g.*, a trace), instead it provides *a model for obtaining data*. It can be either stored for later use as a trace or processed and reacted on at runtime. The latter has lately become an active topic of research with systems like PQL [24].

6. DISCUSSION

We now analyze our behavioral framework with respect to the requirements defined in the preceding section and define future work. Then we briefly discuss the relationship to aspect oriented programming and the usefulness of providing scoping abstractions as part of the framework.

6.1 Next Steps

The implementation as described in section 5.4, already fulfills some of the requirements stated: It can install (and retract) behavioral changes at runtime, provides fine-grained spatial and temporal selection by implementing the Reflex model [30] and supports unanticipated use.

Two requirements are not yet fulfilled:

1. Geppetto needs to be extended to support pluggable backends. We are working on providing a backend based on annotated abstract syntax trees.

2. We need to verify the real world usability: first benchmarks show good performance characteristics, but Geppetto needs to be validated with real world usage. We plan to move the tools and experiments done that currently use ByteSurgeon to use Geppetto instead.

6.2 Aspects

This paper presents the solution from the perspective of behavioral reflection. Another point of view can be that of Aspect Oriented Programming. The proposed abstraction layer could use, as a backend, an existing dynamic aspects implementation. In this case, the aspect framework would be used as a high-level replacement for byte-code manipulation.

Another possibility would be to formulate the middle layer in terms of a dynamic aspect framework instead of meta objects. The problem here is that most aspect systems (*e.g.*, AspectJ [23]) are static: weaving happens at compile or load time. Pure runtime Aspects are not yet very common and those that exist are based themselves in some cases on behavioral reflection facilities, for example AspectS[18] and aspect systems based on Reflex[29].

6.3 Scope Abstractions

Modern implementations like Reflex provide very fine-grained spatial and temporal selection of reification. Here we can select *what* and *where*, in a temporal and spacial way.

This means we can scope the reification towards collections of classes (like modules and packages) or single instances, a single methods of a class, or even to one certain occurrence of a behavioral event. Temporal selection means that we can switch reifications on and off at will, thus we can make the gathering of runtime data be controlled by runtime events.

Another idea of scoping is that of scoping-towards-the-client: We might be interested in events generated only if our package under test is called from a certain other package. This can be useful to limit the amount of unnecessary data when *e.g.*, analysing system classes like Smalltalks collections.

7. CONCLUSION

In this paper we addressed a fundamental problem that faces the developers of tools that exploit runtime information of an application. We propose an new approach to designing dynamic analysis tools for virtual machine based languages that interact with a layer of abstraction, namely a behavioral layer. The behavioral layer should provide a framework for tool developers that encapsulate typical object oriented language constructs at runtime such as object instantiation, message sends and instance variable access. Thus the developer has access to reified first class entities of runtime behavior and focuses on these high level abstractions when designing a specific tool. The main advantage of this layer of abstraction is that the resulting tool should easily portable to use with other virtual machines as the reified entities are independent of the underlying implementation details and byte-codes. Moreover the developer is not concerned with low level details that are specific to a particular virtual machine.

In this paper we provided a short overview of the available technologies and approaches to extract runtime data. We identified problems inherent to these approaches. This motivates our argument that there is a need to introduce a layer of abstraction between low level implementation details and the tools analysing the data.

To better understand the underlying motivation of a behavioral layer we provided a short overview of some of the applications of dynamic analysis. In the field of program comprehension and reverse engineering dynamic analysis approaches are becoming more prevalent. However there is no standard approach to extracting runtime data nor is it clear which type of runtime information to extract. Therefore such tools need to be extensible, as requirements change.

We identified a list of requirements for a behavioral layer. We describe our current implementation of a behavioral layer and illustrate how it can be used to address the problems. We show how we simplify the task of implementing dynamic analysis tools.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Analyzing, capturing and taming software change” (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008)) and “NOREX - Network of Reengineering Expertise” (SNF SCOPES Project No. IB7320-110997).

8. REFERENCES

[1] Giuliano Antoniol and Yann-Gaël 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, September 2005. IEEE Computer Society Press.

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

[3] Alexandre Bergel and Marcus Denker. Prototyping languages, related constructs and tools with Squeak. In *In Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06)*, July 2006.

[4] Walter Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS-2005)*, volume 3780 of *LNCS*, pages 178–194, Tsukuba, Japan, nov 2005.

[5] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. method wrappers.

[6] Walter Cazzola. Smartreflection: Efficient introspection in java. *Journal of Object Technology*, 3(11), August 2004.

[7] S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of GPCE'03*, volume 2830 of *LNCS*, pages 364–376, 2003.

[8] Shigeru Chiba. Load-time structural reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[9] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[10] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.

[11] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[12] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of ICSE '99*, May 1999.

[13] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

[14] Michael Golm and Jürgen Kleinöder. Jumping to the meta level: Behavioral reflection can be fast and flexible. In *Reflection*, pages 22–39, 1999.

[15] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society Press.

[16] A. Hamou-Lhadj. The concept of trace summarization. In *Proceedings of PCODA 2005 (1st International Workshop on Program Comprehension through Dynamic Analysis)*. IEEE Computer Society Press, 2005.

[17] 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.

[18] Robert Hirschfeld. AspectS – aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[19] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, 2006.

[20] Sun microsystems, inc. jvm profiler interface (jvmpi).

[21] Sun microsystems, inc. jvm tool interface (jvmti).

[22] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceeding ECOOP 2001*, number 2072 in LNCS, pages 327–353. Springer Verlag, 2001.

[24] Mickael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 363–385, New York, NY, USA, 2005. ACM Press.

[25] A. Olivia and L. E. Buzato. The design and implementation of Guaraná. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, California, USA, May 1999.

[26] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[27] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2002)*, page 34, Los Alamitos CA, October 2002. IEEE Computer Society Press.

[28] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection. In *Proceedings of ISC 2006 (International Smalltalk Conference), LNCS, to appear*, 2006.

[29] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*, Tallin, Estonia, sep

2005.

[30] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[31] 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, March 2004. IEEE Computer Society Press.

Capturing How Objects Flow at Runtime

Adrian Lienhard [1], Stéphane Ducasse[2], Tudor Gîrba[1] and Oscar Nierstrasz[1]

[1] Software Composition Group, University of Bern, Switzerland
[2] LISTIC, Université de Savoie, France

Abstract

Most of today's dynamic analysis approaches are based on method traces. However, in the case of object-orientation understanding program execution by analyzing method traces is complicated because the behavior of a program depends on the sharing and the transfer of object references (aliasing). We argue that trace-based dynamic analysis is at a too low level of abstraction for object-oriented systems. We propose a new approach that captures the life cycle of objects by explicitly taking into account object aliasing and how aliases propagate during the execution of the program. In this paper, we present in detail our new meta-model and discuss future tracks opened by it.

1 Introduction

Understanding an object-oriented system is not easy if one relies only on static source code inspection [19]. Inheritance, and late-binding in particular, make a system hard to understand. The dynamic semantics of *self* (or *this*) produces yo-yo effects when following sequences of method calls [18]. Moreover, the method lookup depends on the receiver which in turn varies depending on the transfer of object references at runtime.

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [2, 17]. Dynamic analysis was first used for procedural programs for applications such as debuggers [5] or program analysis tools [15].

As object-oriented technology became more widespread, it was only natural that procedural analysis techniques were adapted to object-oriented languages. In this context many dynamic analysis techniques focus on only the execution trace as a sequence of message sends [11, 20, 1]. However, such approaches do not treat the characteristics of object-oriented models explicitly.

Although dynamic analysis has the potential to overcome limitations of static source code inspection, it is not without its own limits. We identify the characteristic of *non-local effects* in object-orientation which renders program comprehension difficult and motivates a need for a dynamic analysis at the level of objects.

Nonlocal effects are possible due to *object aliasing*, which occurs when more than one reference to an object exists at the same time [10]. Objects often are not short-lived but are passed as arguments or return values and hence get aliased (or referenced) by multiple other objects. In this way, object aliasing introduces nonlocal effects: an object can be visible from different locations of the program at the same time and hence, through side-effects, a message sent to the object in one context can influence the behavior of the object in another context.

These effects are hard to understand from method traces alone because object aliasing and the transfer of aliases are not explicit. Another area in which object aliasing complicates the understanding of program execution is debugging. Although the debugger shows the current execution stack in which the error occurred, it is often hard to find the actual cause of the error because object references may have been incorrectly set at some distant time in the past.

The main contribution of this paper is a novel meta-model of object-oriented program execution which explicitly represents object aliasing, the transfer of aliases and the evolution of object state. We believe that such a model opens new ways of understanding the dynamics of object-oriented systems. To illustrate our approach we present three works in progress that are applications of our model

Outline. Section 2 discusses the object flow meta-model. Next we describe in Section 3 three example applications of our model. In Section 4 we describe the implementation of our prototype for obtaining the object flow information. Section 5 reports on the state-of-the-art and Section 6 presents the conclusions.

2 Representing the runtime space

We propose a novel model to capture how objects circulate or flow through a software system. Our model is intended to express the fact that an instance of a class can be

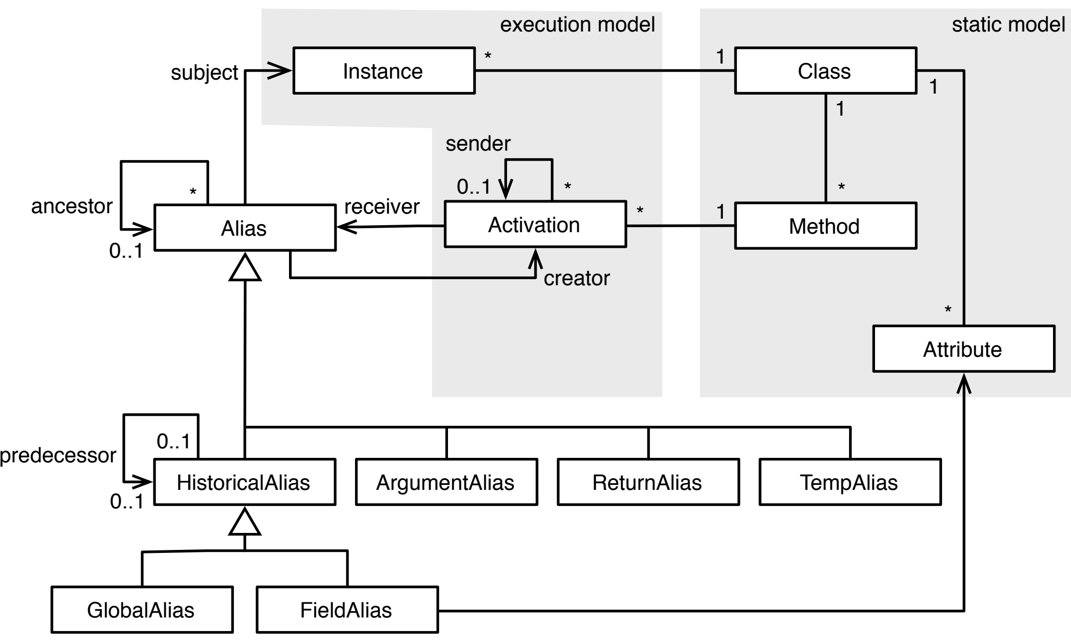


Figure 1. The object flow meta-model

referenced from several places at once, and that those references can be passed around to create other references. To capture this, we put at the center of our model the *alias*, as an explicit reification of an object reference.

Figure 1 shows the class diagram of our model. To get a complete picture of the system, we model the static part (*e.g.*, classes, methods, attributes), the execution part (*i.e.*, instances and method activations) and we complement these parts with aliases. In the following we detail the execution model and its integration with aliases.

Activation. An activation in our model represents a method execution. It holds the *sender* activation and the *method* it executes. This is very similar to commonly-used dynamic analysis approaches. Depending on the usage, approaches additionally identify the receiver instance for which a method is executed.

In our model, however, the *receiver* of an activation is the *alias* through which the message is sent. The fact that an activation does not directly reference an instance but rather an alias of this instance is an important property of our model.

Alias. We define an *alias* to be a first-class entity identifying a specific reference to an object in the analyzed program. An alias is created when:

- an object is instantiated,
- an object is stored in a field,
- an object is stored in a local variable,
- an object is passed as argument, and
- an object is returned from a method execution.

Except for the very first alias which stems from the object instantiation primitive, an alias can only be created from a previously existing alias, its *ancestor*. Based on this relationship we can construct the flow of objects. The flow shows where the instance is created and how it is then passed through the system. Since several new aliases can be created for each alias, the aliases of an instance form a tree.

Each alias is bound to its *creator*, a method activation. By *creator* we understand the activation in which the alias is first visible. For example, when passing an object as argument, the argument alias is created in the activation which handles the message received (rather than the activation in which the message was sent). The same holds for return values: the alias of a return value is created in the activation to which the object is returned.

The rationale is that aliases should belong to the activation in which an object becomes visible. Aliases of arguments, return values and temporary variables are only visible in the activation where they are created whereas field aliases may be accessed in other activations as well.

Special kinds of aliases include field and global aliases, as they additionally carry information about the evolution of the program's state. Field and global aliases point via their *predecessor* to the alias which was stored in the field before the assignment. The impact on our model is that it is capable of capturing the full history of the state evolution of objects. The predecessor reference enables backtracking of the state of objects to any point in the past.

3 Applications

We envision that our model has an impact both on reverse engineering and on forward engineering. In this section we describe three application examples that make use of the data captured in our model.

3.1 Relating Object Flows to Static Structure

The most straightforward application is relating the dynamic information to the static information. Figure 2 shows the hierarchy of the Smalltalk Squeak bytecode compiler [16], and on top of it we show how the aliases have traveled through the system at runtime. We emphasize in red the aliases of the particular instance that is the focus of our attention.

We envision several usages of such views. For example with such a view:

- We can check whether the path of the objects is what is expected.
- We can identify which classes play a primary role in the runtime object flow.

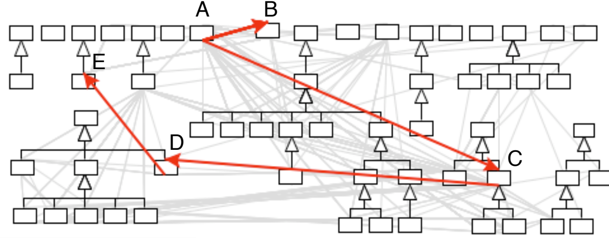


Figure 2. Example of several object flows mapped to a class hierarchy.

3.2 Characterizing Object Flows

Another application is to reason how a certain instance is aliased within the system. We are working on a simple visualization that captures the flow of an object by displaying a tree of aliases.

Figure 3 shows the same instance as in Figure 2, only now we emphasize the different kinds of aliases of this object using distinct colors.

Thus, the initial alias (1) is assigned to the field (2). The following six sequences of yellow and blue aliases (3-4) show that the instance is passed six times as argument to other objects in which it is then stored in a field. If we want to see in which class and method an alias is created,

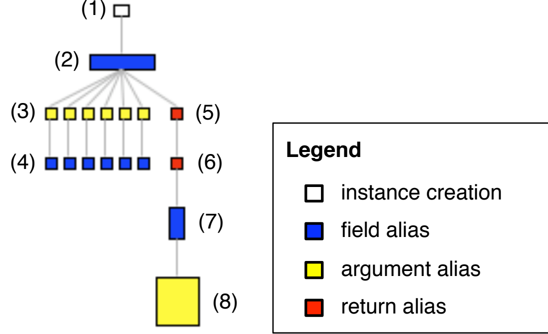


Figure 3. Example of object flow visualization of an instance.

our visualization tool [14] allows us to interactively get this information by moving the mouse over an alias.

The rightmost path shows that the object is passed as return value through (5) and (6) and is then stored in a field (7). Finally the object is passed by argument (8).

We also map metrics to the shape of the boxes in the diagrams [12]. We map the number of messages sent to the alias to the width of a box, and we map the number of messages sent from this alias to other objects to the height. In our example, the field alias (2) is wide which means that many messages have been sent to that alias. On the other hand, the alias (7) has a rather tall shape which means it sent more messages to other object than it received.

This visualization offers useful information regarding how the instance interacted with other objects during its life cycle. From the visualization of Figure 3 we can for example understand the following usage scenario of the instance: in a first stage it is set up, then it is passed around but is never used, and in the last stage, the object is used and itself interacts with other objects.

3.3 Object-centric Debugging

Another application area is debugging. In object-oriented programming, the understanding of problems is often complicated due to the *temporal and spatial gap* between the root cause and the effect of errors.

Figure 4 illustrates an example execution trace of a program. While the cause of the bug is introduced at the beginning of the execution, the effect occurs later (temporal gap). Figure 4 also shows the execution stack at the point when the error occurred. This is the typical view of a debugger showing the method activation in which the bug is manifested. The location of the cause of the bug, however, is hidden because objects have been passed around during execution (spatial gap).

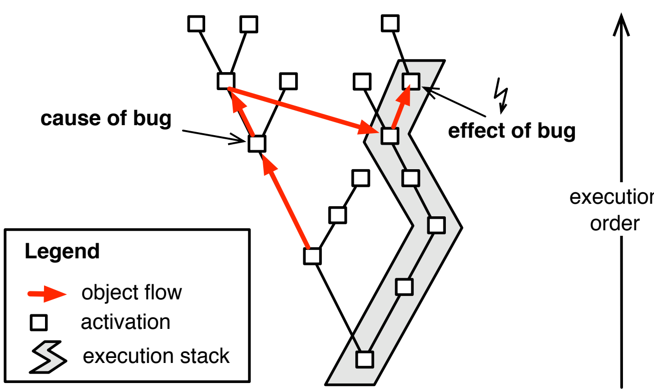


Figure 4. Example of how the cause of a bug can be outside the current stack.

By means of fields the flow of an object can bridge the linear sequence of method executions. With red we illustrate the flow of an object relative to the same program execution. While the object is first passed along with the execution trace, its path later diverges and jumps to distant branches of the tree.

This example illustrates how changes to the software system which modify the behavior of objects may have unexpected effects at distant locations in the program execution. Therefore, to connect the cause and the effect of errors we need to trace the flow of objects. This will support the developer in finding errors by allowing him to follow incorrectly behaving objects back along their path. We call this approach object-centric debugging.

4 Implementation

We have implemented the model extractor in Squeak, a Smalltalk dialect. Since instrumenting method activation and field access would not allow us to trace the flow of objects precisely enough, we also keep track of aliases at runtime. That is, during the program execution we actually instantiate for each reference an alias object. The alias objects then act as proxies which trap message sends and forward them to the object.

The instrumentation of the target program happens in two phases. In the first phase the relevant part of the class hierarchy is replicated to facilitate scoping the effects of the instrumentation (*i.e.*, the classes are copied and the class hierarchy is reconstructed). This is necessary because we also want to instrument core libraries such as the class *Array* which is used by other parts of the system.

In the second phase the classes are instrumented by annotating the abstract syntax tree (AST) of the methods. This means that our approach does not rely on source code modification but rather operates on a more abstract level. The instrumentations modify for example field assignment. In this case the assignment is replaced with bytecode which instantiates a new field alias, sets its ancestor (and predecessor reference if appropriate) and eventually stores it in the actual field.

The performance overhead of the current prototype implementation is around a factor of 10. However, we have not yet done any performance optimization, and we expect to improve the performance in the future. We plan to push aliases down one level into the VM. The responsibility of aliases (capturing a specific reference to an object) can be implemented at this level much more efficiently. Instead of instantiating new alias objects, the indirection can be achieved by using a table which maps object pointers.

5 Related work

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [2, 6]. Many techniques focus on analyzing the program as a sequence of method executions [11, 20].

To better understand object-oriented program behavior various approaches have extended method traces. As an example, Gschwind *et al.* illustrate object interactions taking arguments into account [7] and De Pauw *et al.* exploit visualization techniques to present instance creation events [3].

These approaches extend method traces with some additional information. In contrast, our approach is much more radical as it proposes a new model which is centered around objects, capturing object aliasing, a key characteristic of object-orientation.

Most approaches of dynamic analysis in the context of object-orientation primarily analyze the program's execution *behavior* rather than the *structure* of its object relationships. An exception is Super-Jinsight, which visualizes object reference patterns to detect memory leaks [4], and the visualizations of ownership-trees proposed by Hill *et al.* to show the encapsulation structure of objects [8].

Those two approaches are based on snapshots whereas our model has an explicit notion of the *evolution* of objects: on one hand the *object flow*, and on the other the *object history*. Another practical advantage and difference to the two approaches mentioned above is that we do not only show how objects are referenced and how references evolve, but that we also combine this information with method traces.

Backward-in-time debuggers [9, 13] allow one to navigate back in the history program execution. Those debuggers capture the full execution and object history and like this allow the user to inspect any intermediate state of the program. Although some navigation facilities are provided,

the notion of how objects flow through the system is missing because the event-based tracing approaches do not provide complete information about the flows.

6 Conclusions

Dynamic information contains valuable information about how the systems works at runtime. Most of the approaches to analyze dynamic information use a trace-based view. However, in the case of object-oriented programs, the trace needs to be complemented with a view of how objects are referenced and passed around in the system.

In this paper, we present a novel approach in which we capture object references and explicitly model them as first class entities (*i.e.*, aliases). In our model we distinguish between several types of aliases and we build a meta-model that incorporates static information, trace information, and object flow information.

We have chosen to build our prototype implementation in Squeak because of the flexible nature of Squeak's environment. Until now, we have performed several case studies to test the scalability. Even though we witness a factor of 10 of slowdown, we are optimistic to improve the performance by adding support for aliases to the VM.

We foresee that this model opens new research tracks both from a reverse engineering perspective and from a forward engineering perspective. We have listed three examples of our work in progress, namely: (1) relating object flow to static structure, (2) characterizing objects based on the objects flows, and (3) object-centric debugging.

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 of ICSM 2005 (21th International Conference on Software Maintenance 2005*. IEEE Computer Society Press, Sept. 2005.

[2] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[3] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[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] M. Ducassé. Coca: An automated debugger for C. In *International Conference on Software Engineering*, pages 154–168, 1999.

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

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

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

[9] C. Hofer, M. Denker, and S. Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, 2006.

[10] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[11] M. F. Kleyn and P. C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, Nov. 1988.

[12] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[13] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Oct. 2003.

[14] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, 2006. To appear.

[15] H. Ritch and H. M. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of WCRE '93*, pages 192–201. IEEE, May 1993.

[16] Squeak home page. http://www.squeak.org/.

[17] T. Systä. Understanding the behavior of Java programs. In *Proceedings WCRE '00, (International Working Conference in Reverse Engineering)*, pages 214–223. IEEE Computer Society Press, Nov. 2000.

[18] D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

[19] N. Wilde and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[20] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society Press, 2005.