5th International Workshop on

Program Comprehension through Dynamic Analysis

colocated with the 17th International Working Conference on Reverse Engineering (WCRE’10)

October 14th, 2010 – Beverly, Massachusetts, USA

Technical report TUD-SERG-2010-039
Software Engineering Research Group
Delft University of Technology
The Netherlands
Contents

Reducing the Performance Overhead of Dynamic Analysis through Custom-made Agents
Tobias Gutzmann and Welf Löwe

An XPath-based Query Language for Trace Analysis
Oleksandr Panchenko, Alexander Koglin, Johannes Bohnet and Alexander Zeier

Can we use network analysis methods to discover functionally important method calls in software systems by considering dynamic analysis data?
Anjan Pakhira and Peter Andras

Exploiting Dynamic Information in IDEs Eases Software Maintenance
David Röhlisberger
Program Chairs

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

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

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

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

Program Committee

Constantinos Constantinides
Concordia University, Canada

Simon Denier
INRIA Lille - Nord Europe, France

Abdelouahed Gherbi
École de technologie supérieure (ETS), Canada

Yann-Gaël Guéhéneuc
École Polytechnique de Montréal, Canada

Adrian Kuhn
University of Bern, Switzerland

Adrian Lienhard
Netstyle, Switzerland

David Lo
School of Information Systems, Singapore Management University

Oscar Nierstrasz
University of Bern, Switzerland

Tamar Richner
Mobiliar, Switzerland

Wim De Pauw
Thomas J. Watson Research Center, IBM, USA

Heidar Pirzadeh
Concordia University, Canada
Reducing the Performance Overhead of Dynamic Analysis through Custom-made Agents

Tobias Gutzmann and Welf Löwe
School of Computer Science, Physics and Mathematics
Linnaeus University, Växjö, Sweden
Email: (tobias.gutzmann|welf.lowe)@lnu.se

Abstract

The usefulness of dynamic analysis tools for program comprehension increases with the amount of time a given analyzed program is monitored. Thus, monitoring the analyzed program in a production environment should give the best picture of how the program actually works. However, high performance overhead is especially undesirable in such a setting, because it decreases the user’s productivity. Performance overhead occurs due to recognizing events that are of interest to the agent monitoring the program run and storing those events in data-structures.

We propose to address this issue by creating a custom-made agent for each program. By using static analysis to get a priori knowledge about which events of interest can occur and where they can occur, tailored code for recognizing and storing those events can be generated for each program.

We evaluate our idea by comparing a “general purpose” dynamic agent that captures fine-grained points-to information with custom-made agents that do the same job for specific programs. The latter show highly reduced performance overhead in practice. We also investigate how the precision of the static analysis affects the performance of the custom-made agents.

1. Introduction

Dynamic program comprehension tools build up on data collected through inspecting running programs. The longer programs are monitored, the more data is collected. Therefore, it is straightforward to have the tools running in production environments. However, then the performance overhead must be reduced as much as possible, as users would not accept the decreased usability and productivity of the users would also suffer. Mock [1] stated that the acceptance of dynamic analysis tools is bound to their performance.

To decrease the performance overhead of dynamic analysis tools, we propose creating a custom-made agent for each program of interest. With results obtained from static analysis, a priori knowledge about which events of interest can possibly occur and where they can occur is available. This knowledge can be built into the program, which can then run more efficiently by using custom-made instrumentation code.

The remainder of this paper is as follows: In Section 2, we discuss the motivation for us writing this paper and present the dynamic analysis task that we need to speed up. We will use our task as a running example for the remainder of this paper. Then, in Section 3, we discuss our approach. In Section 4, we compare the performance overhead of a “general purpose” dynamic agent with agents that are custom-made for specific programs. Related work is discussed in Section 5, followed by future work in Section 6. Finally, we conclude this paper in Section 7.

2. Motivation and Background

The motivation for this paper comes from a slightly different direction than program comprehension, but it also benefits from fast dynamic analysis. Our current research aims at creating a Gold Standard for points-to information, i.e., a number of programs annotated with exact points-to information. Points-to analysis\(^1\) is a static analysis that computes reference information, e.g., what methods may be targeted by a given (polymorphic) call instruction, what reference values may be stored in a given field, etc. A Gold Standard is a means to create benchmarks for a given research discipline. The lack of such benchmarks makes it, in turn, hard to further develop a field by adopting the successful and avoiding the less promising approaches, as observed by Sim et al. [3].

\(^1\) A very good introduction is given, e.g., by Hind [2].
Unfortunately, a Gold Standard for points-to analysis is impossible to compute automatically, as the problem is intractable. However, any static points-to analysis – given that it is conservative – computes an upper bound of a Gold Standard. Results obtained by dynamic analysis form a lower bound of a Gold Standard. Bringing both closer together means better approximating the Gold Standard. The motivation of having a fast running dynamic analysis is then to have more people help in collecting data for the lower bound of the Gold Standard.

2.1. Analysis Abstractions

Any static program analysis needs to abstract from the values which expressions may take during a real program run, as it is impossible to model the exact program state at any time of any possible run of a program.

An abstract object \( o \in O \) is an analysis abstraction that represents a set of runtime objects. The mapping from runtime to abstract objects is called a naming scheme. In this paper, we use a naming scheme where each syntactic creation point \( s \) corresponds to a unique abstract object \( o_s \). Variables (including, e.g., method arguments) and expressions are associated with analysis values in \( 2^O \), the power set of \( O \). For interoperability between different static and dynamic analysis implementations, we distinguish abstract objects by the type, the syntactically enclosing class, and the line number where the object is created.

2.2. Assessing Points-to Analysis

The results of points-to analysis are usually very low-level, e.g., for each variable access, a points-to set of abstract objects possibly bound to this variable access is computed. Since such information has little value in itself, we use client analyses as defined below for accuracy assessment. For sake of simplicity, these are rather simple and easy-to-derive. Some of these are low-level and should thus be capable of distinguishing small differences in analysis accuracy.

For a given program \( P \), let \( M \) be the set of its methods and \( O \) the set of all syntactic creation points in \( P \). The exact object call graph then consists of nodes and edges \([o_i, m_j] \rightarrow [o_k, m_l] \in O \times M \times O \times M\) where there exists a concrete execution of \( P \) such that (1) \( m_j \) is called on an instance object \( [i_w] \), (2) during the execution of \( i_w \), \( m_j \) occurs a call to \( m_l \) on instance object \( i_w \), and (3) \( i_w \) were created at the syntactic creation points \( o_i \) and \( o_k \), respectively. The exact call graph is the projection of the exact object call graph so that nodes and edges \([o_i, m_j] \rightarrow [o_k, m_l] \) are projected to nodes and edges \( m_j \rightarrow m_l \).

For a given program \( P \), let \( F \) be the set of all fields and \( O \) be the set of all syntactic creation points in \( P \). The exact abstract heap then consists of all the relations \([o_i, f_s] \leftarrow o_j \in O \times F \times O\) where there exists a concrete execution of \( P \) so that (1) an instance object \( i_w \) is stored into the field \( f_s \) of an instance object \( i_v \), and (2) \( i_v, i_w \) were created at the syntactic creation points \( o_i \) and \( o_j \), respectively.

The exact object call graph, exact call graph, and exact abstract heap can be approximated by both dynamic and static analysis. We then speak of the client analyses object call graph, call graph, and abstract heap, respectively.

2.3. The Dynamic Agent

A dynamic agent for capturing points-to information performs two tasks: First, it maps from runtime to abstract objects. Second, it captures which object call graph edges and field writes occur during a program run, but not how often they occur.

Our general purpose dynamic agent instruments code using the ASM framework\(^2\). Method calls are instrumented so that each existing call site and each store operation to a reference field additionally call the monitoring code. There, the occurring event is stored into a general data-structure. This data-structure is currently a simple HashMap that contains all events that have been observed so far; if an event occurs that is identical to one of these events, it is not added to the HashMap again and thus is not retained in memory, but the HashMap access still causes runtime overhead. It is obviously arguable that this is not the best suited data-structure, but experiments with other data-structures (e.g., trees) have not yielded better performance.

Finally, on program termination, the events that are stored in the HashMap are written to a file.

3. Approach

A dynamic analysis tool for program comprehension usually captures the occurrence (or amount of occurrences) of events, e.g., invocations of methods, thrown runtime exceptions, etc. A priori, it is not known which events may possibly occur, which makes it necessary to have a general data-structure that is able to capture any event. This data-structure must be dynamically expanded to be able to capture events that happen. However, if the events possibly occurring are

\(^2\) http://asm.ow2.org
known to be limited prior to executing the program, the data-structure can be statically optimized for the given task and accessed with higher performance. The general approach that we suggest works as depicted in Figure 1: First, an upper bound for the possibly occurring events is computed with help of a static analysis tool. Then, those results are used to instrument the original code to store events of interest. Finally, the modified program is deployed instead of the original one.

How the instrumentation code is supposed to look like is dependent on the purpose of the dynamic agent and is thus hard to be generalized. We describe the solution for our own purpose, which we believe can be adopted in similar ways for other purposes.

Our tool instruments each object creation site so that all objects created at this program point get a special tag value which is a unique integer value. This corresponds to the creation site naming scheme described in Section 2. This value can be accessed via a special getTag() method that simply returns this value. This id-number can be mapped back to the class and line number where the object is created.

Then, our tool reads the statically computed possible events (object call-graph edges and heap writes) of relevance to our analysis. A helper class containing byte arrays for storing which events have occurred during the run is created. There are two arrays, one for heap events, one for object call-graph events. Whenever a store operation to a reference field occurs, or a method invocation occurs, a byte in this array is set to value “1” to indicate that the event has occurred. Which event number (= index in the byte array) must be set depends, besides the syntactical location of the occurring call, on two variables, namely the this-values of the source and target methods. This mapping at runtime should be as efficient as possible – desirable would be a simple arithmetic function expressed in terms of the two abstract objects (represented as integer values) involved. Since finding such an explicit function is difficult – if not impossible – we encode the

3. Calls to and from static methods are just simpler cases, where one or both of the variables is fixed.

![Figure 1. Process](image)

```java
void foo(A a) {
    // ///// start instrumentation /////
    switch (this.getTag()) {
        case 0:
            switch (a.getTag()) {
                case 13: DynamicResult.ocg[28] = 1; break;
                case 28: DynamicResult.ocg[14] = 1; break;
                // ...
                default: handleDefault();
            }
            break;
        case 7:
            switch (a.getTag()) {
                // ...
                default: handleDefault();
            }
            // ///// end instrumentation /////
    }
}
```

![Figure 2. Instrumentation: Nested Switch-statements](image)

function into the control flow of the program. Consider the program in Figure 2. It contains a single method foo(), which, in turn, contains just one call to another method bar(). This call is instrumented as follows: Depending on the tag of both the caller object and the callee object (this and a, respectively), an index in the storage array (DynamicResult.ocg) is selected and set to value 1, indicating that this call has occurred at some time during the program run. The numbers in the case-statements are arbitrarily selected; they represent the general idea that there is no straightforward mathematical connection between the abstract objects reaching this call and the index selected in the storage array. If no matching pairs for the two tags are found, i.e., one of the switch-statements goes into the “default” branch, then this needs special handling. In our case, this should never happen and indicates an error in the static analysis.

Finally, once the program execution terminates (or optionally, periodically every certain amount of time), the byte arrays are traversed and events that have occurred are written to a file. Since identical events are not written more than once, this is done without much runtime overhead.

The more imprecise the static analysis, the more case-statements are required in the instrumented code. This increases the code size and decreases the performance. We will investigate the effect of the precision of static analysis on our approach in Section 4.

### 3.1. Further Issues

In case of analyzing multi-threaded programs, synchronization might be required to access the common data-structures, e.g., when increasing an event counter.
Optionally, the data-structures could be cloned for each thread in order to reduce synchronization, at the expense of higher memory cost and thread creation time. This approach is successfully applied by Binder et al. [4]. In our concrete case, where we are interested in what happens, but not how often it happens, synchronization is not necessary as our event-capturing code simply writes a “1” into an array. Race conditions thus do not matter in our special case.

An important question that must be answered is: Does an agent require to instrument library code? Especially tools for program understanding may not be interested in what is going on in, for example, the Java runtime library. For instance, users may feel that no insight into a program is gained by looking at how java.util.HashMap works internally. So, the instrumentation can be limited to those classes of interest. This also solves some practical issues: Classes that are loaded at virtual machine startup do not need to be instrumented (either prior to execution or at execution), and classes like java.lang.String and java.lang.Thread are specially treated by the virtual machine and thus can be critical to instrument. Further, less instrumented code means less runtime overhead.

3.2. Limitations

Our approach suffers from the general limitations that any pure instrumentation-based approach suffers from: First, native code cannot be analyzed. That is, objects created in native code will not be tagged, and thus the results of the dynamic analysis will be incomplete. However, Binder et al. point out that programs usually spend only little time in native code [5]. Second, the size of a method’s bytecode is restricted by the Java Virtual Machine Specification [6]. If now, through instrumentation, the bytecode exceeds this limit, the program will not work any longer. However, we agree with Binder et al. who state that “this problem is very unlikely to occur with normal, hand-crafted Java code” [4].

4. Evaluation

Our experimental setup is as follows: We run both the “general purpose” agent (described in Section 2.3) as well as custom-made agents on a set of six benchmark programs. We chose programs that we used in earlier studies [7] and for which our points-to analysis implementation is conservative. For each of the benchmark programs, we created two custom-made agents: One created with the results of points-to analysis, the other with “exact” results obtained by the general purpose agent. The latter allows us to evaluate the minimal, intrinsic overhead of our approach, and we can make conclusions about how important the precision of the underlying points-to analysis is.

All experiments were run on a standard laptop computer, a Dell Latitude E6500, 4GB RAM, 2x2.4GHz (Intel P8600 processor). We run the Java Virtual Machine (JVM, version 1.6.0_20) in both “client” and “server” mode (via the “-server” option); the difference between the two is that the latter optimizes more aggressively, which leads to a larger compiling overhead but possibly better optimized code.

As entry points to the programs, we have created special driver-classes that invoke the different programs’ main-methods several times with different input. We do this instead of running the program over and over again with different input because we want to evaluate the performance overhead on long-running programs; for short running programs, it would not really matter if a program runs, e.g., one or three seconds. All timing results are the average over three runs. In final, we have eight result sets for each of the benchmark programs.

4.1. Performance overhead

Figure 3 shows the relative execution times of all setups as a factor to the execution time of the unmonitored “client” setup (which is thus omitted in the diagram).

The average performance overhead for the six benchmark programs that we investigated is reduced from factor 3.57 (general purpose agent, “client” mode; server mode: 3.30) to 2.62 and 1.94, respectively.

In all cases, the custom-made agents are faster than the general purpose agent. With the exception of javacc, all the agents greatly benefit by the enhanced optimization capabilities of the virtual machine in “server” mode. While the average execution time of unmonitored programs is, in our setting, slightly worse than in the client virtual machine (factor 1.14), the general purpose agent (factor 3.57 vs. 3.30) benefits slightly, the custom made agents (factor 2.62 vs. 1.94) even more.

Looking at extremes, javacc (factor 2.89) and bloat (2.44) suffer the biggest slowdown with custom-made agents. Those two programs are very method-invocation intensive. jlayer experiences hardly any overhead. This program, an MPEG audio to WAV converter, contains a lot of numerical computations, which dominate the execution time of the program. Our agent does not instrument those computations.
5. Related Work

Many dynamic analysis tools have been presented in literature; discussing them with respect to applicability of our approach is beyond the scope of this paper.

We are aware of a few similar attempts to reduce the performance overhead of dynamic analysis by use of static analysis. Suan Hsi and Horwitz [8] reduce the overhead of a runtime type checker for C programs by using static analysis. However, despite reducing the overhead by 40%, their instrumented code still runs 23 times slower than the original code. Their work is closely related to ours, but the difference in supported programming language – C vs. Java – as well area of interest – bug-checking vs. program comprehension – dictate different requirements and challenges. For instance, their approach is to find code points where instrumentation can be removed, not how storing observed information can be made efficient. Ostrand et al. use static analysis to find the places in a program that are most likely to produce errors, and on which then dynamic testing efforts should be focused [9]. Again, their idea is where to put instrumentation code, not how to decrease the performance overhead caused by the instrumentation code.

Different approaches for efficiently collecting dynamic data have been presented in literature. For example, Binder et al. [4] collect profiling information using so-called method call trees. Their approach shows slowdown factors of less than 10 for a number of benchmark programs, with an average slowdown factor of less than 5. Pothier et al. [10] collect complete execution traces, including field writes, method invocations, method return values, etc. They report a performance overhead of 10 and 100 for a realistic and a worst-case scenario, respectively. In general, comparing the performance overheads of different approaches is not trivial because benchmark setups and analysis tasks differ.

6. Future Work

As discussed in Section 3.2, the maximum size of a method’s code is limited. Thus, we do not inline switch-statements into the code as outlined in Figure 2, but create separate methods where each method contains exactly one switch-statement. However, in cases where it is possible to inline the code (i.e., the maximum method code size would not be exceeded), performance could be improved by inlining.

Sometimes there are cases where the static analysis does not compute conservative results, i.e., the results are not an over-approximation of a Gold Standard.
In our case, this can be the case when the points-to analysis does not support features like dynamic class loading, or if the whole program is not at hand at time of creating the custom-made agent, e.g., when the system supports a plugin system. Then, a fallback to a “general purpose” handling, i.e., using a “general purpose” data-structure, should be taken in the handling of the “default”-branches in the switch-statements.

Switch-statements can be compiled into two different bytecode instructions: lookupswitch and tableswitch (see chapter 7.10 in [6] for details). The former is a general case that is interpreted as a sequence of if-then-else statements. The latter is used if the value range of the case-statements is dense, i.e., there is an efficient representation of the values as indices into a table of target offsets. Our current prototypical bytecode instrumentation implementation uses only lookupswitch-instructions, but never tableswitch-instructions, which may be more efficient in many cases. A step further would be to encourage the use of tableswitch-instructions by sorting the result array accordingly, i.e., by ensuring that events that can occur at the same syntactical location as well as abstract object that may be used there get tags close to each other.

If a given event occurs for sure, i.e., it has been observed before, it is no longer necessary to look out for it. Thus, its instrumentation code can be removed (but it must be made sure that no “general purpose” handling takes over then), either by re-creating the custom made agent, or even at runtime by dynamically removing the instrumentation code. This requires that the agent is not interested in the number of its occurrences, as is the case in our example.

When developing dynamic analysis tools, writing them as generators for custom-made agents is probably not on the top of the list of the developer. It is more likely that such tools get developed as general purpose tools first. Further, for creating the custom-made agents, a static analysis tool for the same purpose is required. If now these two tools are at hand, a third tool, the generator for custom-made agents, must be developed additionally. Future work should look at which requirements are necessary that this generator can be generated by the specification of the static analysis results and the general purpose tool. Methods from model-driven software development could be applicable if the general purpose agent is developed with this in mind.

7. Conclusions

Building custom-made agents for dynamic program analysis can greatly reduce the performance overhead. Our current custom-made agents show much better performance than a general purpose dynamic agent that does the same job – this despite the fact that our implementation still has room for improvement, as discussed in Section 6.

Noteworthy is that using the “server” mode of the Java Virtual Machine reduces the performance overhead for the instrumented code in many cases: five out of six of our benchmark programs ran faster with the server virtual machine when using any of the agents.

References

An XPath-based Query Language for Trace Analysis

Oleksandr Panchenko, Alexander Koglin, Johannes Bohnet, Alexander Zeier
Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany
panchenko@hpi.uni-potsdam.de

Abstract

Debugging and analyzing software is a major development task. One possibility of analyzing runtime behavior of a program is trace analysis. However, trace analysis is a tedious process because of the vast amount of recorded data. One way to cope with this scalability issue is applying a query language to locate task-relevant parts within the trace. This paper proposes an expressive and efficient way of searching within traces. A descriptive XPath-based query language is used for fast, precise, and standardized querying of traces. With it, complex query conditions can be specified to answer questions, such as “how long did certain function call take on average?” or “which parameters were passed between a caller-callee pair?”. This paper illustrates the feasibility of the proposed approach by providing an implementation and applying it to a trace generated by executing a 200kLOC C++ software system. This trace contains 3.4 Mio function calls.

1 Introduction

Maintenance represents a core challenge and major cost factor within a software life cycle. Tools that facilitate maintenance by supporting developers in revealing the structure and the behavior of a system can significantly reduce costs. Trace analysis represents one approach to understand runtime behavior. It is applicable to a wide range of maintenance tasks, e.g., debugging, performance optimization, and feature location [5]. However, trace analysis is difficult because of the vast amount of runtime data that is typically captured. Hence, efficient trace exploration techniques are required. One class of trace exploration approaches relies on trace visualization. These techniques support developers in performing top-down or bottom-up exploration strategies (or both). Another class of approaches enables developers to formulate precise questions on trace data, which return a list of results that serve as entry points for further (detailed) trace analysis. Such techniques are referred to as trace searching techniques throughout this paper.

In this paper, we propose a query language for traces that enables developers to effectively search within traces. The term execution trace (or trace) is defined throughout this paper as a sequence of nested function calls that is annotated with information of variable states. A trace can be represented as a call tree. After trace data is collected by an existing tool, it can be queried using a query language. The proposed query language enables developers to formulate questions as patterns that are applied on a call tree.

As call trees can be stored in a relational or XML database, they can be queried by using the native databases’ query languages. However, those query languages are typically too generic and verbose for being effectively used. Moreover, relational data models for representing traces are relatively complex. Since XPath is one of the standard query languages for tree-shaped data, it can be used to formulate patterns applicable to call trees. The query language proposed here extends XPath and is optimized for trace analysis.

The key idea behind the proposed concept is to translate XPath queries to SQL queries. The SQL queries are then run on the relational database where the traces are stored. The advantages include: (1) the declarative nature of XPath hides complex relational data models from the developer; (2) XPath expressions are significantly more compact than the respective SQL-queries; (3) XPath is a standardized query language, it can be easily adopted by developers and tool vendors; (4) the proposed approach is a fast and precise way to locate relevant parts of large traces.

To prove the feasibility of the approach, we apply our implementation of the approach on traces generated from a C++ software system. Questions such as “when and from where was certain method called?” and more complex questions can effectively be answered.
2 Searching within Traces

The underlying data model of traces that is used throughout this paper comprises the nested structure of function calls in addition with parameter values forming a call tree. Each function can have several function calls at runtime. For illustration, Figure 1 depicts parts of the call tree contained in a trace recorded by executing the 200kLOC software system Audacity\textsuperscript{1}—the figure shows a screenshot of the trace analysis tool Software Diagnostics Developer Edition\textsuperscript{2}, which provides multiple synchronized views on traces. The Audacity trace that was used for our experiments in this paper contains 3.4 Mio function calls.

![Figure 1. Various views on the call tree.](image)

Each node of the call tree represents a function call holding (a) the signature of the respective function, (b) the start time, (c) the end time, (d) the function’s parameters names and their types, and (e) the values of the parameters. This data enables developers to obtain answers on a variety of maintenance-related questions. Searching within a trace can be used, for instance, to find frequently used function calls, look up indirectly called function calls, or check that functions are executed within given time or value bounds. The parameter values can be exploited to constrain the search result to certain parameter values and parameter types.

Simple keyword search would be a way to identify parts within a trace relevant to a maintenance task at hand. However, questions and issues posed by developers are typically more versatile and involve combinations of conditions and complex patterns. Thus, there is a need for a more expressive way to search within traces. An example would be: “find all functions with an execution time >50ms, called by the function DrawTracks(tracklist) with their one parameter value being null”. An answer on this question provides the developer with concrete entry points for further trace analysis—performed with a visual trace analysis tool, for instance. Further analysis questions can include (partially based on the catalog of questions by Sillito et al. \cite{9}):

- When and from where was this function called?
- How was control getting to this function?
- What were the values of parameters at runtime?
- How long did this function call take?
- Which functions took the longest time frames?

3 Related Work

The reverse engineering community has proposed various graph-based approaches to model and query facts on software systems. Holt \cite{4} discusses a whole family of languages for querying and manipulating a high-level architectural model of a system that is represented as a typed graph. Each vertex of these graphs can have a number of attributes which carry additional information. In our formalization, we use trees with vertices annotated with attributes.

In principle, there are two ways of working with traces: offline analysis (capturing events to a trace file or a database and analyzing them after this) and online analysis. We use offline analysis because in our scenarios the developer does not exactly know which parts of the program are relevant and does not want to formulate the queries upfront. In this case, it is useful to collect traces first and then formulate queries over the trace data. Moreover, trace analysis is an explorative and, therefore, iterative process. Developers refine findings and formulate subsequent queries; since online analysis may require a new run for each new query, it may slow down analysis.

A group of approaches for offline trace exploration are based on visualization techniques. In contrast to these approaches, we propose a search technique. In the following, we outline related work focusing on search techniques for trace exploration.

Although the program trace query language (PTQL) \cite{2} belongs to online analysis, we mention it here because PTQL is used for answering questions of the same kind. PTQL is an SQL-like query language over program traces, which relies on a relational data model. One of the advantages of PTQL is the relatively small amount of trace data because only the relevant part of the trace is captured. However, developers are...
forced to learn the data model and tediously formulate corresponding join clauses. Our approach seems to be easier to use because of its declarative nature.

Anslow et al. [1] use XQuery to extract information from program traces for visualization purposes. Their infrastructure can be used for searching, too. However, a problem with techniques based on XML databases is that the process of inserting documents is known to be slow. One of our goals is to enable developers to search interactively and to issue ad-hoc queries. In such an explorative setting, it is important to keep the cycle of trace generation and exploration short. Thus, our approach can be applied directly after the raw trace data has been inserted into a relational database, which is much faster than in the case of XML databases. Anslow et al. propose two XML-based formats for storing static information and runtime information, respectively. Although we also distinguish between static and runtime information, our approach stores trace data as is, the transformation from a higher level description into the raw trace format is done at query runtime.

A similar approach was proposed by Pietschker and Ulrich [7]. They proposed to use a set of predefined XSLT stylesheets to filter the parts of traces stored as XML files and visualize the results properly. Although they use XML technology too, their approach differs from ours in two aspects. Firstly, the XSLT stylesheets are predefined and developers can apply them just as a filter. It is useful in a number of scenarios where developers perform standard tasks, e.g., performance analysis. In contrast, our approach can be used in interactive scenarios. Secondly, they transform traces into XML files and apply XSLT stylesheets to these instead of using a database. Although they do not provide any performance measurements, we assume that our approach is faster.

Pothier et al. [8] used a high-speed database backend to store traces and enable backward navigation while debugging. Although their approach is scalable because of parallelization and indexing, the proposed queries are rather simple and aim at retrieving single events (stepping) or details to object state.

Program Query Language (PQL) [6] is an expressive language which is evaluated on a model of the dynamic program execution. Although PQL is quite powerful, it is evaluated based on an abstraction which hides some of runtime information, e.g. execution times.

### 4 XPath-based Trace Query Language

Since a call tree is an attributed tree, standard XPath can be used to formulate search patterns on it. Starting from the root node of a call tree, which is the context node describing the program’s main function, our XPath dialect allows traversing call trees using (1) the step’s child-axes with respective node tests for access to direct child functions via the parent ID, e.g., /DrawTracks; (2) the step’s self-or-descendant-axes with respective node tests to check for a contained function, e.g., //DrawTracks/DrawEverythingElse; (3) the step’s child-axes with attribute node tests to check for contained variable or parameter values, e.g., //DrawTracks/@tracklist; or (4) the predicates to check parameters, variables or time boundaries, e.g., //DrawTracks[@tracklist=\text{nil}] /DrawEverythingElse.

Besides the actual call tree, runtime data consists of information about functions, probes, files, etc. To access this information we extended standard XPath with a set of additional functions:

- `id(function)` returns vendor-specific ID (integer)
- `name(function)` fetches the function name of the call (return type string)
- `time(function)` is defined on a function name and returns its function call’s execution times (return type list of integers)
- `times(function)` takes a function name to check, how many times the function was called during execution time (return type integer)
- `vars(function)` and `params(function)` return a list of variable or parameter names for the function respectively (return type list of strings)
- `called(function)` checks, if at least one call has been made for the function (return type boolean)
- `file(function)` retrieves the file of the observed function (return type string)
- `value(param)` retrieves the probe of the selected parameter (return type string)
- `type(param)` returns the type of the parameter (return type string)
- `count(coll)` and `contains(coll, val)` do evaluations on value collections

All these implementation-independent XPath queries must be translated into the respective queries of the underlying database. A query translator parses the query and evaluates each XPath step, all the axes, node tests and predicate checks.

### 5 Feasibility Evaluation

For translation of XPath to SQL we implemented the Apache Xalan<sup>3</sup> XPathVisitor. Figure 2 shows the component architecture that we used to build our parser on top of the SQLite database provided by the

<sup>3</sup>http://xalan.apache.org/
trace generation and analysis tool Software Diagnostics Developer Edition.

![Figure 2. The architecture of our prototype.](image)

**5.1 Database Schema**

In our case, the aforementioned tree structures are described by an underlying relational data model, which is shown in Figure 3. We rely on this data model to describe an example mapping from XPath to SQL in Section 5.3. Later on, we abstract from this implementation of storage by introducing a mapping mechanism. Conceptually, the tool Software Diagnostics Developer Edition generates SQLite\(^4\) data based on a data model consisting of static code and dynamic runtime data. Static data contains functions and variable names, runtime data is linked to the static data via an m:1 relationship and contains concrete values of function executions and parameter values. Parameter values are identified by an id, carry their types and values and link to the function executions, they were used in.

![Figure 3. An excerpt of the used data model.](image)

**5.2 Optimization of Query Execution**

As can be seen in Figure 1, one way to represent trace data is as a directed tree. Therein, each node represents call of a function. In complex software systems, call trees may have deep hierarchies if they use recursion or use framework functions. Whereas the root node acts as the main function of the program instance, the leaves represent the actually executed functions. Each node contains its start time and its end time, which together make up a timeframe that always resides between the parent node’s timeframe, i.e., the root node has the earliest start time and the latest end time. Instead of recursively joining function executions relation when looking for descendants or ancestors of a current node, we use only one selfjoin and check if the timeframes are within the current node. Grust et al. \cite{3} propose efficient access to XML data stored in a relational database by making up a set-oriented index structure for path evaluation using B-trees and R-trees. We use a similar approach by utilizing time stamps.

**5.3 Translating XPath to SQL**

A query language for traces must be independent of an implementation-specific target scheme and system. As there is no established standard for storing trace data in databases, we aim at defining a descriptive and implementation-independent query language that is applicable to various data schemas. Therefore, a tool-dependent mapping strategy needs to be defined beforehand that translates queries to the concrete storage model.

This query mapping translates a descriptive tree-based query into queries for the specific storage systems and data schemes, which return results accurately. For our implementation and use case scenarios, we reach this goal by defining the syntax for XPath in such a way that it maps to SQL queries understood by SQLite.

The example question given in Section 2 is formulated as:

```xml
//*[time(.)>50 and name(../)='DrawTracks' and contains(params(../), 'null')]
```

This XPath query is automatically translated into an SQL query as follows:

```sql
SELECT E2.id
FROM functionExecutions E2 JOIN
functions F3 JOIN
functionExecutions E3 JOIN
dataProbes D3
WHERE E2.grossCosts > 50 AND
E3.parentId = E2.id AND
E3.callId = F3.id AND
F3.name = 'DrawTracks' AND
E3.id = D3.functionId AND
D3.variableValue = 'null';
```

\(^4\)http://www.SQLite.org
As seen in this example, the SQL query contains many implementation-specific joins, conditions, keys, and foreign keys. A developer analyzing a trace is likely not to be interested in these details. Hence, the declarative XPath-based query seems to be better suited for the purpose of searching within traces. The developer can concentrate on trace analysis and the maintenance task at hand and is freed from defining complex query patterns.

To support the developers writing their queries, we define generalized expression syntax for XPath:

\[
\text{XPath} = \left( \mathit{path} \right) * \left( \mathit{filter} \right) * \left( \mathit{declare-variable} \right) *
\]

Queries contain an arbitrary amount of cascaded function calls with defined variable names and values. Additionally, the XPath to SQL Query Translator supports functions listed in Section 4.

5.4 Estimation Cost Model

We use timestamps of the call entities to join nested function calls. It has a linear complexity. The additional XPath functions have different complexities:

- The \text{id()} function just modifies the selection of the SQL query and has constant complexity. The query \text{id}(/LWSlider/*) took 7.4 secs.
- The \text{file()} function joins two relations \text{functions} and \text{files} and thus has under linear complexity. The query \text{name}(/LWSlider/*) took 8.9 secs.
- The \text{called()} function selfjoins the \text{function executions} relation and thus has linear complexity. The query /LWSlider/*\text{[called(slide)]} took 14.8 secs.

The query translation costs are neglectable (\text{< 100 ms}). The measurements were done on a laptop with 2GHz Intel CPU and 2GB of RAM under Windows. The major point of concern is the formulation of the query. But as soon as XPath query is correctly formulated by a developer, the translation to SQL and the execution is accurate and robust.

6 Conclusion

Fast and precise query analyzers for trace exploration with an expressive and standardized query language are desirable tools for developers to understand runtime behavior while performing maintenance tasks. The query language XPath seems to be a good candidate as basis for building such query analyzers because XPath is capable of formulating patterns of trees and because XPath allows efficient translation of its queries to relational database formats—most trace analysis tools use common SQL databases to store trace data.

In this paper, we propose a descriptive XPath-based query language for trace analysis and exploration. We describe one of the possible ways of defining such a language and demonstrate that—though query translation is an additional processing step—the proposed approach is time-efficient compared to writing actual underlying SQL queries as XPath queries can benefit from automated translation rules to SQL. We demonstrate the feasibility of our approach by applying our prototypical implementation to traces gathered from a C++ software system. Due to the separation of the query language from the underlying trace storage format, the prototype can be modified to support data models of other tools. Future work includes the efficiency evaluation of the proposed language in terms of query formulation.

References

Can we use network analysis methods to discover functionally important method calls in software systems by considering dynamic analysis data?

Anjan Pakhira and Peter Andras
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
{anjan.pakhira, peter.andras}@ncl.ac.uk

Abstract

It is difficult to analyse large-scale integrated software systems with the purpose of improving their functionality through maintenance and evolution. Such systems contain many interactions between their components and can be represented as complex interaction networks similar to complex biological and socio-technical systems. Here we aim to check whether the combination of dynamic analysis and network analysis can determine method calls of high functional importance in a software system. We use as a test case the JHotDraw 6.01b software and predict the method calls with high functional importance using network analysis methods. We validate the predictions by disabling the methods predicted to have high functional importance and evaluating the behaviour of the software following this. Our results indicate that some of the considered network analysis methods are relatively good in predicting method calls of high functional importance.

1. Introduction

Large-scale software systems became ubiquitous parts of everyday life in the last decades – just consider the collection of software that integrates work done on mobile phone PDAs, office desktop at work, and laptop used at home and at conferences. While software components became more reliable in parallel with the rapid expansion of software systems due to better development support and improved software development and management practices, the large scale of current software systems presents new challenges for the maintenance of these systems [1]. Such software systems evolve by integrating new components with old ones and as a whole they are practically tested mostly by their users as the usage of the software expands. This makes formal analysis aimed to support software maintenance [2] difficult since usage patterns evolve as the software system is used by an increasing number of users and the blueprints of integrated components may not be available or compatible in the context of possible analysis or test scenarios[3].

Dynamic analysis, [4] [5] captures only interactions between components that actually happen during the running of the software. It offers a dynamic slice of the software that is characteristic of the software system in some usage context. Dynamic analysis is likely to be able to capture what is truly important in terms of software component interactions within the software system, given a range of typical usage scenarios. However, dealing with dynamic analysis data may present difficulties related to data complexity [6], visualization [7, 8] and practical interpretation [8]. As we noted above, dynamic analysis generates a description of the software system in terms of interactions between components (e.g. method calls between classes or objects). This description of the software system can be considered as graph or network with nodes being classes (or objects) and edges (arcs) being the method calls between these. Thus, network analysis [9], [10] may lend us some help in extraction of meaningful information of highly complex networks of software component interactions [11], [12]. Often, network analysis is used to determine functionally important components of the system represented by the network [13], [14]. In this context a component of the system being ‘functionally important’ means that this component of the system is critical for the delivery of the normal overall function of the system (this overall function is often the regular existence of the system, for example in case of an organism being alive and normally reproducing is this overall functionality). For example, in case of bacteria, network analysis is used to determine proteins that are functionally important (also called essential proteins) for the survival of the bacterium (i.e. if any of these proteins gets blocked or is not produced within the cell the bacterium dies or is unable to reproduce) [14]. In this case, the network of protein interactions is analysed. Functional importance of a protein is predicted on the basis of the structural...
importance of the node in the network that corresponds to the protein. The structural importance of nodes is evaluated using a range of network analysis methods. We define the functionality of the software system in the context of a usage scenario. The ‘normal’ or expected user experience in a usage scenario is the functionality of the software, where the user may be a human, a machine, or another software. We follow the biological analogy to define the concept of functional importance. A component (e.g. a class, a method) is functionally important in the context of a usage scenario if the functionality of the software is significantly altered if this component is not functioning normally (e.g. the method call is executed, but an execution branch is always avoided due to a revision that alters the test in an if-then-else branching). The software functionality is considered to be significantly altered, if the user experience in the usage scenario changes to the extent that renders the software dysfunctional from the perspective of the user. For example, if a software has a graphical user interface that does not allow the user to select menu elements from a menu bar renders the software dysfunctional from the perspective of the user in a usage scenario that includes the selection of menu elements from this menu bar. However, if the alteration of some software components means that instead of light blue background the menu bar uses light red background, the user experience changes, but it is not significantly altered in the context of this usage scenario, since the selection of menu elements from the menu bar works in the same way as it did before the alteration of the software.

Understanding the software by knowing the functionally important components (e.g. methods of classes) can help considerably the improvement of software functionality through re-engineering. These are the parts of the software that have a significant impact on the user experience. Consequently, altering them should be done with extra care to avoid negative impact on user experience and their alteration may also provide the key to improving the user experience. For example, one way to find such functionally important methods of classes is to look at bug reports and find the methods that are involved in generating the reported bugs. Alternatively, one may experimentally alter methods of classes and examine the user experience following the alteration. However, both approaches are very time consuming and resource hungry.

In, most cases of network analysis applied to software systems aim primarily to show that software as a network has certain network properties (e.g. being a small-world or a scale-free network)[11], [15], [16], but do not really give guidance about how to use network analysis to make software better. Dynamic analysis data represents the exact set of method calls within the system that are involved in the analysed usage scenario. However, not all of these method calls are necessarily functionally important for the executed usage scenario (according to our definition). Our aim is to find out to what extent can we use network analysis combined with dynamic analysis of software to find functionally important methods of classes. If network analysis of dynamic analysis data can provide reliable prediction of functionally important methods in object-oriented software that would significantly reduce the time and resource needs for finding the functionally important methods, for a defined functionality of the software (i.e. defined as user experience). This could help the quicker and better understanding of the software and support the re-engineering of the software with the aim of improving the user experience.

We report the first results of our evaluation of the combination of dynamic analysis with network analysis methods aimed to detect functionally important method calls in complex software. Our work shows that some network analysis methods are better than others in predicting the functional importance of network edges in the context of dynamic analysis of software. For the purpose of evaluation we chose to use the JHotDraw 6.01b software. The reported evaluation is focused on a single usage scenario of this software that has been considered by others [7].

The rest of the paper is structured as follows. First we discuss related work followed by software systems as networks and the relationship of this with static and dynamic analysis. Next we present our data and results. The paper is closed by the conclusions and future work section.

2. Related work

Recent heightened research interest in application of network analysis methods and measures in physical, biological, and social sciences has motivated similar work in the broad field of software engineering. Myers [12] discusses application of complex network analysis to static collaboration directed graphs of a selection of procedural software systems, and object oriented systems., He discusses correlations and anti-correlations of connectivity measures, and how these graphs are characterized by scale-free properties. This paper also briefly discusses how network analysis may be used for software refactoring. Myers [12] questions the analogy between the network analysis of software and biological systems, however we believe that such analogy works up to some level.
Software evolution and stability measurement has been discussed by Jenkins and S. R. Kirk [11]. In this paper software stability is studied through the lens of structural evolution through consecutive versions. The authors also propose metrics to measure this. The study uses complex network analysis, and finds how connectivity pattern evolves with software age.

Zimmermann and Nagappan [18] apply social network measures for defect prediction, this work and the following work by Tosun et al [19] validates the results. In both these cases the number of positive defect predictions improved by 10% over prediction based on traditional complexity metrics.

In most of these works, directed graphs are used to represent software networks. However, we use undirected graph for our analysis. The use of undirected graph was preferred as this simplifies the application of the network analysis methods and measures that we use, while not altering the meaning of the data that we use.

3 Software Systems as Complex Networks

Here we consider software systems developed in an object oriented language environment (e.g. Java, C++). Such software systems are built by defining classes forming class hierarchies through ancestor – descendent relationships. Class definitions imply instantiations of other classes as objects. The software system delivers its intended service by instantiating one or a few initial objects, which trigger the instantiation of many other objects. The interactions between objects are defined in the class specifications in form of method calls. Events are changes in the environment of the software system triggered either by the software itself or by other sources (e.g. the user, another computer, etc.). Objects may be notified of such events, which may imply the calling of their appropriate methods that handle the presence of these events. The dispatching of events is done by core or system objects that monitor the presence of such events (e.g. they monitor input from the mouse or keyboard). The distribution of event notifications in form of method calls and method calls in general constitute messages, which often also have parameters (i.e. the input variables of the called methods). This brief summary captures key concepts and elements of object oriented software. Of course, various realisations of object oriented development environments may have additions and variations in terms of actual implementation and usage of these concepts. Dynamic analysis considers only instantiated objects and the classes to which these belong together with the actual method calls that are effectuated during the run-time of the software systems [20],[21]. This gives a dynamic slice of the software system, which indicates the actual likelihoods of instantiating classes as objects and of calling given methods of these classes in the context of some usage scenario (e.g. typical everyday usage). The objects / classes and the method calls linking them can be seen as a network of objects / classes linked by edges (arcs) representing the called methods. Considering software systems as complex systems represented as complex networks (i.e. network of classes / objects and method calls resulting from dynamic analysis) [12], [15], [16], [17] means that we
can apply network analysis methods to search for functionally important components (classes, objects, method calls) of these systems. The core assumption of network analysis is that there is a good correlation between the functional importance of system components and the contribution of corresponding network components to the structural integrity of the network (i.e., functional and structural importance correlates). The latter can be measured using network integrity measures (e.g., average shortest path length) and the change in such integrity measures following the removal of the network element (e.g., node or edge) from the network. If network analysis works for software systems in this sense, it can lead to a better understanding of functionally important parts of the software, helping the evolution through functional improvements of the system. Here we present the network analysis of the JHotDraw 6.01b software. Our results indicate that some network analysis methods can indeed detect functionally important method calls in the analyzed software system. We also show that some of these methods are less effective than expected in terms of finding functionally important method calls.

4. Determination of vulnerabilities in software systems

4.1 Data collection

We chose the JHotDraw 6.01b (www.jhotdraw.org) software as our test bed software. This is software resulted originally from a design experiment and consequently it is considered a well designed software. The code has over 66K lines of code and includes 344 classes with a few thousand methods that can be called. Since this software has been subjected by others to dynamic analysis [7] it is a good starting point for the combined use of dynamic analysis and network analysis of software systems. To collect the dynamic analysis data we need to trace and log the interactions between objects / classes and the methods of which call instantiate these interactions. Dynamic analysis is practiced by many groups but there are relatively little details available about the actual techniques that are used to gather dynamic analysis data (see for example [7], [8], and [18]). The main technical options that we considered were as follows: (1) the use of the TPTP Probekit agent in Eclipse (www.eclipse.org/tptp/); (2) using the Java NetBeans profiler (netbeans.org); (3) aspect oriented implementation of crosscutting concerns for the detection of entry and exit of methods using AspectJ (eclipse.org/aspectj). We used as our dynamic analysis data generation method the TPTP Probekit agent including tracking the entry and exit of methods and the analysis of the stack trace. At the time points of entry and exit checking the Probekit agent writes into a log file tracking the execution of the program, and following the entry phase the agent also investigates the stack trace in order to determine the current class, the caller class, and the current class method that has been called by caller class. To start the evaluation of the joint application of dynamic analysis and network analysis we used the same single usage scenario that has been used by others [7]. The same scenario was run many times (>20) to make sure that we get reliable data. The operation sequence was the same each time: generate three drawing panels, place on each after being generated five drawing objects [7]. The data that we analysed included around 900,000 entries for each run. The entries that we analysed include the names of the caller class, the called class and the called method of the called class. We also used the Java NetBeans profiler approach to collect comparable data to check and compare with the results generated by the TPTP Probekit agent. For the purpose of visualization we used the Pajek graph visualization software (pajek.imfm.si).

4.2 Network analysis

First we processed the data to generate a network representation of it. We found 195 classes that were active during our sequence of operation. There were 817 methods of these classes that were called during the runs of the software. For the sake of simplicity, we ignored the direction of the calls and considered all method calls as undirected edges (and not as directed arcs). Note that two nodes representing classes may be connected by many edges representing different method calls between the two classes. It should be also noted that a method of a class may be called by more than one objects belonging to different classes, in such cases the method will be used to label all these interactions between classes, i.e. the same method label may appear attached to different edges. A network representation of the dynamic analysis data derived from the software is shown in Figure 1A. We analysed the connectedness distribution of the network nodes, i.e. the connectedness of a node is the number of edges connecting a node to other nodes. Considering all edges for all nodes representing classes the best fitting distribution of the connectedness values is log-linear (see Figure 1B) with the probability density function of the connectedness values:

\[ p(x = a) = \frac{-4.9 \ln(a) + 45.167}{a} \]  

(1)
While this is not a probability density function corresponding to a power law distribution (e.g. $p(x = a) = \alpha \cdot \frac{1}{a^\gamma}$, a typical connectedness distribution of a scale-free network has a $\gamma$ value in the range of 2 – 4) this distribution also provides a much longer tail than an exponential distribution (e.g. $p(x = a) = \lambda e^{-\lambda x}$) – i.e. having very highly connected nodes is relatively likely – which makes meaningful the application of network analysis methods for the determination of components that have high importance for the structural integrity of the network. We used network analysis methods to determine important edges of the network that are likely to contribute significantly to the network's structural integrity. Such edges are likely to represent method calls that are functional important for the software system. We used three network analysis methods to calculate such importance values of edges and their corresponding methods:

1. we calculated the hub connection score (HCS) of edges as the sum of the frequencies of calls of a method across all recorded calls of the method by all classes – $f(e \mid n)$ is the frequency of the calls of the method $m$ represented by the edge $e$, originating from object instances of the class represented by node $n$:

$$HCS(m) = \sum_n f(e \mid n)$$  \hspace{1cm} (2)

2. the weighted connection score (WCS) of a method $m$ represented by edges $e$ that connect nodes $n$ to the node representing the class of the method $m$, with connectedness values $v(n)$, is calculated as:

$$WCS(m) = \sum_n f(e \mid n) \cdot v(n)$$  \hspace{1cm} (3)

3. considering the call frequency of the method corresponding to the edge, $f(e \mid n)$, as:

$$WCS(m) = \sum_n f(e \mid n) \cdot v(n)$$  \hspace{1cm} (3)

4. the betweenness score (BWS) of a method $m$ represented by edges $e$ is the maximum betweenness score of these edges, the edge betweenness score being the number of shortest paths connecting nodes of the network that contain the edge; there may be more than one alternative shortest paths between two nodes; the length of an edge was set to be the inverse of the call frequency of the method represented by the edge:

$$BWS(e) = \left[ \left\{ e_1, \ldots, e_k \right\} \mid e \in \left\{ e_1, \ldots, e_k \right\}, \frac{1}{\sum_{j=1}^{k} f(e_j)} \leq \frac{1}{\sum_{j=1}^{k} f(e_j')}, \forall \{e', \ldots, e_k'\} \right]$$  \hspace{1cm} (4)

Where, $nodes(e)$ determines the two nodes that are connected by the edge $e$.

All methods were ranked according to these edge importance metrics and we considered the highest ranked methods as the ones that are likely to have the highest contribution to the structural integrity of the network, according to the considered importance metric. Consequently, the prediction according to the assumptions of network analysis is that these methods are likely to be functionally important for the software.

### TABLE I. THE TOP-5 RANKED METHODS ACCORDING TO THE THREE NETWORK ANALYSIS BASED SCORES AND THE CALL FREQUENCIES. ALL METHOD NAMES START WITH 'ORG.HOTDRAW'.

<table>
<thead>
<tr>
<th>Method</th>
<th>Rank</th>
<th>HCS scoring</th>
<th>Rank</th>
<th>WCS scoring</th>
<th>Rank</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 standard.StandardDrawingView.tool</td>
<td>1</td>
<td>util.PaletteButton.mousePressed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 contrib.AutoScrollHelper.Constructor</td>
<td>3</td>
<td>contrib.zoom.ZoomDrawingView$2$.mouseMoved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 framework.FigureAttributeConstant.getName</td>
<td>4</td>
<td>application.DrawApplication.toolDone</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 framework.FigureAttributeConstant.hashCode</td>
<td>5</td>
<td>contrib.AutoScrollHelper.Constructor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE II. FREQUENCIES OF METHOD CALLS CORRESPONDING TO THE THREE NETWORK ANALYSIS BASED SCORES.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>standard.StandardDrawingView.tool</td>
<td>util.PaletteButton.mousePressed</td>
</tr>
<tr>
<td>2</td>
<td>framework.FigureAttributeConstant.getName</td>
<td>figures.FigureAttributes.get</td>
</tr>
<tr>
<td>3</td>
<td>standard.StandardDrawingView.paintComponent</td>
<td>figures.AttributeFigure.getAttribute</td>
</tr>
<tr>
<td>4</td>
<td>framework.FigureAttributeConstant.getName</td>
<td>figures.AttributeFigure.getDefaultAttribute</td>
</tr>
<tr>
<td>5</td>
<td>framework.FigureAttributeConstant.getName</td>
<td>figures.AttributeFigure.getAttribute</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Rank</th>
<th>NetBeans call frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 standard.StandardDrawingView.tool</td>
<td>5</td>
<td>435.374</td>
</tr>
<tr>
<td>2 framework.FigureAttributeConstant.getName</td>
<td>5</td>
<td>172.374</td>
</tr>
<tr>
<td>3 standard.StandardDrawingView.paintComponent</td>
<td>5</td>
<td>145.374</td>
</tr>
<tr>
<td>4 framework.FigureAttributeConstant.getName</td>
<td>5</td>
<td>125.374</td>
</tr>
<tr>
<td>5 framework.FigureAttributeConstant.getName</td>
<td>5</td>
<td>105.374</td>
</tr>
</tbody>
</table>
system represented by the network. In addition, we also ranked the methods according to their call frequency determined through the use of NetBeans profiling. The top-5 ranked methods for all three network analysis based rankings and for the NetBeans call frequency ranking are shown in Table1.

4.3 Evaluation

In order to check the functional importance of the method calls selected on the basis of network analysis we approximated the removal of the corresponding edges of the network by disabling these methods one-by-one and trying to execute the same sequence of operations that we executed to generate our dynamic analysis data. For each disabled method we tried to do the minimal damage to the software code in order to avoid trivial errors (e.g. if the method is expected to return a pointer to an object and does not return anything then it generates an error immediately). Note that we do not fully disable a class, as in many cases that will cause immediate crash of the software, at the same time disabling a method of a class (i.e. not executing the contents of the method, while providing an output of the right kind, but with a default contents to objects that call the method) in many cases do not have user-observable effects on the execution of the software.

We note that the choice of method disabling may be to some extent subjective. However we tried to do this in a principled manner by applying the same kind of disabling to similar kinds of methods and by keeping the way of disabling as simple as possible, while avoiding causing trivial errors. In case of methods for which the return type is ‘void’ we simply shortcut the method’s entry and exit, without executing anything in between. In case of methods that return an object we created a default object of the right type, which is returned by the method, while the actual execution of the method is skipped.

Running the software after making the damage (i.e. disabling the normal functioning of a method) is expected to lead to a crash or some other user-observable significant alteration of the functionality of the software, if the network analysis prediction about functional importance of the method is correct. Here we assume that the functionality of the software is its expected interaction with the user, i.e. anything that happens or does not happen such that the user is not aware of this does not affect this user-centered functionality of the software system.

For the purpose of comparison we randomly selected 101 methods and evaluated their functional importance as described above – we found that 59% of these methods were functionally important. The results show (see Figure 2) that top-ranked methods according to the

![Figure 2. Graphical representation of the effectiveness of edge ranking methods. The lines present results for the four ranking methods (based on network metrics HCS, WCS, BWS, and NetBeans call frequency – CFr) by showing the percentage of the ranked methods up to a given rank, which are experimentally checked to be functionally important methods. For example, 100% for a rank r means that all methods up to rank r are functionally important, 50% means that only half of them up to this rank are functionally important. The horizontal dashed line shows the percentage of randomly chosen methods that are functionally important. The vertical axis shows the percentage of methods that are checked to be functionally important out of the top r ranked methods. The value of r is given by the horizontal axis.](image)

HCS, WCS and NetBeans call frequency rankings are more likely to be functionally important than randomly picked methods. However the functional importance prediction performance of these rankings is comparable or below the random chance for ranks above 14/15 in case of call frequency and HCS ranking, and above 20 in case of WCS ranking. The ranking based on BWS metric does not predict important methods better than chance.

One way to assess the functional importance of method calls contained in the dynamic analysis data is to disable them one-by-one and evaluate the effect of this disabling on the functionality of the program in the context of the considered execution scenario. However this is likely to be very time consuming (e.g. in our case the evaluation of 100 disabled method calls took around 16 hours of work). Applying network analysis to the dynamic analysis data is much less time consuming than the disabling analysis of all identified method calls (e.g. in our case the application of the network analysis methods to the dynamic analysis data took around 2 minutes).

5. Conclusions and future work

We analysed here the joint application of network analysis methods [10] and dynamic analysis of software systems [4] to find functionally important method calls in the JHotDraw 6.01b software. Our results indicate that some network analysis methods
(HCS, WCS and NetBeans call frequency ranking) can detect top-ranked edges that correspond with better than chance likelihood to functionally important method calls. However, the BWS score ranking does not work better than chance. We aim to improve the prediction performance of the network analysis methods by considering combined rankings as well.

The way of disabling a method is equivalent to the removal of more than one edge from the network. We aim in the future to use new reflection and introspection features provided by some software development environments to investigate the context dependent disabling of method that is equivalent of removal of single edges of the network.

The definition of functional importance that we used here is qualitative. A quantitative definition could be based for example on repeated execution of the usage scenario by a large group of users and quantitative evaluation of their user experience reports.

We aim to extend the presented work to much larger software systems and also to software written in other language environments (e.g. C++). We expect that by appropriate selection of network analysis methods we can build a combined methodology of dynamic analysis and network analysis that can support improvement of software understanding and re-engineering.

6. References


Exploiting Dynamic Information in IDEs Eases Software Maintenance

David Röthlisberger
Software Composition Group, University of Bern, Switzerland
roethlis@iam.unibe.ch

Abstract

The integrated development environment (IDE) is the primary tool used by developers to maintain software systems. The IDE, however, narrowly focuses on the static structure of a system, neglecting dynamic behavior and dynamic relationships between static source artifacts such as classes and methods. Developers often have difficulties to understand the dynamic aspects of a system just based on the static source perspectives provided by IDEs. Existing IDE tools to analyze the running of software systems such as debuggers or profilers present volatile dynamic information from specific system executions, requiring developers to manually trigger debugging or profiling sessions. To better support the understanding and maintenance of software systems, we developed several extensions to traditional IDEs to incorporate dynamic information in the static source perspectives. In this paper we describe these extensions and report on the empirical experiments we conducted to evaluate the practical usefulness of these IDE extensions.

Keywords: dynamic analysis, development environments, software maintenance, program comprehension

1 Introduction

Object-oriented language features such as inheritance, abstract types, late-binding, or polymorphism lead to distributed and scattered code, rendering a software system hard to understand and maintain when just looking at its static source artifacts such as classes or methods [1, 3, 13]. By means of dynamic analysis developers can explore the dynamic aspects of software systems and hence better understand the implementation of these systems [2, 13]. Even though the concept of dynamic analysis has been widely studied [2, 3, 8, 12, 13], traditional integrated development environments (IDEs), the primary tools used by developers to maintain software systems, still purely operate on static source code and do not reveal dynamic relationships between distributed source artifacts, which makes it difficult for developers to understand and navigate software systems in these IDEs [10, 11]. Due to the lack of dynamic information in IDEs developers are forced to build up a mental model of a system’s dynamic behavior based on a static view of the system. Such a mental model, however, is error-prone as it is very difficult to understand a large object-oriented system without having available information about its running [1, 3].

In this paper we propose to enhance the Eclipse IDE [4] for Java with dynamic information in order to help developers during program comprehension and software maintenance activities. To achieve the goal of supporting developers to more efficiently maintain object-oriented code, we augment the static source perspectives of IDEs with dynamic information. We contribute several different techniques to integrate dynamic information seamlessly into the IDE. The implemented techniques are:

- Source code enrichments to embed dynamic information such as runtime types of variables or callers of methods directly in an IDE’s source code view
- Presenting dynamic metrics such as number of method invocations or number of objects created in a method next to the source code perspectives
- Collaboration view, a navigable view showing the dynamic collaborators of source artifacts, for instance all callers and callees of a method

We validate these techniques by means of empirical experiments with professional developers to assess the practical usefulness of the availability of dynamic information in the traditional perspectives of IDEs.

In the following we first describe these techniques in detail and subsequently report on how we evaluated their practical usefulness. The paper is structured as follows: In Section 2 we present the enrichments to the IDE’s source code view to present dynamic information, Section 3 reports on the integration of various dynamic metrics, and Section 4 explores the collaboration view we integrated into the IDE. In Section 5 we describe the conducted empirical experiment to evaluate the implemented techniques embedding dynamic information in IDEs. Section 6 concludes the paper and outlines ideas for further work.
2 Source Code Enrichments

All techniques to present dynamic information have been implemented for the Eclipse IDE. We opted for this IDE since it is the most widely adopted IDE in industry [5], at least for developing Java software. The dynamic information that is exploited by these techniques typically stems from different system executions. We aggregate dynamic information over multiple executions to obtain a more complete picture of the dynamics of a system compared to debuggers or profilers that just focus on a specific execution. The aggregation of dynamic information in the IDE supports developers in understanding the general execution patterns of a system whereas the traditional debugger can be applied to study the execution flow of a specific scenario.

As a technique to complement source code with dynamic information without impeding its readability we opted to use hovers, small windows that pop up when the mouse hovers over a source element (a method name, a variable, etc.). Hovers are interactive, which means the developer can for instance open the class of a receiver type by clicking on it. We now describe the integration of dynamic information into the source code:

![Figure 1. Hover appearing for a method name in its declaration.](image1)

### Method header

- The hover that appears on mouse over the method name in a method header shows (i) all senders invoking that particular method, (ii) all callees, that is, all methods invoked by this method, and optionally (iii) all argument and return value types. For each piece of information we also show how often a particular invocation occurred. For instance for a sender, we display the qualified name of the method containing the send (that is, the calling method) and the number of invocations from this sender. Optionally, we also display the type of object to which the message triggering the invocation of the current method was sent, if this is a sub-type of the class implementing the current method. For a callee we provide similar information: The class implementing the invoked method, the name of the message, and how often a particular method was invoked. Additionally, we can show concrete receiver types of the message send, if they are not the same as the class implementing the called method. Figure 1 shows a concrete method name hover for method `readFileEntriesWithException`.

In a method header, we can optionally show information about argument and return types, if developers have chosen to gather such data. Hovers presenting this information appear when the mouse is over the declared arguments of a method or the defined return type. These hovers also include numbers about how often specific argument and return value types occurred at runtime.

### Method body

We also augment source elements in the method body with hovers. For each message send defined in the method, we provide the dynamic callee information similarly as for the method name, optionally along with argument or return types that occurred in this method for that particular message send at runtime, as shown in Figure 2. Of course all these types listed are always accompanied with the number of occurrences and the relative frequency of the specific types at runtime.

3 Dynamic Metrics

There are two kind of rulers next to the source editor: (i) the standard ruler on the left showing local information and (ii) the overview ruler on the right giving an overview over the entire file opened in the editor. In the traditional Eclipse IDE these rulers denote annotations for errors or warnings in the source file. Ruler (i) only shows the annotations for the currently visible part of the file, while the overview ruler (ii) displays all available annotations for the entire file. Clicking on such an annotation in (ii) brings the developer to the annotated line in the source file, for instance to a line containing an error.

We extended these two rulers to also display dynamic metrics. For every executed method in a Java source file the overview ruler presents, for instance, how often it has been executed on average per system run using three different icons colored in a hot/cold scheme: blue means only a few, yellow several, and red many invocations. Clicking on such an annotation icon causes a jump to the declaration of the method in the file. The ruler on the left side provides more detailed information: It shows on a scale from 1 to 6 the
frequency of invocation of a particular method compared to all other invoked methods. A completely filled bar for a method denotes methods that have been invoked the most in this application. The dynamic metrics in these two rulers allow developers to quickly identify hot spots in their code, that is, methods being invoked frequently. The applied heat metaphor allows different methods to be compared in terms of number of invocations.

![Figure 3. Rulers left and right of the editor view showing dynamic metrics.](image)

To associate the continuous distribution of metric values to a discrete scale with for instance three representations (e.g., red, yellow, and blue), we use the k-means clustering algorithm [7].

To see fine-grained values for the dynamic metrics, the annotations in the two columns are also enriched with hovers. Developers hovering over a heat bar in the left column or over the annotation icon in the right bar get a hover displaying precise metric values, for instance exact total numbers of invocations or even number of invocations from specific methods or receiver types. Furthermore, developers can choose between different dynamic metrics to be visualized in the rulers. Besides the number of invocations of methods, we also provide metrics such as the number of objects a method creates, the number of bytecodes it executes, and the amount of memory it allocates, either on average or in total over all executions. Such metrics allow developers to quickly assess the complexity of the execution of very complex code, that is, many bytecode instructions. To assess the complexity of the execution of a piece of code we also gather the number of bytecode instructions executed when invoking a particular method. This metric can be combined with the number of created objects metric to reveal which types of objects consume most memory and thus are candidates for optimization.

Number of created objects. By reading static source code, a developer usually cannot tell how many objects are created at runtime in a class, in a method or in a line of source code. It is unclear whether a source artifact creates one or one thousand objects — or none at all. This dynamic metric, however, is useful to assess the costs imposed by the execution of a source artifact, to locate inefficient code, or to discover potential problems, for instance inefficient algorithms creating enormous numbers of objects.

Allocated memory. Different objects vary in memory size. Having many but very tiny objects might not be an issue, whereas creating a few but very huge objects could be a sign of an efficiency problem. Hence, we also provide a dynamic metric recording memory usage of various source artifacts such as classes or methods. This metric can be combined with the number of created objects metric to reveal which types of objects consume most memory and thus are candidates for optimization.

Number of executed bytecode instructions. The static source code does not disclose how many bytecode instructions have to be executed during the runtime of the code. Calling a particular method in a piece of code might trigger the execution of very complex code, that is, many bytecode instructions. To assess the complexity of the execution of a piece of code we also gather the number of bytecode instructions executed when invoking a particular method. This metric is also an estimator for the execution time of a particular method invocation.

4 Collaboration View

In a separate view next to the source code editor of Eclipse (Figure 4), we present all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the collaboration view shows the collaborators at the package, class, or method level; that is, it lists all packages or classes invoking methods of the package or class in which the selected method is declared (callers). The collaboration view also shows all packages or classes with which the package or class declaring the method is actively communicating (callee). For the method itself, the collaboration view lists all direct callers and callees.

This collaboration view allows developers to navigate the callers and callees. If for instance a caller of a method is selected, the view is refreshed to show all callers of the selected caller, and so on. Like this, developers can easily navigate through all dynamic callers of source artifacts of interest.
Figure 4. A view of all collaborators of the selected artifact (package, class, or method).

5 Validation

We conducted a controlled experiment with 30 professional Java developers to evaluate the benefits for software maintenance that arise from the three presented means to integrate dynamic information into IDEs.

Experimental Procedure. We asked the experiment subjects to solve five typical software maintenance tasks and analyzed the time spent to solve these tasks and the correctness of the solutions. Each subject was either assigned to the control group or to the experimental group. The experimental group had available dynamic information integrated with the aforementioned techniques while the control group used a standard Eclipse IDE. The tasks we gave the subjects are concerned with analyzing and gaining an understanding for various features of jEdit. We selected tasks representative for real maintenance scenarios and not being biased towards dynamic analysis by following the framework of Pacione et al. [9]. For solving the tasks, subjects had to provide an answer in free text, they could not select from multiple choices.

The dynamic information shown in Eclipse to the subjects of the experimental group was obtained by executing all actions from the menu bar of jEdit to make sure that this pre-recorded information is not biased towards the experiment tasks. As the control group did not receive any dynamic information, we clearly stated in the task descriptions how to run and analyze the feature under study with the conventional debugger in Eclipse.

The two dependent variables studied in this experiment, time the subjects spent to answer the questions, and correctness of the answers, were manually determined by the experimenters. The time spent on a task is the time span between the starting time of one task and the next. Correctness is measured using a score from 0 to 4 according to the overlap with the model answers, which forms a set of expected answer elements that have been identified by the experimenters beforehand.

To determine whether dynamic information has a statistically significant effect on the variables time spent and correctness, we applied the parametric, one-tailed Student’s t-test at a confidence level of 95% ($\alpha=0.05$).

Results. The results of the experiment were promising. On average, the experimental group spent significantly less time solving the maintenance tasks, we could measure a 17.5% decrease in time spent on the tasks for the experimental group compared to the control group. With the Student’s t-test we verified whether the availability of dynamic information in the IDE had an impact on the time to solve the maintenance tasks. The p-value resulting from the t-test is with 0.0016 considerably lower than $\alpha=0.05$, which means that the time spent is statistically significantly reduced by the availability of dynamic information.

For the correctness variable we could discover a 33.5% increase for the subjects having available dynamic information compared to those using a traditional Eclipse IDE. Applying the t-test to the correctness variables shows that this increase is statistically significant: the t-test gives a p-value of 0.0001 which is clearly below $\alpha=0.05$, which means that the time spent is statistically significantly reduced by the availability of dynamic information.

Qualitative Feedback. In a debriefing questionnaire the subjects provided qualitative feedback about the usefulness of dynamic information integrated into the IDE. On a Likert scale from 0 (useless) to 4 (very useful), the subjects

1http://www.jedit.org/
rated the various techniques to embed dynamic information in the IDE. The source code enrichments obtained an average rating of 3.6, the dynamic metrics embedded in the ruler columns were rated with 3.2, and the collaboration view got an average rating of 3.7. These ratings clearly show that subjects considered all techniques to be useful for solving the software maintenance tasks of this experiment. In particular the collaboration view but also the source code enrichments have been an important aid to quicker and more accurately complete the imposed tasks.

Subjects also gave feedback about how often they used what kind of dynamic information during the experiment. It turned out that information about dynamic collaborators has been used the most, nearly by all subjects in all tasks, while dynamic information shown in the source code views, e.g., information about runtime types has been used less often. Most subjects have just occasionally used dynamic metrics, for instance for tasks where they had to assess performance aspects of the application.

These results from the controlled experiment make us confident that the implemented techniques to embed dynamic information seamlessly and tightly in the IDE are indeed helpful for developers to more efficiently and more correctly solve typical software maintenance tasks. A more detailed report on this experiment is available in the Masters thesis of Marcel Haerry [6].

6 Conclusions

In this paper we described three different techniques we implemented to seamlessly integrate dynamic information into the Eclipse IDE to help developers to easier and more accurately understand the dynamic behavior of Java systems. These three techniques are (i) source code enrichments embedding dynamic information such as runtime types, callers or callees of a method; (ii) dynamic metrics such as number of created objects, memory size of these objects, or number of executed bytecode instructions; and (iii) a collaboration view presenting the dynamic collaborators, that is, the callers and callees of particular source artifacts such as packages, classes, or methods. We evaluated the practical usefulness of these three contributed techniques by means of a controlled experiment with 30 professional software developers solving typical software maintenance tasks in a large unfamiliar application (i.e., jEdit). The results of this experiment show that developers can 17.5% faster and 33.5% more correctly solve the maintenance tasks when Eclipse is enhanced with the aforementioned dynamic information compared to a standard Eclipse installation.

In the future we aim at extending the integration of dynamic information into Eclipse, for instance by embedding visualizations of the collaboration between source artifacts to ease the navigation and understanding of collaboration patterns. Furthermore, we plan to apply the proposed techniques to large industrial systems such as rich client platform systems in the finance sector to gather more empirical feedback from practice.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Bringing Models Closer to Code” (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010)

References