Hierarchical Abstraction of Execution Traces for Program Comprehension

Kaj Dreef
Hierarchical Abstraction of Execution Traces for Program Comprehension

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Kaj Dreef
born in Amstelveen, the Netherlands
Hierarchical Abstraction of Execution Traces for Program Comprehension

Author: Kaj Dreef
Student id: 4107861
Email: kdreef@student.tudelft.nl

Abstract

Understanding the dynamic behavior of a software system is one of the most important and time-consuming tasks for today’s software maintainers. In practice, understanding the inner workings of software requires studying the source code and documentation and inserting logging code to map high-level descriptions of the program behavior with low-level implementation, i.e., the source code. Unfortunately, for large codebases and large log files, such cognitive mapping can be quite challenging. To bridge the cognitive gap between the source code and detailed models of program behavior, prior software-execution mining research primarily focused on reducing the size of the low-level instruction execution traces. In contrast, in this thesis we propose a generic approach to present a semantic abstraction with different levels of functional granularity from full execution traces. Our approach mines multiple execution traces to identify frequent behaviors at multiple levels of abstraction, and then analyzes and labels individual execution traces according to the identified major functional behaviors of the system. To validate our technique, we conducted a case study on a large-scale subject program, JAVAC, to demonstrate the effectiveness of the mining result. Furthermore, the results of a user study demonstrate that our technique is capable of presenting users with a high-level comprehensible abstraction of execution behavior.

Thesis Committee:
Chair: Prof. Dr. A. van Deursen, EEMCS, Delft University of Technology
Company supervisor: Prof. Dr. J.A. Jones, University of California, Irvine
Committee Member: Dr. A. Vilanova, EEMCS, Delft University of Technology
Committee Member: Dr. A. Bacchelli, EEMCS, Delft University of Technology
Preface

This thesis is the end product of a year long research project. It was a collaboration between Yang Feng, a Ph.D. student at the University of California Irvine (UCI), and me, a Master student at the Delft University of Technology (TUD), under supervision of professor James A. Jones (UCI) and professor Arie van Deursen (TUD).

First, I would like to thank Arie for bringing me into contact with Jim and always finding the time to Skype and giving me valuable advice even though there was a nine hour time difference between Irvine and Delft. Also, I want to thank Jim for inviting me to be part of the SpiderLab, and advising me during the course of the project.

I also want to thank Vijay, Yang, and JY at the SpiderLab for their support and help while doing the research. Their insights made me a better researcher and programmer and lifted up the quality of the research we did. And a special thanks to Yang for being a great project collaborator.

Finally, I want to thank my friends, parents, sister, and Samantha for their support and telling me that “being a big ball of stress is just part of writing your thesis.” Thank you all for the support and love during this last year. Also, I trust this thesis is enough “proof” that I actually did do research and did not just enjoy the California sun.

Kaj Dreef
Delft, the Netherlands
June 9, 2017
# Contents

**Preface** iii

**Contents** v

**List of Figures** vii

**List of Tables** ix

1 Introduction 1

1.1 Thesis Statement ........................................... 1
1.2 Contributions ............................................. 2
1.3 Research Context ........................................... 3
1.4 Thesis Structure ........................................... 3

2 Motivation and Challenges 5

2.1 Challenge 1: Information Overload ......................... 5
2.2 Challenge 2: Behaviors Contain Sub-Behaviors ............. 6
2.3 Challenge 3: Incomprehensible Execution Traces .......... 6

3 Approach 7

3.1 Data Collecting and Preprocessing ........................ 8
3.2 Model Building ............................................ 18
3.3 Execution Abstraction ..................................... 23

4 Hierarchical Phase Visualization 25

4.1 Data Model ................................................ 26
4.2 Shneiderman’s Criteria .................................... 28
4.3 SAGEVIS .................................................. 28
4.4 Multithread Visualization .................................. 33
4.5 Implementation of SAGEVIS ................................. 34
## Contents

5 Evaluation 35
- 5.1 Experimental Design ................. 35
- 5.2 Information Overload and Computational Efficiency ................. 37
- 5.3 Behavior Identification Based on Generated Labels ................. 41
- 5.4 Case Study on Javac ................. 42
- 5.5 Impact of the Frequent Pattern Mining ................. 46

6 Discussion 51
- 6.1 Addressing Information Overload ................. 51
- 6.2 Addressing Behavior Subsumption ................. 51
- 6.3 Addressing Comprehensibility of Execution Traces ................. 52
- 6.4 Computational Efficiency ................. 52
- 6.5 Advantages and Limitations of Frequent Pattern Mining ................. 53
- 6.6 Limitations of Model ................. 53
- 6.7 Threats to Validity ................. 53

7 Related Work 55
- 7.1 Execution Trace Visualization ................. 55
- 7.2 Execution Mining ................. 56
- 7.3 Trace Reduction and Abstraction Techniques ................. 56

8 Conclusions and Future Work 57
- 8.1 Conclusions ................. 57
- 8.2 Future Work ................. 58

Bibliography 61

A Glossary 65

B Ranking of Labels for JEdit Behaviors 67

C Questionnaire 69
List of Figures

3.1 The technique’s framework for abstracting and labeling execution events, i.e.,
the training stage for the SAGE technique .............................................. 7
3.2 Dynamic call tree of an execution trace with call depth .......................... 9
3.3 Dynamic call tree of an execution trace with phases ............................. 10
3.4 Hierarchical phase structure of the dynamic call tree ............................. 10
3.5 Round 1 of the phase detection algorithm .......................................... 15
3.6 Round 2 of the phase detection algorithm .......................................... 15
3.7 Round 3 of the phase detection algorithm .......................................... 15
3.8 Round 4 of the phase detection algorithm .......................................... 16
3.9 Round 5 of the phase detection algorithm .......................................... 16
3.10 Round 6 of the phase detection algorithm ........................................ 16
3.11 Round 7 of the phase detection algorithm ........................................ 17
3.12 Round 8 of the phase detection algorithm ........................................ 17
3.13 Round 9 of the phase detection algorithm ........................................ 18
3.14 Final preliminary phase tree .......................................................... 18
3.15 Phase tree to demonstrate the model building steps ............................. 20
4.1 Different hierarchical structure based on technique used .......................... 27
4.2 Core idea of how to present the hierarchical structure ............................ 29
4.3 Clicking inside the red box unfolds the phase in the timeline ..................... 30
4.4 Clicking inside the red box collapses the phase and its siblings in the timeline 30
4.5 Zoom feature demonstrated on a NanoXML visualization ......................... 31
4.6 Panning feature demonstrated on a NanoXML visualization ..................... 32
4.7 Phase highlighting feature demonstrated on a NanoXML visualization ........ 32
4.8 Comparison of SAGEVis at the same level for NanoXML with and without
frequent pattern mining ............................................................................... 34
5.1 Trace size (in blue) and average number of phases at each top-8 levels of the
hierarchy (in red). Note log scale. .............................................................. 41
5.2 Exploring the abstracted execution trace of JAVAC with SAGEVis ............. 44
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>GSON visualization demonstrating navigating</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>GSON visualization presenting the three high-level behaviors</td>
<td>48</td>
</tr>
<tr>
<td>5.5</td>
<td>NanoXML visualization with (top) and without (bottom) frequent pattern mining</td>
<td>49</td>
</tr>
<tr>
<td>8.1</td>
<td>An example of SAGEVis with and without the collapsing of threads</td>
<td>59</td>
</tr>
<tr>
<td>8.2</td>
<td>Dynamically Inferred State Machine at different levels of granularity</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Small excerpt from an execution trace of JODATIME
3.2 Phases and their children of Figure 3.2
3.3 Round 5: Finding phases in the history stack
3.4 Round 8: Finding phases in the history stack
3.5 Round 9: Finding phases in the history stack
3.6 Method name and its corresponding method ID of Figure 3.15
3.7 Global keys for the phases found in Figure 3.3
4.1 Tasks introduced by Shneiderman to assess the usefulness of a GUI for information visualizations
5.1 Size abstraction for each step of the training procedure
5.2 Time cost in seconds for each step of the training procedure
5.3 Total time cost for execution abstraction
5.4 Distribution of the phases per level in NANOXML
5.5 Distribution of the phases per level in JAVAC
5.6 Distribution of the phases per level in JEDIT
5.7 The user study results of JEdit
5.8 Labels belonging to phases of JAVAC
B.1 Behavior and the corresponding labels in order of rank
Chapter 1

Introduction

With the increasing size and complexity of modern software, software maintenance has become one of the most expensive tasks of today’s software development. Because software must be sufficiently understood before it can be properly modified or enhanced, program comprehension plays an essential role in software maintenance. Further, in practice, understanding dynamic software behavior and mapping high-level functionalities with their corresponding implementation details are often necessary for the developers to efficiently and correctly perform maintenance tasks, such as adding features or debugging.

Unfortunately, understanding software execution behavior can be difficult, particularly given the exceptionally large execution traces that describe all events within executions. To facilitate these tasks, researchers have proposed many techniques to help developers identify the high-level functionality from program’s dynamic behaviors, i.e., comprehending software executions with their large number of execution events.

One approach (e.g., [32, 26, 34]) is to summarize executions into “phases,” i.e., groups of execution events that constitute functionality behaviors within the execution. Another approach (e.g., [23, 26, 10]) is to visualize executions, allowing developers to observe interesting internal behaviors of the software execution.

These approaches are promising but limited in a number of ways that might be improved upon. For example, (1) execution abstractions are limited to a single level of granularity and thus might not provide sufficient support to a variety of maintenance tasks that require comprehension at either a coarse or fine grain, or both (e.g., [26, 34, 32]); (2) executions events are filtered to support understanding of overall behavior but may omit important events (e.g., [32]); and (3) execution events are visualized to allow developers to examine behavior, but rely on the developer to infer the higher-level functionalities without guidance on the functionalities (e.g., [23, 10]).

1.1 Thesis Statement

In this work, we present SAGE—a novel technique that (1) identifies a dictionary of frequent functionalities that occur within multiple software executions for the software system under investigation, (2) creates a hierarchical representation of an execution under investigation
using the functionality dictionary, (3) labels the hierarchical execution representation comprehensibly, and (4) visualizes the representation in a way that allows the developer to explore the execution at multiple levels of granularity. The hierarchy provides representations of execution functionality at multiple levels of abstraction—for example, at a high-level, SAGE might describe a behavior as “processInput,” but at a lower level, it might describe that phase with multiple sub-behaviors such as “readFile,” “parseTokens,” and “createParseTree.”

Our technique provides two main advantages: (1) the output is a hierarchical structure that provides the maintainer with multiple comprehension levels as well as the guidance of locating the functionalities of interests, and (2) functionalities are abstracted and presented to include multiple frequent patterns of behavior to help identify significant behaviors.

1.2 Contributions

With our prototype implementation of SAGE, we evaluated its ability to (1) reduce the number of events within an execution trace to a comprehensible number, (2) create hierarchical representations of execution behavior that allow for user investigation of both high-level execution behavior and lower-level constituent behaviors within high-level behaviors of interest, and (3) produce a comprehensible and meaningful representation of software execution.

We present an evaluation that includes: (1) a quantitative evaluation of the size reduction of the execution trace to a more comprehensible abstraction, along with its computational costs; (2) a user study that assesses users’ ability to understand the technique’s generated behavior abstractions; (3) a case study that demonstrates how the technique works for a large program (JAVAC) and execution; and (4) a case study on the impact of frequent pattern mining. In summary, we found that SAGE substantially abstracts execution traces to a set of hierarchical functionalities that allow for both high-level and low-level developer investigations of execution behavior, in a way that is comprehensible and meaningful for many behaviors.

The main contributions of this thesis are as follows:

- We present our technique for mining execution traces to identify functionality behaviors (i.e., “execution phases”) for the software under investigation.

- Based on these identified behaviors, we present a technique for generating a hierarchical abstraction of execution behavior for an execution under investigation.

- With this hierarchical representation of an execution under investigation, we present our technique for labeling and visualizing it for developer inspection and exploration.
1.3 Research Context

Setting

This thesis is the result of a research project the author contributed to while visiting the Department of Informatics at the University of California, Irvine (UCI) between February 2016 and January 2017. Here the author collaborated with Yang Feng, a Ph.D. Student at the University of California, Irvine, under supervision of Professor James A. Jones (UCI) and Professor Arie van Deursen from Delft University of Technology (TUD).

The results of this research project was a paper, which is currently under review:

Hierarchical Abstraction of Execution Traces for Program Comprehension, by
Yang Feng, Kaj Dreef, James A. Jones, and Arie van Deursen.

Attribution

This research is based on a collaborative research project between the author and Yang Feng, with the following division of labor and responsibility. Yang Feng, was responsible for the duplicate detection, clustering, and frequent pattern mining described in Section 3.2.1, 3.2.2, and 3.2.3 respectively. The author was responsible for the phase detection, described in Section 3.1.2, and the visualization of the resulting data structure, described in Chapter 4. The remainder of the research was a joint effort.

Each chapter of this thesis except for the visualization chapter (4) is based on a section of the paper. However, some of the sections are extended for this thesis. The phase detection section (3.1.2) contains more details on the internal workings of the phase detection algorithm and an example for the phase detection. Furthermore, the discussion, evaluation, and conclusion have been extended to also cover the visualization.

1.4 Thesis Structure

This thesis is structured as following. In Chapter 2, we discuss the challenges that dynamic analysis faces and why they need to be solved. In Chapter 3, we present SAGE, our technique to overcome the challenges, and in Chapter 4, the visualization SAGEVIS. The technique and visualization are evaluated in Chapter 5 and discussed in Chapter 6. Related work to the approach presented in this thesis is discussed in Chapter 7. Finally, in Chapter 8 we conclude this thesis and discuss the areas where SAGE and SAGEVIS can be improved further improved in the future.
Chapter 2

Motivation and Challenges

Modern software is large in its codebase and its runtime behavior is complex. To observe and reason about low-level software execution, developers can produce “execution traces” (i.e., a log of internal execution events) by using an execution-trace instrumenter. Unfortunately, these execution traces often contain millions, or even billions, of low-level execution events, and their files are often sized in the gigabytes. Such large trace sizes can limit developer ability to understand and reason about the software execution behavior, which could assist in software-engineering activities, such as debugging or performance optimization.

To assist developers in understanding software execution and reduce the cognitive burden of interpreting extremely large execution traces, researchers have identified that software execution often includes “phases,” or commonly recurring behaviors that perform functionalities within the software. Reiss [26] articulates this point:

Software executes in phases. A simple system first does initialization, then reads input, then processes that input, and finally writes the result out. Actual systems typically go through various phases depending on different input commands and external events, varying processing requirements, and other related factors.

Cornelissen et al. [10] observed execution phases with their EXTRAVIS visualization. They observed that for a program they were studying, it contained a number of phases: “(1) an input phase, (2) a calculation phase, and (3) an output phase,” and moreover that these phases contained several repeating sub-phases. This observation by Cornelissen et al. particularly motivates this work to produce a hierarchical abstraction of an execution trace and its execution phases, which allows developers to understand behaviors and their sub-behaviors to assist in their maintenance tasks.

To achieve such a goal, we face three main challenges:

2.1 Challenge 1: Information Overload

Execution traces are typically very large—in the millions or billions of events, and file sizes in the gigabytes. Such large sizes of execution events creates difficulties for developers to
understand and act upon the data. Moreover, execution traces are often presented in terms of low-level events that do not convey much meaning on their own. Hence, techniques that abstract this detailed information to fewer, higher-level, and more comprehensible behaviors may be beneficial.

2.2 Challenge 2: Behaviors Contain Sub-Behaviors

As observed by Cornelissen et al., behaviors (i.e., execution phases) contain other behaviors, and their relationships between higher-level behaviors and lower-level constituent sub-behaviors are often hierarchical. As such, much of the prior work in execution-phase detection that detects phases at only a single level of granularity cannot express such relationships or allow developers to interactively explore and dissect phases. Hence, we seek to provide phase detection that is hierarchical and enables interactive investigation and exploration of parts of an execution.

2.3 Challenge 3: Incomprehensible Execution Traces

Not only are execution traces large (as described in Challenge 1), but they are also composed of many low-level execution events that without context or abstraction can be difficult to comprehend on their own. Hence, we seek to provide mechanisms that not only abstract coarser-grained execution-phase behaviors, but also provide helpful labels to ease comprehension of those phases.
Chapter 3

Approach

In this chapter, we introduce SAGE, our approach for hierarchically abstracting an execution trace and addressing the challenges mentioned in Chapter 2. In Figure 3.1, we present the framework of our technique to abstract and label execution-behavior phases. The input of our approach are execution traces of the subject program containing method-call events, which can be obtained with a dynamic instrumenter. In our implementation, we employed BLINKY [22] as the instrumenter to collect execution traces.

As depicted in Figure 3.1, our approach consists of two primary steps: (1) data collection and preprocessing and (2) model building and execution abstraction. In the following sections, we describe the major steps in each of these two stages.

![Figure 3.1: The technique’s framework for abstracting and labeling execution events, i.e., the training stage for the SAGE technique](image)
3. APPROACH

3.1 Data Collecting and Preprocessing

3.1.1 Execution Trace Collection

In our approach, execution traces are obtained using the dynamic instrumenter, BLINKY [22]. We configured BLINKY to collect method-invocation event traces. For our purpose, we instrumented the method-enter events because methods are intended to provide behavior-granular functionality. This intuition that method invocations (and their invocation sequences) provide a useful source for our behavior phase detection is mirrored by the work of both Pradel and Gross [24] and Salah et al. [28], who each found that method invocation sequences can represent usage scenarios within software execution.

Each execution of the instrumented program results in a method-level trace that consists of a sequence of method-invocation events. Each method-invocation event is captured as a triple of a (1) execution ID, (2) method signature, and (3) call depth.

3.1.2 Phase Detection

The original execution traces are difficult to comprehend, because they are large and contain voluminous information. Although it could help developers gain a better understanding of their program’s behavior, analyzing this kind of data is a time-consuming task. One way of making execution traces easier to comprehend is by reducing their size or summarizing them. However, reducing the size of the execution trace would require filtering out some information details, which could be important for the analyzing tasks. However, there are two essential disadvantages with filtering and summarizing: first, both kinds of techniques would require filtering out some information details, which could be important for the analyzing tasks; second, these kinds of techniques fail to maintain the method-invocation context in an human-friendly form, making it difficult for developers to find parts to their interests.

Our goal for the phase detection is to determine which parts of the trace perform functional behavior. By automatically and hierarchically determining where a phase begins and ends, the developer has a less time-consuming method through which they can gain understanding about the programs dynamic behavior. Contrast this method with going through the whole trace, which can consist of millions of method events.

Thus, based on the dynamic call trees (DCT), which presents a node for each method invocation and an edge from the caller to callee, we proposed a phase detection algorithm to recover the invocation context and identify the high-level functionality from massive execution traces.

One point worth noting is that the classic DCTs of modern large programs are still difficult to understand and trace, because the classic DCT is the most precise but space-inefficient data structure that can present the calling context of the execution [3]. Based on the call depth changes, our algorithm is capable of compressing the DCT structure and further identify the basic functionality units.

The program execution trace can be represented as a Dynamic Call Tree (DCT), where each node is a method event. The root node is the entry method of the program, and its children are the methods being invoked by that node. Those children, in turn, invoke other
3.1. Data Collecting and Preprocessing

Table 3.1: Small excerpt from an execution trace of JODA TIME

<table>
<thead>
<tr>
<th>Execution ID</th>
<th>Class Name</th>
<th>Method Name</th>
<th>Call Depth</th>
<th>Thread ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>404</td>
<td>BasicChronology</td>
<td>getDayOfMonth</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>405</td>
<td>BasicChronology</td>
<td>getYear</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>406</td>
<td>GregorianChronology</td>
<td>getAverageMillisPerYearDividedByTwo</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>407</td>
<td>GregorianChronology</td>
<td>getApproxMillisAtEpochDividedByTwo</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>408</td>
<td>BasicChronology</td>
<td>getYearMillis</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>409</td>
<td>BasicChronology</td>
<td>getYearInfo</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>410</td>
<td>BasicGJChronology</td>
<td>getMonthOfYear</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>411</td>
<td>BasicChronology</td>
<td>getYearMillis</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>412</td>
<td>BasicChronology</td>
<td>getYearInfo</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>413</td>
<td>GregorianChronology</td>
<td>isLeapYear</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>414</td>
<td>BasicChronology</td>
<td>getDayOfMonth</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.2: Dynamic call tree of an execution trace with call depth

methods. However, traces directly obtained through an instrumenter are not in the form of a DCT. In Table 3.1 an excerpt of an execution trace of JODA TIME is shown.

In Figure 3.2 we show a simplified dynamic call tree for the execution trace excerpt presented in Table 3.1. The vertical axis represents the call depth, and the order of method-enter events is represented on the horizontal axis. We can notice that the execution trace starts from the root phase `getDayOfMonth`, which consists of three sub-phases that perform their own behavior: the `getYear`, `getMonthOfYear`, and `getDayOfMonth`. These three branches perform their own task and they hide the details through encapsulation. Each of those functionalities of the execution can be seen as a phase of the program.

Phases can be found on each hierarchical level within the program. The end of a phase is marked by a return to a lower call depth. Equivalently, branching within a dynamic call-invocation tree signifies a phase boundary. In Figure 3.2 we can locate the phases by looking for branches in the dynamic call tree. For example, in the previous example we had a program that starts with `getDayOfMonth`, this phase has three sub-phases `getYear`, `getMonthOfYear` and `getDayOfMonth`. By locating all the branches in the dynamic call tree we can locate six phases, which are denoted in Figure 3.3 with the dashed boxes. As such, we discover a hierarchical phase tree depicted in Figure 3.4. The leaves of the hierarchical
3. Approach

Figure 3.3: Dynamic call tree of an execution trace with phases

phase tree are referred to as basic phases, and its parents are referred to as general phases.

Figure 3.4: Hierarchical phase structure of the dynamic call tree

3.1.3 Phase Detection Algorithm

We present the phase detection algorithm in Algorithm 1 and 2. The input of the phase detection is an ordered list of events for a single thread. The algorithm uses the change in call depth to determine when a phase starts and ends. The output is the root node of the phase tree. Each of the phases has its own unique ID and stores references to its children.

The main body of the phase detection can be found in Algorithm 1 and 2. It consists of a for-loop which iterates through the method events in the execution trace and two IfElse blocks. At the beginning we initialize an empty stack, called the history stack. The purpose of this stack is to store events and phases that have happened before the current event ($U_i$) that we are looking at.
A phase’s end point can be discovered by looking for a stagnation in the call depth by comparing two events: the previous event (i.e., top of the history stack) and the current event (i.e., $U_i$). Two types of stagnations can occur. First, the call depth of the previous and current event are equal, as illustrated in Figure 3.2. In this example, you will see a basic phase (identified as $U_6$), due to the stagnation between the events $U_6$ and $U_7$.

Second, the call depth of the previous event is higher than the current event, meaning there is a drop in the call depth. This stagnation shows us the end of the phase. Even though it shows us the end of the phase, we still need to determine the beginning of the phase, described in Algorithm 2. The current event’s call depth can be used to determine the beginning of the phase. To find the beginning of the phase, we need to look at previous events and phases, for this we can use the history stack. By popping events and phases of the stack we go back into time until the history stack is empty, which means we reached the start of the program (see line 19-22 in Algorithm 2), or we continue to pop off events until we find an event which has the same call depth as the current event’s call depth (see line 9-13 in Algorithm 2). When the call depths of the two events are equal then we have found the start of the phase. An example can be seen in Figure 3.2 here we see a stagnation between $U_4$ and $U_5$. As a result, the events $U_2$, $U_3$, and $U_4$ all need to be subsumed into a phase. Note that the call depth of event $U_5$ is the same as the call depth of $U_5$.

The algorithm returns the root node of the dynamic call tree. The creation of a phase of the tree is presented in Algorithm 3. Each phase will contain the following information:

- **Phase ID**: A unique ID for each phase
- **Parent**: The ID of the parent phase (empty in case of the root node)
- **Call depth**: The call depth of where the phase starts
- **Head method**: Contains the method signature of the first event in the phase
- **Children**: A list of phase IDs of all its children
- **Signatures**: A list of signatures IDs in the order of invocation.
- **Minimum Execution ID**: Smallest execution ID among its children.
- **Maximum Execution ID**: Largest execution ID among its children.

The execution ID is an ID generated by the instrumenter which presents the order in which the events occurred in the execution trace. Finally, the hierarchical phase-tree contains information about which methods were called and the order that they were called, as well as the call depth of each phase. Such information will be used in the subsequent clustering step of the approach.
Algorithm 1: Phase Detection

**Input**: `eventList`: List of execution events

**Output**: `root`: The root node of the phase tree

**Initialization**: Initialize an empty stack `historyStack` to hold previous execution events or phases.

1. Initialize an empty `root` node.

2. `eventList.append(end of file event with call depth of -1)`

3. `foreach event u in eventList do`

4. `if historyStack.size() == 0 then`

5. `historyStack.push(currEvent)`

6. `continue`

7. `end`

8. `prevEvent ← historyStack.peek()`

9. `if prevEvent.getCallDepth() < u.getCallDepth() then`

10. `historyStack.push(u)`

11. `end`

12. `else if prevEvent.getCallDepth() == u.getCallDepth() then`

13. `historyStack.pop()`

14. `node ← createNode(prevEvent)`

15. `historyStack.push(node)`

16. `historyStack.push(u)`

17. `end`

18. `else if prevEvent.getCallDepth() > u.getCallDepth() then`

19. `root ← findPhasesInHistory(u, historyStack)`

20. `end`

21. `return root`

22. `end`

### 3.1.4 Example of the Phase Detection

In this section, a step by step demonstration of the phase detection is presented. The example execution is the dynamic call tree presented in Figure 3.2. Each round shows an execution trace as a dynamic call tree where the vertical axis symbolizes the call depth and the horizontal axis represents time. The blue-colored event (solid box) is the current event, and the red-colored event (dashed box) is the previous event.

In this example, six phases are found. The children of each phase are presented in Table 3.2. When we work our way back from the final discovered phase, i.e., the root phase $P_6$, we see among its children the phases $P_1$, $P_4$, and $P_5$. $P_4$ has two sub-phases, namely $P_2$ and $P_3$; based on this, it is possible to reconstruct the preliminary phase tree for this execution trace. This hierarchical tree structure will be the basis for later steps in SAGE.
3.1. Data Collecting and Preprocessing

Algorithm 2: Function `findPhasesInHistory` used in the phase detection algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>begin</td>
</tr>
<tr>
<td>2</td>
<td>while <code>historyStack.size() &gt; 0</code> do</td>
</tr>
<tr>
<td>3</td>
<td>topStack ← <code>historyStack.pop()</code></td>
</tr>
<tr>
<td>4</td>
<td>if <code>topStack.getCallDepth() == prevStack.getCallDepth()</code> then</td>
</tr>
<tr>
<td>5</td>
<td>p ← <code>createNode(prevStack, children)</code></td>
</tr>
<tr>
<td>6</td>
<td>children.append(topStack)</td>
</tr>
<tr>
<td>7</td>
<td>children.append(p)</td>
</tr>
<tr>
<td>8</td>
<td>else if <code>topStack.getCallDepth() == endEvent.getCallDepth()</code> then</td>
</tr>
<tr>
<td>9</td>
<td>children.append(topStack)</td>
</tr>
<tr>
<td>10</td>
<td>node ← <code>createNode(topStack, children)</code></td>
</tr>
<tr>
<td>11</td>
<td><code>historyStack.push(node)</code></td>
</tr>
<tr>
<td>12</td>
<td><code>historyStack.push(endEvent)</code></td>
</tr>
<tr>
<td>13</td>
<td>return None</td>
</tr>
<tr>
<td>14</td>
<td>else</td>
</tr>
<tr>
<td>15</td>
<td>children.append(topStack)</td>
</tr>
<tr>
<td>16</td>
<td>end</td>
</tr>
<tr>
<td>17</td>
<td>prevStack ← topStack</td>
</tr>
<tr>
<td>18</td>
<td>end</td>
</tr>
<tr>
<td>19</td>
<td>root ← <code>createNode(topStack, children)</code></td>
</tr>
<tr>
<td>20</td>
<td>return root</td>
</tr>
<tr>
<td>21</td>
<td>end</td>
</tr>
</tbody>
</table>

Based on these phases and sub-phases, the tree structure in Figure 3.14 can be built.

In the main body of the algorithm, we use a sliding window to analyze two sequential events to see if there is a stagnation in the call depth. When there is no stagnation then we push the events to the history stack. In round 1, see Figure 3.5, the history stack is initially empty. As a result, the initial event $U_1$ is pushed to the history stack. In round 2, see Figure 3.6, the history stack is not empty anymore, however the current event has a higher call depth than the previous events, thus the event is pushed to the history stack. The same situation occurs in round 3 and 4, see Figures 3.7 and 3.8 respectively.

In round 5, see Figure 3.9, we see a drop in the call depth. In Table 3.3, we show how we pop off events of the history stack to find phase $P_1$.

In round 6, there is again no stagnation, so we push $U_6$ to the stack. However, in round 7 (Figure 3.11), there is a stagnation, but the call depth stays equal. When this occurs it means the previous event, in this case $U_6$, is a phase, also referred as a basic phase.
3. Approach

Algorithm 3: Create new phase node for tree.

**Input**: event: First event in of the phase.
  children: Contains all the children.

**Output**: phase: A phase

**Initialization**: Initialize a new phase node called phase

```plaintext
1 begin
2   phase ← new Phase()
3   phase.calldepth ← event.getCallDepth()
4   phase.id ← getUniqueID()
5   phase.headMethod ← event.getHeadMethod()
6   phase.minExecID ← event.getMinExecID()
7   phase.maxExecID ← event.getMaxExecID()
8
9   phase.signatures.append(event.getSignatures())
10
11   foreach child in children do
12     if child is a Phase then
13        child.setParent(phase)
14        phase.children.append(child)
15     end
16     phase.signatures.append(child.getSignatures())
17   end
18
19 return phase
end
```

Table 3.2: Phases and their children of Figure 3.2

<table>
<thead>
<tr>
<th>Phase</th>
<th>All Children</th>
<th>Only Phases</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>U2, U3, U4</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>U6</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>U7</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td>U5, P2, P3</td>
<td>P2, P3</td>
</tr>
<tr>
<td>P5</td>
<td>U8</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>U1, P1, P4, P3</td>
<td>P1, P4, P3</td>
</tr>
</tbody>
</table>

Table 3.3: Round 5: Finding phases in the history stack

<table>
<thead>
<tr>
<th>Round</th>
<th>topStack</th>
<th>prevStack</th>
<th>CD(topStack)</th>
<th>CD(prevStack)</th>
<th>children</th>
<th>historyStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U4</td>
<td></td>
<td>4</td>
<td>2</td>
<td>U4</td>
<td>U1, U2, U3</td>
</tr>
<tr>
<td>2</td>
<td>U3</td>
<td>U4</td>
<td>3</td>
<td>4</td>
<td>U4, U5</td>
<td>U1, U2</td>
</tr>
<tr>
<td>3</td>
<td>U2</td>
<td>U1</td>
<td>2</td>
<td>3</td>
<td>U4, U5</td>
<td>U1, P1, U5</td>
</tr>
</tbody>
</table>
3.1. Data Collecting and Preprocessing

Figure 3.5: Round 1 of the phase detection algorithm

Figure 3.6: Round 2 of the phase detection algorithm

Figure 3.7: Round 3 of the phase detection algorithm
3. Approach

Figure 3.8: Round 4 of the phase detection algorithm

Figure 3.9: Round 5 of the phase detection algorithm

Figure 3.10: Round 6 of the phase detection algorithm
3.1. Data Collecting and Preprocessing

Figure 3.11: Round 7 of the phase detection algorithm

Figure 3.12: Round 8 of the phase detection algorithm

Table 3.4: Round 8: Finding phases in the history stack

<table>
<thead>
<tr>
<th>Round</th>
<th>topStack</th>
<th>prevStack</th>
<th>CD(topStack)</th>
<th>CD(prevStack)</th>
<th>children</th>
<th>historyStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U7</td>
<td>U8</td>
<td>3</td>
<td>2</td>
<td>U7</td>
<td>U1, P1, U5, P2</td>
</tr>
<tr>
<td>2</td>
<td>P2</td>
<td>U7</td>
<td>3</td>
<td>3</td>
<td>P1, P2</td>
<td>U1, P1, U5</td>
</tr>
<tr>
<td>3</td>
<td>U5</td>
<td>P2</td>
<td>2</td>
<td>3</td>
<td>P1, P4</td>
<td>U1, P1, P4, U5</td>
</tr>
</tbody>
</table>

In round 8, see Figure 3.12, we again see a drop in the call depth, and we discover two more phases \( P_3 \) and \( P_4 \), see the details in Table 3.4.

In round 9, we reach the end of the trace and again see a drop in the call depth due to the end of trace event placed by the algorithm (see Line 2 of Algorithm 1). In Table 3.5 we again start to pop off the events of the history stack. However, this time, we continue to pop off events until the history stack is empty, because we have reached the end of the trace and there are no more future phases. As a result, we find the root phase of the program, i.e., \( P_6 \).

Table 3.5: Round 9: Finding phases in the history stack

<table>
<thead>
<tr>
<th>Round</th>
<th>topStack</th>
<th>prevStack</th>
<th>CD(topStack)</th>
<th>CD(prevStack)</th>
<th>children</th>
<th>historyStack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>U6</td>
<td>EOF event</td>
<td>2</td>
<td>-1</td>
<td>U6</td>
<td>U1, P1, P6</td>
</tr>
<tr>
<td>2</td>
<td>P3</td>
<td>U8</td>
<td>2</td>
<td>2</td>
<td>P3, P5</td>
<td>U1, P1</td>
</tr>
<tr>
<td>3</td>
<td>P1</td>
<td>P4</td>
<td>2</td>
<td>2</td>
<td>P1, P3, P1</td>
<td>U1</td>
</tr>
<tr>
<td>4</td>
<td>U1</td>
<td>P1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finally, we have discovered six phases in the execution trace and can construct the preliminary phase tree as presented in Figure 3.14.

### 3.2 Model Building

After execution traces have been recorded and each of their preliminary phase trees have been created, we learn a set of primary behaviors exhibited by these traces. This stage of the approach produces two main outputs: (1) a set of hierarchical execution abstractions for each of the input dynamic call trees, and (2) a reusable model that can be used to abstract further execution traces. The model-building stage consists of the following four fundamental steps:

1. **Duplicate phase detection**: Based on the *preliminary phase forest*, we identify the duplicate *preliminary phases* on each layer of the hierarchical structure. In our technique, we create a global key for the nodes of the *preliminary phase trees* based on the layer and the method-invocation orderings. The set of phases outputted in this step are referred to as *unique phases*. This step will be further elaborated in Section 3.2.1.

2. **Phase clustering**: Some *unique phases* perform similar functionality by invoking a similar set of methods. Our technique attempts to cluster the unique phases that are performing similar functionality. The output of this step will be referred to as the *clustered phases*. This step is further elaborated in Section 3.2.2.
3.2. Model Building

3. **Frequent pattern mining**: To reveal functionality units with different granularities that appear across the executions, we perform a technique known as frequent pattern mining (FPM) on each layer of the hierarchical structure of clustered phases. In our technique, we define the mining results as functionality units of the program. The output of this step will be referred to as **frequent pattern phases**. This step is further elaborated in Section 3.2.3.

4. **Semantic labeling**: After we identify the functionality units from the execution traces, we label these functionalities to ease comprehensibility for developers. Generally, the method signatures contain the most important words to describe the main functionality of the methods (as found by De Lucia et al. [11]). We use an information retrieval technique on the method signatures contained in each of **frequent pattern phases**. We treat the method signatures as weighted labels for the **frequent pattern phases**. This step is further elaborated in Section 3.2.4.

In the following sections, we describe each of the steps of model building in more detail.

### 3.2.1 Duplicate Phase Detection

Within a preliminary phase tree, some identical preliminary phases may occur multiple times throughout the forest. To assist developers in understanding the execution, it is important to know which phases accomplish the same functionality because those are the recurrent behaviors of the program and the building blocks of the execution. Moreover, by identifying duplicate phases and thus creating a set of unique phases, the subsequent steps of the approach can be much more efficient.

To identify duplicate preliminary phases, we create a **global key** for each phase, across all hierarchical phase trees (for every execution that we parsed as part of the training). The global key is a representation of each preliminary phase that allows for easy comparison of phase structures. The global key is comprised of: (1) the hierarchical level of the root of its phase tree, and (2) a list of methods, ordered by their first invocation in the sequence of methods called in that phase. For example, a phase that is rooted at hierarchical level 2 that contains the list of methods \( \langle m_a, m_b, m_c, m_a, m_b, m_c \rangle \) would have a global key of “2: m_a, m_b, m_c”.

The motivation for using the order of **first** invocations of methods is to treat repeated sequences (e.g., loops of method calls) of varying lengths to be identified as the same behavior. We can easily take two phases that each read a file with a sequence of the method calls to a **readLine()** method as an example: if these two phases each read files in the same way, but are reading files of different lengths, our approach to building the global key will intentionally identify these two phases as duplicates. Once the global key is generated for all phases, we recreate the hierarchical phase structure for each execution using the global keys to represent the phase nodes.

Let us assume we take a look at the six phases found in Figure 3.15. First, we give every method its own id, giving us Table 3.6. Based on that, we can construct the global keys for each phase, see in Table 3.7. As a result, phases with the the same order of method invocations will have the same global key, see \( P_2 \) and \( P_3 \).
### 3. Approach

Figure 3.15: Phase tree to demonstrate the model building steps

#### Table 3.6: Method name and its corresponding method ID of Figure 3.15

<table>
<thead>
<tr>
<th>Method Name</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse</td>
<td>$M_1$</td>
</tr>
<tr>
<td>scanData</td>
<td>$M_2$</td>
</tr>
<tr>
<td>ReadNextTag</td>
<td>$M_3$</td>
</tr>
<tr>
<td>processSchema</td>
<td>$M_4$</td>
</tr>
<tr>
<td>processElement</td>
<td>$M_5$</td>
</tr>
<tr>
<td>BuildResult</td>
<td>$M_6$</td>
</tr>
<tr>
<td>startElement</td>
<td>$M_7$</td>
</tr>
<tr>
<td>addAttribute</td>
<td>$M_8$</td>
</tr>
<tr>
<td>closeReader</td>
<td>$M_9$</td>
</tr>
</tbody>
</table>

#### Table 3.7: Global keys for the phases found in Figure 3.3

<table>
<thead>
<tr>
<th>Phase</th>
<th>Global Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$2:M_2,M_3,M_4$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$2:M_2,M_3,M_5$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$2:M_2,M_3,M_5$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$3:M_7$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$3:M_8$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$2:M_6,M_7,M_8$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$2:M_9$</td>
</tr>
<tr>
<td>$P_8$</td>
<td>$1:M_1,M_2,M_3,M_4,M_5,M_6,M_7,M_8$</td>
</tr>
</tbody>
</table>
3.2.2 Phase Clustering

The prior step identified the strictly identical preliminary phases across the whole forest. However, the executions may contain similar behaviors that have only slight variations on the method-invocation profile. For example, parsing two lists of tokens that contain mostly similar (but not equal) tokens will likely produce method-call sets that contain most of the same methods but not the exact same sets. This clustering step goes a step further to identify similar phases.

To identify the phases revealing similar functionalities, we applied agglomerative hierarchical clustering (AHC) on the set of unique phases. In this step, there are three fundamental parameters: (1) the threshold to stop the agglomerating, i.e., maxDis; (2) the distance metric, which is used to measure the distance between clusters; and (3) the linkage type that are used to agglomerate the nodes. A threshold is necessary due to not knowing in advance the final number of clusters. As a result, the threshold helps determining when to stop merging clusters.

To measure the distance between phases, we use the Jaccard Distance, which is widely used for comparing the similarity and diversity of data sets, here we use it to measure the similarity between invoked methods among the unique phases. We defined in Jaccard Distance in Equation (3.1) \( P_i \) and \( P_j \) denote the ith and jth phase, and the \( M_i \) and \( M_j \) denote the invoked-method set in the \( P_i \) and \( P_j \) respectively.

\[
DS(P_i, P_j) = 1 - \frac{|M_i \cap M_j|}{|M_i \cup M_j|}
\]  

(3.1)

The agglomerative clustering process iteratively clusters the two most similar (as measured by Equation (3.1)) phase nodes at each clustering level across all executions. When two phases are clustered, the linkage type is defined as the “average linkage,” which will consider the distance from one cluster to another to be equal to the average distance from each element of one cluster to each element of the other cluster.

Agglomerative clustering continues until one cluster has been formed. To terminate the clustering process before that we use predefined threshold \( \text{maxDis} \), which is the maximum distance that we allow to occur between all pairs of phase nodes. As a result, when in one of the iterations of clustering the smallest distance between two clusters exceeds the \( \text{maxDis} \) threshold then the clustering process is terminated.

For example, in Figure 3.3 there are two methods, i.e., scanData and readNextTag, are invoked in both \( P_1 \) and \( P_2 \), thus, the size of the intersection of \( M_1 \) and \( M_2 \) is 2. And as we can see, there are four different methods invoked in \( P_1 \) and \( P_2 \), thus, the size of the union of \( M_1 \) and \( M_2 \) is 4. So, the \( DS(P_1, P_2) = 1 - \frac{2}{4} = 0.5 \).

As soon as we identified the clustered phases, we map the clustered phases back to the unique phases and maintain the hierarchical structure. Furthermore, using the clustering result, for each hierarchical structure layer we compress the consecutive unique phases coming from the same clusters to reduce the method-invocation in the loops. In this way, we reduce the size of the hierarchical phase tree.

So when we have a set of phases and their global keys, then we could cluster similar phases together using the agglomerative clustering. To determine which phases should be
3. Approach

clustered together, we calculate the distance between each phase based on the global keys and cluster the two phases with the smallest distance together. We will repeat this process until the max distance has been reached.

3.2.3 Frequent Pattern Mining

Because developers often invoke methods in patterns to implement functionalities (as described in [1,7,31]), our technique reveals the program’s functionalities by employing a frequent pattern mining technique to identify such functional units. Frequent pattern mining is a family of techniques that seek to efficiently identify frequent patterns within datasets. A particular frequent pattern mining technique is “sequential pattern mining” (SPAM) [5], which is an efficient technique created for discovering frequent sequential patterns from very large transactional databases. The algorithm is especially efficient when the sequential patterns in the database are very long [5]. SPAM employs a depth-first search strategy to generate candidate sequences and implements an a-priori-based pruning mechanism to reduce the search space.

We utilized SPAM to identify and aggregate frequent sequences of behavior that we observed in and across the clustered phase trees. We apply SPAM at each level of the clustered phase trees to identify our highest-level abstraction of execution functionality. For example, if we observe that many executions contain a frequent behavior sequence, such as “isEndOfFile, readChar, appendToList,” we can identify that this sequence represents a functionality that appears to be important for these executions. As such, we can further abstract our phase trees to present these higher-level functionalities.

SPAM utilizes some key parameters that influence the effectiveness and efficiency of our mining technique. The first one is the minimum support value, which we denote as minSup. In frequent pattern mining techniques, support is an indication of how frequently the pattern appears in the database. Formally, as shown in Equation 3.2, the support value of pattern \( P \) with respect to transaction set \( T \) is defined as the proportion of transactions \( t \) in the database that contains pattern \( P \). \( \text{minSup} \) assists in identifying the patterns with frequencies higher than the threshold. The second maximum gap between the items of the patterns, denoted as \( \text{maxGap} \), specifies if gaps are allowed in frequent patterns and the allowed size of those gaps. For example, if \( \text{maxGap} \) is set to 1, no item is allowed, i.e., each consecutive item of a pattern must appear strictly consecutively in a sequence. If \( \text{maxGap} \) is set to \( N \), a gap of \( N-1 \) items is allowed between two consecutive items of a pattern.

\[
\text{support}(\text{Pattern}_i) = \frac{|\{t \in T; \text{Pattern}_i \subseteq T\}|}{|T|}
\]  

Finally, we create a hierarchy of the frequent-pattern-mined phases for each hierarchy level. At each level of the hierarchy, we replace consecutive sequences of repeated frequent-pattern-mined phases with a single phase entity, and hence further reduce the number of phases.
3.2.4 Semantic Labeling

One of the challenges of assisting developers in understanding the software behavior from execution traces is to present the functionality units in a comprehensible way. We accomplish this by creating and applying labels to the final phases in the hierarchy. Many prior works (e.g., [16, 20, 12, 11]) have found that source code contains valuable and meaningful clues for describing functionalities. Our technique utilizes method names as labels that succinctly describe the developers’ intended behavior for the encapsulated functionality. De Lucia et al. [11] found that method signatures provide useful indication of the functionality provided by the methods.

In large programs, each of the high-level frequent-pattern phases may contain hundreds of methods. Simply providing a list of all constituent method names would produce a labels that would be difficult for developers to identify the most relevant operations. As such, we seek to provide the most relevant and distinguishing terms for our phase labels. In order to do so, we adopt the \(\text{TF-IDF}\) (term frequency-inverse document frequency) metric as the weight of our labels \[29\]. Our technique presents the most highly weighted labels to describe the phases.

Next, we treat the method names as the terms and the phases as the documents. In addition, to penalize the utility methods that have a very high frequency in the frequent pattern phases, we adopt log normalization to calculate the term-frequency weight \(tf\). For the inverse-document frequency \(idf\), we adopt the inverse-document-frequency-smooth weighting scheme. The final term frequency-inverse document frequency computation equation is defined in Equation 3.3. After we calculate the weight for each of the labels, we sort them and extract the top \(-X\) as the final label set for each frequent pattern phase.

\[
tf\text{-idf}_{t,d} = (1 + \log(tf_{t,d})) \cdot \log\left(1 + \frac{N}{df_t}\right) \tag{3.3}
\]

3.3 Execution Abstraction

Once the SAGE approach completes the training stage of all prior steps for any set of training executions, a model of all behaviors found in the training executions is available for abstracting any execution (whether seen before or new) so it can be provided to a developer for inspection and comprehension. For the execution under inspection, the phase detection step is first performed followed by duplicate detection. The subsequent steps are performed by utilizing the set of phases identified in the trained model of mined phases. Finally, labeling is done on the phases that are not covered by the model and, as a result, do not have a label yet. Moreover, for the phases covered by the model, the semantic labels are precomputed in the training stage. It should be noted that the training stage of the SAGE approach can be performed infrequently when developer time is not needed and can be trained with a test suite—for example, during an overnight build. The individual execution abstraction is a relatively efficient process, and can be performed quickly when an execution needs to be evaluated. Section 5.2 provides a quantitative evaluation that includes training- and application-stage timings.
An important thing to note here is that the model can only cluster phases and abstract frequent patterns as long as they are covered by the model. The model depends for that on the execution traces given to train the model. If the traces only cover a small subset of the behaviors than the model is less effective. However, due to the phase detection and duplicate detection the phases will still be discovered, but patterns will not be detected as a frequent pattern phase.
Chapter 4

Hierarchical Phase Visualization

SAGE is able to reduce the number of events considerably; however, the output on its own is difficult to comprehend. To assist software developers with comprehending the output of SAGE, a visualization is needed that makes it easy to take navigate through the data structure and give the developer insight into the program’s dynamic behavior. SAGEVIS enables developers to explore the phases within the execution and gain a better understanding of the system and its functional behaviors.

In this chapter, we introduce our visualization, SAGEVIS, to present the hierarchical structure created by SAGE to developers. The goal of SAGEVIS is to present the phases found in the execution trace to developers to assist in software comprehension.

We use the five dimensions of an information visualization introduced by Maletic et al. [21] to characterize SAGEVIS. The different dimensions are task, audience, target, representation, and medium. Each of them characterize a different part the software visualization:

- **Task**: Why is the visualization needed? The overall goal for the tool is to assist developers in software comprehension, e.g., feature comprehension.
- **Audience**: Who will use the visualization? The main audience are software developers or software maintainers who need to gain understanding in the program’s runtime behavior.
- **Target**: What is the data source to represent? The goal is to present the program’s functional behaviors in a hierarchical and chronological way.
- **Representation**: How to represent it? SAGEVIS is an interactive overview of the phases within the execution trace. The user can explore the data at different levels of granularity.
- **Medium**: Where to represent the visualization? The visualization uses the current web standards. It utilizes the D3.js library [6]. As a result, it can be used in most modern browsers, lowering the barrier for developers to use the tool.

The current implementation focuses on assisting developers to comprehend the runtime behavior of a single thread, because multithreaded support is not supported in SAGE. As
4. Hierarchical Phase Visualization

a result, visualizing the interactions between threads is outside of the scope for SAGEVis. However, we will discuss how we envision multithreaded support for SAGEVis in the future in Section 8.2.2.

4.1 Data Model

The visualization was designed with a hierarchical tree structure in mind. In Figure 4.1a, we present an example of a hierarchical tree structure that could be obtained through SAGE. The closer one gets to the leaves of the hierarchical structure the more details will be presented. The phases closer to the root of the tree present the higher level phases within the execution.

In the hierarchical data structure, there is a clear relation between a phase and its sub-phases, as seen in Figure 4.1a. Furthermore, each phase contains a unique ID, execution ID, head method, and a set of labels that further describe the phase. Depending on the data structure it also contains information on which phase is its parent and which are its children. The execution ID is used to determine the order in which the phases were executed.

4.1.1 Breaking the Tree Structure

As mentioned before, the visualization was designed with a hierarchical tree structure in mind, making it possible to explore parts of the trace at a different granularity. However, during the development of SAGE, it became apparent that to minimize the number of phases presented in the visualization we needed to discover the frequent patterns in the execution. The following three options were considered for SAGE and are presented in Figure 4.1:

1. Do not use frequent pattern mining;
2. Use frequent pattern mining, as explained in Section 3.2.3
3. Use frequent pattern mining and draw boundaries.

However, each option has its pros and cons. First, in Figure 4.1a, we present the direct output of SAGE without frequent pattern mining. This gives us a tree of phases, making it possible to unfold parts of the tree to explore different parts of the execution at different levels of granularity. However, the ability to navigate through the tree comes at a trade-off. Loops or behaviors consisting of a sequence of phases add to the number of phases presented to the user. Discovering patterns and presenting them as one phase reduces the overload of information.

Second, in Figure 4.1b, we present the frequent pattern mining. Here, the number of phases are reduced the most out of the three options due to the ability of finding patterns that cross cut the tree structure. This comes at the cost of breaking the tree structure, making it not possible to present different parts of the execution at different levels of granularity.

Third, in Figure 4.1c, we present an example of frequent pattern mining with boundaries. Thanks to the boundaries we retain the tree structure, which is beneficial to developers because it enables displaying different parts of the execution at different levels of granularity at the same time. However, due to retaining the tree structure it becomes impossible to
detect patterns cross cutting the tree structure, resulting in possibly presenting developers with more phases than necessary.

Frequent pattern mining with boundaries or without frequent pattern mining at all would result in a tree structure (see Figure 4.1a or 4.1c), which has a parent-child relation model. The frequent pattern mining data structure however only has a set of layers, as presented in Figure 4.1b. Placing boundaries—finding frequent patterns within a branch—can keep the tree structure in place but limits us finding phases that cross cut the tree structure. Not finding patterns at all can make exploring the trace complicated due to the enormous number of nodes that are presented to the user.

Finally, it is a trade-off between discovering behaviors that cross cut the tree structure and having the tree structure, which enables exploring parts of the trace at different levels of granularity. Both sides can be useful depending on the execution under inspection. We decided to focus on the tree data structure with no frequent pattern mining and the frequent pattern mining structure, presented in Figure 4.1a and 4.1b respectively. By having both options it is possible to still give the developer the option of navigating the visualization at different levels of granularity, and to simplify the timeline view by showing the frequent patterns. Due to limited time, we chose limit the study to these two data structures. In the future, the frequent pattern mining with boundaries could be an option to further explore.

Figure 4.1: Different hierarchical structure based on technique used
4. **Hierarchical Phase Visualization**

4.2 **Shneiderman’s Criteria**

Shneiderman introduced seven tasks to assess the usefulness of a graphical user interface for information visualizations [30]. In Table 4.1, we show how SAGEVIS satisfies most of these criteria, which helped to guide the design of SAGEVIS.

Six out of the seven tasks are supported in SAGEVIS—one task is not supported in the implementation namely history. The history of interactions is currently not being recorded, however, in the future, this could be added by keeping track of the user’s interactions and provide buttons to go back and forth.

Table 4.1: Tasks introduced by Shneiderman to assess the usefulness of a GUI for information visualizations

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>SAGEVIS main view gives an overview of all the phases within the execution trace.</td>
</tr>
<tr>
<td>Zoom</td>
<td>Zooming in on the SAGEVIS main view.</td>
</tr>
<tr>
<td>Filter</td>
<td>Collapsing of phases.</td>
</tr>
<tr>
<td>Details-on-demand</td>
<td>Hovering over a phase highlights similar phases within the trace.</td>
</tr>
<tr>
<td>Relate</td>
<td>Temporal, Hierarchical and functional similarity relationship.</td>
</tr>
<tr>
<td>History</td>
<td>In its current implementation not supported, but can be extended by keeping track of actions that have been performed.</td>
</tr>
<tr>
<td>Extract</td>
<td>The input data of SAGE.</td>
</tr>
</tbody>
</table>

4.3 **SAGEVIS**

The objective of SAGEVIS is to visualize the phases found within the execution trace on different levels of granularity to developers to assist in program comprehension. Given an execution trace that has gone through SAGE, we can use SAGEVIS to demonstrate the discovered phases in the execution trace to the developers.

4.3.1 **Overview**

SAGEVIS is an interactive visualization that gives an overview of the phases within the execution trace. It presents a timeline of the phases within the execution in a hierarchical fashion. The horizontal axis represents the chronology of the phases, where the beginning is on the left. The width of each phase is proportional to the numbers of method invocations contained within the phase. Because some phases are short, it is not possible to determine based on our instrumenter’s information how long the phases took. That is why the number of method invocations was chosen to use as the proportion of width of a phase. As a result, the width does not give a direct representation of time in seconds, but a proportion of method invocations and total number of method invocations in the whole program. The vertical axis is used to extend towards multithreaded applications, more on this in Section 8.2.2. For additional threads, one or more extra timelines can be placed above the main thread.
SAGEVIS needs to be able to visualize the hierarchical data structure at multiple levels. As a result, the visualization has not only a x-axis (time) and y-axis (multiple threads), but also a third axis—depth. To minimize the amount of data that is presented to the users, we only present one level at a time. However, behind each timeline, there can be multiple levels of granularities that can be explored by a developer. Finally, colors are used to assist developers in recognizing similar phases in the visualization.

In Figure 4.2 a mock version of SAGEVIS is given, showing how the visualization for a single thread ideally would look. The top level is the main phase, which in this case is “Compile.” The main phase can contain two or more sub-phases. Each of those sub-phases again could consist of their own sub-phases, etc. This would create a hierarchical structure. Each of those phases can be unfolded, which, in the example, would present us five sub-phases. The arrows in the figure present the boundaries of the phase. The figure also shows us similar phases by presenting them with the same color. For example, the two phases “Code Generation” have the same color, meaning they contain a similar behavior. Upon hovering over one of these phases, the other will be highlighted so similar phases can be easily spotted. By unfolding the higher level phases the internal structure of each phase can be explored without directly presenting the details of all the other phases. As a result, the information can be reduced to what the developer is interested in.

4.3.2 Interactions

Interacting with SAGEVIS to explore the execution trace can be done in four ways: (un)folding phases, zooming, panning, and phase highlighting. In the following sections, we present how each interaction works and how they can assist developers.

Expanding and Hiding Phases

The initial view of SAGEVIS presents the top level of the hierarchical structure. Exploring the different levels of granularity in the hierarchical structure is possible by expanding a phase by clicking on its body, this will present the sub-phases of that phase, see Figure 4.3. Expanding phases enables the developer to choose where she wants to further explore the program, resulting in being presented more fine-grained view of the behavior.
4. **Hierarchical Phase Visualization**

The opposite is also possible; instead of expanding phases, a developer can choose to view a coarser-grained view of a part of the trace by hiding the lower details. This can be accomplished by clicking on the smaller bar above the phase, see Figure 4.4, this hides the sub-phases with the same parent, resulting in presenting only the parent phase.

**Figure 4.3:** Clicking inside the red box unfolds the phase in the timeline

**Figure 4.4:** Clicking inside the red box collapses the phase and its siblings in the timeline
4.3. SageVis

Zooming

Utilizing the hierarchical structure limits the amount of information directly presented to the user. However, when expanding phases to its sub-phases, more and more information is presented to the user. As a result, the width of phases would become too small to obtain any information from them. For this kind of scenario, we added the support to zoom in on the timeline.

Zooming gives the developers the option to zoom in on the relevant information, while retaining the level of granularity for the other phases. In Figure 4.5, we demonstrate how the zooming works. The initial view shows us a phase that parses something. However, the parsing phase is preceded by other phases that are too small to understand. After zooming in we can clearly see that before parsing, a XML parser is created. Subsequently, it provides the reader a data source. After creating the XML parser and setting the data source, the parse phase follows. Without zooming, the user would not have been able to understand the phases preceding the “parse” phase.

![Figure 4.5: Zoom feature demonstrated on a NanoXML visualization](image)

Panning

When a developer has zoomed in on a specific part of the timeline, it is sometimes necessary to contrast the phase currently looked at with the preceding or succeeding phase without having to zoom out again. For this, we added support for panning.

Panning allows developers to move from left to right through the timeline while maintaining the current level of zooming, making it easier for a developer to move quickly through the timeline to find the relevant information. In Figure 4.6, we present how panning
works. In this way, a user can quickly dive into a part of the trace and use panning to move through the time line without having to zoom out.

**Phase Highlighting**

The duplicate detection and clustering of SAGE make it possible to detect similar and duplicate phases as the same. This results in similar phases having the same ID and, in the visualization, the same color. However, when there are a lot of different phases presented in the visualization, it becomes more difficult to distinguish phases from one another.

To make detecting similar phases across the timeline easier, developers can hover over a phase to highlight all the phases with the same global key and cluster ID. In Figure 4.7, we demonstrate how this works. When a developer hovers over the phase “scanSomeTag” the similar phases are highlighted.

Figure 4.6: Panning feature demonstrated on a NanoXML visualization

Figure 4.7: Phase highlighting feature demonstrated on a NanoXML visualization
4.3.3 Visualizing the Frequent Phases Data Structure

The visualization discussed so far is focused on the ideal scenario, which is a hierarchical tree structure as shown in Figure 4.1a. However, the frequent pattern mining SAGE does not produce a tree structure. The frequent pattern mining outputs multiple levels describing the execution trace at different levels of granularity. So if a developer wants a more fine-grained view of the behaviors within the execution trace, it is only possible to show the complete level, instead of only showing parts of the finer-grained level. The lower levels show more details, but become very difficult to comprehend with the large number of frequent patterns phases.

The broken tree structure causes a change in how the visualization works. Most of the interactions stay the same, except for how the folding and unfolding between the different levels works. Due to the broken tree structure, it is not possible to click on one phase and show only its sub-phases, as explained in Section 4.1.1. As a result, when a user wants to see more fine-grained details, she can switch to between levels by clicking on the phase. This will unfold the lower, more fine-grained level timeline of the execution trace.

The lack of a tree structure is inconvenient for the developer, because during exploration of the phases in the trace the developer is faced with a lot of fine-grained information in which she may not be interested. However, the frequent pattern phases that are presented can give a different insight into the program’s behavior. Patterns in the execution can be clustered together. For example, a pattern of behaviors like “readFile,” “parseTokens,” and “createParseTree” could be detected by frequent pattern mining as “processInput.” When visualizing, we can show one long phase instead of three short ones, which aids the comprehension of the program, because there is less information to process.

In Figure 4.8 an example is shown of how the visualization looks for a NanoXML execution trace with and without frequent pattern mining. The first thing that can be noted is the big difference of amount of presented phases between with and without frequent pattern mining. The frequent pattern mining is able to abstract the recurring pattern of scanSomeTag. As a result, the frequent pattern mining clusters the phases together as a frequent pattern phase titled read.

4.4 Multithread Visualization

Currently, SAGEVIS focuses on visualizing single threaded applications. The vertical axis can be used to present additional threads. However, the multithreaded support is limited to showing the additional thread, but not how the threads are interacting with each other. For each thread running simultaneously the visualizations becomes larger, because an additional bar needs to be added for each additional thread. This can lead to scalability problems, especially for programs relying heavily on multithreading. In this area the visualization can be further improved in the future.
4. Hierarchical Phase Visualization

(a) NanoXML visualization without frequent pattern mining

(b) NanoXML visualization with frequent pattern mining

Figure 4.8: Comparison of SAGEVis at the same level for NanoXML with and without frequent pattern mining

4.5 Implementation of SAGEVis

SAGEVis is built around the current web standards, so it uses JavaScript, HTML, SVG, and CSS to present the data to the users. It uses the D3.js library, which can manipulate documents based on data, making it suitable for visualizations based on the web standards.

The architecture of SAGEVis is based on a Model-View-Controller (MVC), because it makes SAGEVis easier to extend with additional views in the future. Those views could, for example, simultaneously be presented and updated when the user interacts with the visualization. Giving the developer multiple viewpoints that complement each other can help the user comprehend the execution of the program.

The model in this case is a hierarchical tree structure created through SAGE. The view is the timeline with phases of the execution. Lastly, the controller connects the two components. When there is an interaction with the visualization, the model gets updated, and after that, the views get updated. Currently, we only have one view, but the design permits multiple views of the same dataset.
Chapter 5

Evaluation

To evaluate our technique to hierarchically abstract execution traces for assisting developer comprehension and evaluation, we implemented SAGE and SAGEVIS, and with it, conducted four studies. We first present a quantitative evaluation on three real programs to assess the benefits and costs for those subjects and executions. Second, we present a user study to investigate to what extent the output of our technique is able to assist developers in accurately identifying functionalities within execution traces. Third, we present a case study to demonstrate the technique and the way in which it is capable of helping developers gain a better understanding of the behaviors performed in a large execution trace. Finally, we present a case study comparing the visualizations with and without frequent pattern mining to assess the pros and cons of frequent pattern mining.

These evaluations are motivated by five research questions:

RQ1: To what extent does SAGE alleviate the information-overload challenge?

RQ2: Does the hierarchical phase abstraction provide substantially different levels of granularities of behavior?

RQ3: What is the computational efficiency of each abstraction step of SAGE?

RQ4: Are the derived labels sufficiently meaningful?

RQ5: What is the impact of frequent pattern mining on the visualization?

5.1 Experimental Design

The experimental design is split into two parts. First, we take a look at the decisions behind choosing the projects for which evaluation, and second, we explain how the model is build and how the execution traces are obtained.

5.1.1 Setup of Experiments

To evaluate the effectiveness of SAGE at reducing the information overload of large execution traces (RQ1) and if the hierarchical phase abstraction provides substantially different
levels of granularity in behavior (RQ2), we conducted a quantitative evaluation on three projects and calculated for each level the average number of phases, standard deviation and other statistics to answer RQ1 and RQ2. Here, we also present the computational efficiency for each step of SAGE (RQ3), see Section 5.2.

For this evaluation, we used three subjects: JAVAC, JEDIT, and NANOXML. Each subject is an open-source Java program that has a high test coverage. Also, the projects differ from one another considerably—JAVAC is a compiler, JEDIT is a text editor, and NANOXML is a XML parser library.

To determine if the labeled phases are comprehensible for developers (RQ4), we surveyed a group of software developers to assess if they could determine the behavior of a phase based on a set of labels, see Section 5.3. We provided labels for one test subject JEDIT, an open-source text editor, with the assumption that most users would be familiar with text editor functionality. JEDIT supports common operations such as typing, keyword highlighting, and search.

To further support the results to answer RQ2 and RQ4, and to demonstrate SAGE and SAGEVIS, we performed a case study on JAVAC, to demonstrate how a user could use the visualization and different levels of granularity to gain an better understanding of the program, see Section 5.4. We chose JAVAC as a subject program because the functionality of a compiler is relatively familiar to software-engineering researchers and has a well known common design with explicit phases.

Finally, we want to determine what the impact of frequent pattern mining on the visualization (RQ5), see Section 5.5. To answer this question, we did a case study and looked at the visualization and how it differs with and without frequent pattern mining. In the evaluation, we will look at three properties of the visualization, namely: navigation, behavior detection, and pattern visualization. For each property, we compare how they perform with and without frequent pattern mining and what are the pros and cons.

For the visualizations, we look at execution traces obtained from two different projects, namely Gson and NANOXML. These projects were chosen because both libraries often use loops to go through the JSON and XML data structures, making them suitable subjects to demonstrate the impact of frequent pattern mining. Also both libraries are open-source, well-documented and well-tested.

5.1.2 Obtaining Traces and Building the Model

The execution traces were obtained for each project in the same way as described in Section 3.1.1. However, there is a difference between the projects in the type of execution traces and the number of traces. For NANOXML, the test suite of the program was used to train the model for SAGE, because of the high test coverage. In the case of JAVAC, we train the model by using 19 test cases provided by the Open JDK project that target the “Warning,” “Assert,” and “Enum” features. Each test case is a small Java program that will be compiled, which makes sure it goes through all the steps of the compiler.

Similar to JAVAC, we used a smaller set of traces for JEDIT and Gson that only cover a smaller subset of the features. For JEDIT we obtained 18 execution traces where a combination of the following behaviors were performed: “New File,” “Change Font,” “Word Count,” “Delete File,” “Cut,” “Copy,” “Paste,” “Undo,” “Redo,” “Search,” “Replace,” “Find,” “Find Next,” “Find Previous,” “Select All,” “Deselect All,” “Open,” “Save,” “Save As,” “Print,” “Print Preview,” “Print Range,” “Print All,” “Close,” “Minimize,” “Maximize,” “Restore,” “Minimize to Tray,” “Maximize to Full Screen,” “Close Window,” “Close All Windows,” “Close This Window,” “Close This Tab,” “Close This Group.”
5.2 Information Overload and Computational Efficiency

“Close&Exit,” “KeywordHighlight,” “Open File,” “Typing,” “Search,” and “Indent.” The resulting traces were used to build the model for the quantitative and behavior identification study.

For GSON, we obtained 10 execution traces where a subset of the functionality of GSON is performed. We chose 10 key behaviors of GSON, e.g., creating a JSON object, parsing a string to JSON object, etc. Using those traces we trained the model.

Finally, in all studies, we used a standard maxDis threshold value of 0.2 for the frequent pattern mining.

5.2 Information Overload and Computational Efficiency

For each of these three subjects (JAVAC, JEDIT, and NANOXML), we computed the hierarchical phase model for all training test-case executions and report the reduction in model size to answer RQ1 and RQ2 respectively. We also report the time costs for each step of our training technique to answer RQ3. Then, we also provide average model sizes at each hierarchical level for execution abstraction during the application stage of our approach, as well as the time cost for abstracting an execution subsequent to the building of the trained model.

5.2.1 Training Results

We present our results for the training stage of our approach for these three subjects in Table 5.1 and 5.2. In Table 5.1, each row shows the results for each subject program, and the columns present the model sizes for each step of the training process. The first four columns provide the project names, the number of test-case execution traces that were used for training our model, the total number of execution events among all execution traces, and the total number of unique methods that were executed across all traces. In the subsequent five columns, we provide the model size (in terms of the total number of identified phases) for each of the steps in our training process (described in Chapter 3). The time cost can found in the same way in Table 5.2.

By examining the results in Table 5.1, we observe that despite the fact that NANOXML is provided with more execution traces (235 versus 19 and 18), the total size of the traces is much smaller, which may not be surprising given the smaller size of the subject and the task that it performs. JAVAC and JEDIT have a relatively small number of execution traces, but their complexity is much larger in comparison to NANOXML.

In Table 5.2, we can observe that the phase detection step for NANOXML requires only 3.5 seconds in comparison to JAVAC and JEDIT, which require 248 and 144 seconds, respectively, which is unsurprising given the differences in trace sizes for these subjects. However, number of events is not directly linked to the number of phases that are found. Even though JAVAC has considerably more phases than NANOXML, JAVAC on the other hand only has a quarter of the amount of phases in comparison to NANOXML. This can be credited to the fact that JAVAC is a compiler, meaning it is structured to run always through certain phases like parse file. As the result, the phase detection algorithm identified 225430, 51878, and 692286 preliminary phases in the top 8 levels of NANOXML, JAVAC,
Table 5.1: Size abstraction for each step of the training procedure

<table>
<thead>
<tr>
<th>Projects</th>
<th>Traces</th>
<th>Events</th>
<th>Methods</th>
<th>Phase Detection [phases]</th>
<th>Duplicate Detection [phases]</th>
<th>Clustering [phases]</th>
<th>Pattern Mining [phases]</th>
</tr>
</thead>
<tbody>
<tr>
<td>NANOXML</td>
<td>235</td>
<td>247,471</td>
<td>100</td>
<td>225,430</td>
<td>786</td>
<td>160</td>
<td>47</td>
</tr>
<tr>
<td>JAVAC</td>
<td>19</td>
<td>22,439,965</td>
<td>3919</td>
<td>51,878</td>
<td>1695</td>
<td>650</td>
<td>57</td>
</tr>
<tr>
<td>JEDIT</td>
<td>18</td>
<td>10,129,771</td>
<td>5431</td>
<td>692,286</td>
<td>17,088</td>
<td>3838</td>
<td>84</td>
</tr>
</tbody>
</table>

Table 5.2: Time cost in seconds for each step of the training procedure

<table>
<thead>
<tr>
<th>Projects</th>
<th>Phase Detection</th>
<th>Duplicate Detection</th>
<th>Clustering</th>
<th>Pattern Mining</th>
<th>Labeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>NANOXML</td>
<td>3.5</td>
<td>0.1</td>
<td>0.5</td>
<td>50.8</td>
<td>1.7</td>
</tr>
<tr>
<td>JAVAC</td>
<td>247.8</td>
<td>0.0</td>
<td>20.1</td>
<td>428.9</td>
<td>1.0</td>
</tr>
<tr>
<td>JEDIT</td>
<td>143.6</td>
<td>0.4</td>
<td>1994.3</td>
<td>5846.0</td>
<td>20.7</td>
</tr>
</tbody>
</table>

and JEDIT, respectively. We limit ourselves to the top 8 levels, because we assume the high-level functionality can be found in the top levels of the program.

Next, we observe that the duplicate-detection step is quite fast—in all cases requiring less than a second. This efficiency is due to the use of the global key, which is then used as a hash for quick look-up. Despite the efficient computation, this step substantially reduces the number of phases, which results in the number of phases being reduced to 0.03%, 3.10%, and 2.47% of the preliminary phases, for NANOXML, JAVAC, and JEDIT, respectively.

After the duplicate detection step, the clustering technique will further reduce the number of phases. However, even though this technique is capable of substantially reducing the number of phases, the time cost of this step is relatively expensive in comparison to the previous two steps due to agglomerative clustering complexity of $N \cdot \log(N)$. This expense is especially noticed for JEDIT, which requires around 33 minutes to cluster the unique phases for the training.

The next step is the frequent pattern mining. We observe this step is the most time-consuming step of the whole training procedure. JEDIT requires around 98 minutes to complete the mining. However, this step has successfully reduced the large event trace into a manageable size (47–84).

It is worth noting that the training process can be performed “offline”—the computational costs can be incurred during off hours (e.g., an overnight build) and can be trained infrequently. As long as the method signatures do not change drastically, the model can be used for subsequent builds. Another important point is that for the frequent pattern mining to work properly, it is important that the patterns are covered by the training model, e.g., the traces obtained through the test suite. Without the coverage, the frequent patterns can not be detected.

5.2.2 Application Results

In Tables 5.4, 5.5, and 5.6, we present our results for applying our SAGE technique to execution traces for creating the hierarchical phase abstraction. In all columns, we present the average results for each of the executions that we abstracted. In addition, we depict the reduction in the number of final phases for each level of the hierarchical abstraction model in Figure 5.1.
Table 5.3 shows the average time cost of abstracting new execution traces, which is quite efficient in practice—requiring an average time of 13 seconds or less. The reason for this efficiency is twofold: (1) the model training procedure is an off-line process, which can be done at any time; and (2) after the model is built, the time cost of labeling a new execution trace consists of only three steps: phase detection, duplicate detection, and labeling. As discussed in the training results, the efficiency bottleneck of technique is in the steps of clustering and pattern mining, which consume more than 95% of the total time cost.

Based on Tables 5.4, 5.5, and 5.6, it is clear that the top level of our hierarchical structure is capable of highly abstracting the execution traces: the NANOXML execution abstractions contain an average of 3.5 phases, the JAVAC execution abstractions contain an average of 2.0 phases, and the JEDIT execution abstractions contain an average of 19.7 phases. The big difference in average amount of phases on the top level (and lower levels) is twofold. First, when we look at Table 5.1 we notice that JEDIT in comparison with NANOXML and JAVAC has more than 3 and 14 times the number of phases discovered by the phase detection respectively. When we look at the clustered phases JEDIT still has 20 and 5 times more phases then NANOXML and JAVAC respectively. Due to the high amount of clustered phases, the resulting number of frequent phases can be expected to be considerably higher than the other two test subjects.

Second, as explained in Section 5.1.2, the traces looked at for JEDIT are vastly different traces than the ones for NANOXML and JAVAC. The traces for NANOXML are all unit tests of the program, making them all relatively small in size with limited number of patterns. JAVAC’s execution traces all follow the same pipeline, and because of that, the number of frequent phases can be easily reduced, because every time a file is compiled it goes through the same steps. However, in the case of JEDIT, the execution traces contain per trace multiple different behaviors that are executed plus interactions with the GUI that result in a higher number of phases overall, see Table 5.1. At the top level, the frequent pattern mining is able to reduce the number of phases considerably, because high level behaviors can be more easily detected, e.g., clicking on the GUI, go to preference, and then change font. However, with each level, the increasing number of phases makes it more difficult finding frequent patterns.

Figure 5.1 graphically depicts the reduction in execution abstraction from the full execution trace (drawn in blue) and the number of execution phases at each of the top eight hierarchical levels. This figure is drawn in log scale to enable the final average number of phases to be visible in the same plot as the execution trace size. Note that the execution trace size is multiple orders of magnitude larger than all levels of the hierarchical execution phase model.

SAGE is able to reduce the amount of information considerably and provide per levels a gradually increase in levels. As a result alleviate the information-overload a developer would experience. Finally, the cost of building the model does take up a considerable amount of time, but the abstracting of a single execution trace can be done quite efficiently.
5. Evaluation

Table 5.3: Total time cost for execution abstraction

<table>
<thead>
<tr>
<th>Project</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NANOXML</td>
<td>0 s</td>
</tr>
<tr>
<td>JAVAC</td>
<td>13 s</td>
</tr>
<tr>
<td>JEDIT</td>
<td>9.1 s</td>
</tr>
</tbody>
</table>

Table 5.4: Distribution of the phases per level in NANOXML

<table>
<thead>
<tr>
<th>Level</th>
<th>Average</th>
<th>Median</th>
<th>Max</th>
<th>Min</th>
<th>Std</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.5</td>
<td>3.0</td>
<td>20</td>
<td>1</td>
<td>2.6</td>
<td>4.9</td>
<td>27.6</td>
</tr>
<tr>
<td>2</td>
<td>5.6</td>
<td>5.0</td>
<td>32</td>
<td>1</td>
<td>4.6</td>
<td>3.4</td>
<td>16.2</td>
</tr>
<tr>
<td>3</td>
<td>12.0</td>
<td>6.0</td>
<td>38</td>
<td>1</td>
<td>10.8</td>
<td>0.7</td>
<td>−1.0</td>
</tr>
<tr>
<td>4</td>
<td>12.5</td>
<td>6.0</td>
<td>42</td>
<td>1</td>
<td>11.7</td>
<td>0.8</td>
<td>−0.8</td>
</tr>
<tr>
<td>5</td>
<td>19.4</td>
<td>15.0</td>
<td>90</td>
<td>2</td>
<td>15.6</td>
<td>1.4</td>
<td>2.7</td>
</tr>
<tr>
<td>6</td>
<td>23.8</td>
<td>19.0</td>
<td>140</td>
<td>2</td>
<td>19.1</td>
<td>2.6</td>
<td>13.0</td>
</tr>
<tr>
<td>7</td>
<td>30.3</td>
<td>24.0</td>
<td>269</td>
<td>2</td>
<td>32.5</td>
<td>5.1</td>
<td>34.3</td>
</tr>
<tr>
<td>8</td>
<td>43.5</td>
<td>41.5</td>
<td>414</td>
<td>3</td>
<td>50.4</td>
<td>4.6</td>
<td>28.7</td>
</tr>
</tbody>
</table>

Table 5.5: Distribution of the phases per level in JAVAC

<table>
<thead>
<tr>
<th>Level</th>
<th>Average</th>
<th>Median</th>
<th>Max</th>
<th>Min</th>
<th>Std</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
<td>2.0</td>
<td>2</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
<td>−3.0</td>
</tr>
<tr>
<td>2</td>
<td>3.0</td>
<td>3.0</td>
<td>3</td>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>−3.0</td>
</tr>
<tr>
<td>3</td>
<td>3.0</td>
<td>3.0</td>
<td>3</td>
<td>3</td>
<td>0.0</td>
<td>0.0</td>
<td>−3.0</td>
</tr>
<tr>
<td>4</td>
<td>4.6</td>
<td>5.0</td>
<td>5</td>
<td>4</td>
<td>0.5</td>
<td>−0.3</td>
<td>−1.9</td>
</tr>
<tr>
<td>5</td>
<td>8.5</td>
<td>9.0</td>
<td>10</td>
<td>7</td>
<td>1.1</td>
<td>−0.4</td>
<td>−1.3</td>
</tr>
<tr>
<td>6</td>
<td>10.3</td>
<td>10.0</td>
<td>15</td>
<td>7</td>
<td>1.8</td>
<td>0.8</td>
<td>0.3</td>
</tr>
<tr>
<td>7</td>
<td>28.5</td>
<td>16.0</td>
<td>86</td>
<td>11</td>
<td>21.7</td>
<td>1.3</td>
<td>0.5</td>
</tr>
<tr>
<td>8</td>
<td>93.6</td>
<td>38.0</td>
<td>375</td>
<td>20</td>
<td>98.9</td>
<td>1.5</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 5.6: Distribution of the phases per level in JEDIT

<table>
<thead>
<tr>
<th>Level</th>
<th>Average</th>
<th>Median</th>
<th>Max</th>
<th>Min</th>
<th>Std</th>
<th>Skewness</th>
<th>Kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.7</td>
<td>16.0</td>
<td>88</td>
<td>6</td>
<td>17.5</td>
<td>3.2</td>
<td>9.9</td>
</tr>
<tr>
<td>2</td>
<td>109.6</td>
<td>96.0</td>
<td>252</td>
<td>25</td>
<td>66.0</td>
<td>0.7</td>
<td>−0.6</td>
</tr>
<tr>
<td>3</td>
<td>481.2</td>
<td>381.0</td>
<td>1695</td>
<td>206</td>
<td>359.9</td>
<td>2.4</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>834.8</td>
<td>723.5</td>
<td>2338</td>
<td>424</td>
<td>457.1</td>
<td>2.1</td>
<td>4.2</td>
</tr>
<tr>
<td>5</td>
<td>1599.8</td>
<td>1319.5</td>
<td>4375</td>
<td>731</td>
<td>865.5</td>
<td>2.2</td>
<td>3.9</td>
</tr>
<tr>
<td>6</td>
<td>2633.3</td>
<td>2248.0</td>
<td>6302</td>
<td>1153</td>
<td>1200.8</td>
<td>1.6</td>
<td>2.4</td>
</tr>
<tr>
<td>7</td>
<td>4642.2</td>
<td>4913.5</td>
<td>8340</td>
<td>2200</td>
<td>1646.7</td>
<td>0.3</td>
<td>−0.5</td>
</tr>
<tr>
<td>8</td>
<td>6955.8</td>
<td>7071.5</td>
<td>11,804</td>
<td>2673</td>
<td>2741.9</td>
<td>0.1</td>
<td>−0.9</td>
</tr>
</tbody>
</table>
5.3 Behavior Identification Based on Generated Labels

To evaluate the degree to which our approach provides comprehensible meanings of actual behavior of execution phases (RQ4), we conducted a user study. To establish that all of the participants have an adequate understanding of the test program’s functionality, but have no idea of its implementation, we chose JEDIT as the subject program, because JEDIT is a text editor and most users would be familiar with text editor functionality. JEDIT supports common operations such as typing, keyword highlighting, and search.

The experiment consists of four steps. First, we identified nine basic functionalities of JEDIT, which are “New File,” “Change Font,” “Word Count,” “Close&Exit,” “Keyword-Highlight,” “Open File,” “Typing,” “Search,” and “Indent.” Based on the basic functionality set, we collected a trace with each behavior performed in isolation. By “in isolation” we mean that after the program starts up, only one behavior is performed, and then the program exits. After applying our technique on these traces to train the model, we were able to find and remove the initialization phase and finalization phases, because they were found in all results across all basic functionalities at the beginning and end of the execution and we are only interested in the pre-defined behaviors in isolation. One point worth noting is that for many programs, the ground truth of the internal behaviors would be unavailable and unknowable. However, because JEDIT is a GUI program with simple functionality, we can identify basic functionalities for this evaluation.

Secondly, to extract the experimental behaviors, we conducted 18 compound behaviors based on the combination of some of the 9 selected behaviors, as explained in Section 5.1.2. The resulting traces were processed by our technique. In the end, for each trace a set of phases was found.

Third, to determine a ground truth for each of the unknown phases found in the 18 compound behaviors, we compared the invoked methods of them with the isolated phases’ invoked methods set. If the similarity is high, we set ground truth behavior of the unknown phases as the basic functionality of the isolated phase. If the invoked methods set is not close to any isolated phases, we set the ground truth to “Unknown.”

Finally, to evaluate the accuracy of the phase comprehension, we generated two questionnaires with 15 multiple-choice questions each, see Appendix C. Each question presents
the top 15 labels from one of the discovered phases. The participants were asked to choose from a set of five options which behavior the given label-set describes. Beside one correct option, three options are randomly selected from the basic functionality set, and the fifth one is always the “Unknown” option.

We recruited 28 participants, all of whom were graduate students whose majors are computer science or software engineering. All the participants had neither used JEDIT nor read its source code. The participants were required to finish the questionnaire within 30 minutes, this would give 2 minutes per question to determine its behavior. The time constraint was added so developers had enough time to go through the presented labels, but also not enough to overthink it. In this way, we wanted to see how well the labels could inform a software engineer without knowledge of the behaviors they were looking at.

We present the comprehension accuracy of each behavior in Table 5.7. In this table, the first, second, and third columns represent the functionality studied, the number of responses per functionality, and the number of correct responses. The fourth column represents the accuracy that we found for that functionality. Based on these results, we observe that the accuracy values ranges widely, i.e., from 0.39 to 0.86, and the average accuracy is 0.70.

We further investigate the reasons by talking with some of the participants face to face. They thought the labels of “New File,” “Close&Exit,” “Open File” and “Indent” functionalities are pretty clear and comprehensible for them. That is because the top 5 labels of the identified phases contains the keywords revealing the functionality, such as “NewFile,” “Close,” “Open,” and “shiftIndentRight.” Furthermore, we also investigate the reasons for the low accuracy of “Keyword Highlight” and “Word Count” functionalities. We found the most meaningful label of the “Keyword Highlight” functionality is the “KeywordMap.” However, a large number of the participants did not know the meaning of this term, so they chose the “Unknown” option as the answer. As for the “Word Count” functionality, we found its representative labels were “showWordCountDialog” and “doWordCount,” but those labels had a relatively low TF-IDF value and were thus not ranked highly among the labels (ranked 13 and 14, out of 15 labels). As such, the participants failed to notice them. Ultimately, TD-IDF shows that, for the majority of the behaviors, it is suitable for developers to match the correct behavior to the set of labels. However, for some behaviors, the important labels are ranked low, limiting the users to comprehend the set of labels properly. The full list of labels and their rank for each behavior can be found in Table B.1 in the appendix.

5.4 Case Study on Javac

In this case study, we apply our technique to JAVAC\[1\], an open-source Java compiler, to demonstrate our technique. The goal is to demonstrate how the visualization and labels can assist developers gain a better understanding of how a project works (RQ4), in this case JAVAC, and to demonstrate that the hierarchical structure shows different levels of granularity (RQ2). We especially want to find the high-level behaviors that a compiler steps through to compile a Java source file to class file.

\[1\]http://openjdk.java.net/
To train the model, we used the 19 test cases provided by the OpenJDK project that target the “Warning,” “Assert,” and “Enum” features. The total number of executed methods was 3,919. The combined size of the 19 trace files was 995 megabytes. In total, the trace files contained more than 20 million trace events combined—far too large for developer comprehension, see Table 5.1.

Using this trained model, we can demonstrate the output for an execution that we wish to abstract. The test case Serial.java is a file for JAVAC to compile. We produced the execution trace for JAVAC when compiling the Serial.java source-code file. The execution trace for this specific execution contains 1,206,388 events.

Visualizing the hierarchical tree structure without frequent pattern mining allows us to easily navigate through the visualization by only unfolding the parts of the execution in which we are interested. In this case, the goal is to find which major phases are executed by the compiler and if that aligns with what the compiler actually does.

In step 1 of Figure 5.2, we demonstrate the top-level through SAGEVIS without the frequent pattern mining. We observe that the JAVAC execution for compiling Serial.java is unsurprisingly dominated by a compile phase in the top most-abstracted level. When we unfold compile its sub-phases are presented, see step 2 in Figure 5.2, we observe that the compile phase can be decomposed into multiple sub-phases that account for constituent behaviors. Those behaviors can be split into two main parts (1) parsing the files and processing the annotations, and (2) desugaring and code generation. First, a large phase enterTrees is presented. Left from that phase is a short phase parseFiles and right from enterTrees the processingAnnotations phase can be found. This presents the first part of a program compilation.

Second, the phase right from processAnnotations is the compile2 phase, which should present the the “desugaring and code generation” part. Upon zooming in on compile2 (step 3) and unfold to further inspect the phase, demonstrated in step 4, we discover that it consists of a repetition of four phases: attribution, flow, desugar, and finally code generation. When zooming in further (step 5), these four phases become better visible. This aligns with the second part of the compile phase, which accounts for the desugaring and code generation.

To train the model, we used the 19 test cases provided by the OpenJDK project that target the “Warning,” “Assert,” and “Enum” features. The total number of executed methods was 3,919. The combined size of the 19 trace files was 995 megabytes. In total, the trace files contained more than 20 million trace events combined—far too large for developer comprehension, see Table 5.1.

Using this trained model, we can demonstrate the output for an execution that we wish to abstract. The test case Serial.java is a file for JAVAC to compile. We produced the execution trace for JAVAC when compiling the Serial.java source-code file. The execution trace for this specific execution contains 1,206,388 events.

Visualizing the hierarchical tree structure without frequent pattern mining allows us to easily navigate through the visualization by only unfolding the parts of the execution in which we are interested. In this case, the goal is to find which major phases are executed by the compiler and if that aligns with what the compiler actually does.

In step 1 of Figure 5.2, we demonstrate the top-level through SAGEVIS without the frequent pattern mining. We observe that the JAVAC execution for compiling Serial.java is unsurprisingly dominated by a compile phase in the top most-abstracted level. When we unfold compile its sub-phases are presented, see step 2 in Figure 5.2, we observe that the compile phase can be decomposed into multiple sub-phases that account for constituent behaviors. Those behaviors can be split into two main parts (1) parsing the files and processing the annotations, and (2) desugaring and code generation. First, a large phase enterTrees is presented. Left from that phase is a short phase parseFiles and right from enterTrees the processingAnnotations phase can be found. This presents the first part of a program compilation.

Second, the phase right from processAnnotations is the compile2 phase, which should present the the “desugaring and code generation” part. Upon zooming in on compile2 (step 3) and unfold to further inspect the phase, demonstrated in step 4, we discover that it consists of a repetition of four phases: attribution, flow, desugar, and finally code generation. When zooming in further (step 5), these four phases become better visible. This aligns with the second part of the compile phase, which accounts for the desugaring and code generation.

Table 5.7: The user study results of JEdit

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Handouts</th>
<th>Correct</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>New File</td>
<td>56</td>
<td>46</td>
<td>0.82</td>
</tr>
<tr>
<td>Change Font</td>
<td>28</td>
<td>21</td>
<td>0.75</td>
</tr>
<tr>
<td>Word Count</td>
<td>56</td>
<td>28</td>
<td>0.50</td>
</tr>
<tr>
<td>Close&amp;Exit</td>
<td>70</td>
<td>56</td>
<td>0.80</td>
</tr>
<tr>
<td>Keyword Highlight</td>
<td>28</td>
<td>11</td>
<td>0.39</td>
</tr>
<tr>
<td>Open File</td>
<td>28</td>
<td>22</td>
<td>0.79</td>
</tr>
<tr>
<td>Typing</td>
<td>70</td>
<td>52</td>
<td>0.74</td>
</tr>
<tr>
<td>Search</td>
<td>56</td>
<td>36</td>
<td>0.64</td>
</tr>
<tr>
<td>Indent</td>
<td>28</td>
<td>24</td>
<td>0.86</td>
</tr>
<tr>
<td>Total</td>
<td>420</td>
<td>296</td>
<td>0.70</td>
</tr>
</tbody>
</table>
Figure 5.2: Exploring the abstracted execution trace of JAVAC with SAGEVis
A point worth noting is that at step 4 of the visualization we see a recurrent phase of desugar, however, not all the desugar phases are given the same color. This can have two reasons: (1) they are different phases, or (2) they are not detected as a similar phase even though they should be. A reason for this problem can be due to the max distance threshold for the agglomerative hierarchical clustering.

Using SAGEVIS without the frequent pattern mining allows us to find the high-level behaviors that are performed by the compiler. It starts with parsing the files and processing annotations to continue with data flow checks, desugaring, and finally generating the bytecode.

In Table 5.8, we present a set of labels generated for the phases parseFiles, enterTrees, compile2, and compile. We chose to look at those phases because of two reasons: (1) parseFiles, enterTrees, and compile2 are all children of compile, so we expect an overlap in labels between compile and the other three phases, and (2) compile is the largest and most prominent phase in the execution trace, so that phase and its sub-phases are most interesting to examine.

The labels of parseFiles describe clearly what this phase does: it parses and tokenizes all the files. On the other hand, enterTrees is more complex to understand: it enters a tree and visits the different tokens, e.g., class, method, or variable definitions. However, next to that, the labels do not clearly describe what is done upon visiting these definitions. The labels of compile2 seem to describe the behavior well with labels such as analyze and translate.

To check if the labels indeed describe the behavior well, we turn to the source code of the program. Here, the following description are given for compile and compile2:

**compile:** “Compile a list of files, return all compiled classes.”

**compile2:** “The phases following annotation processing: attribution, desugar, and finally code generation.”

Based on this, we can conclude that the labels for compile2 explain the behavior well with labels such as analyzeTree and translate. Knowing now that the phases before needed to process the annotations, the enterTrees labels also make sense with visiting the method definitions.

A point worth noting, is that important labels of the sub-phases like JavaTokenizer and parseType for parseFiles, enterAnnotation for enterTrees, and translate and analyzeTree for compile2 do not appear in compile even though these labels describe the sub-phases.

Finally, we abstracted an incomprehensible execution trace containing 1,206,388 events to a hierarchical tree structure where we showed it is possible to determine the high-level behaviors of JAVAC, and that those behaviors break down in smaller sub-phases, resulting in a better understanding of the program thus answering RQ2 that it provides substantially different levels of granularities of behavior. We also showed that the generated labels can help developers gain a better insight in the behavior of the phase thus answering RQ4 that the labels are sufficiently meaningful. However, even though the labels assist in further explaining what the phases are doing, the labels are not enabling developers to derive which sub-phases are within a higher-level phase.
5. Evaluation

Table 5.8: Labels belonging to phases of JAVAC

<table>
<thead>
<tr>
<th>Head Method</th>
<th>parseFiles</th>
<th>enterTrees</th>
<th>compile2</th>
<th>compile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label 1</td>
<td>DiagnosticSource</td>
<td>visitClassType</td>
<td>visitMethodType</td>
<td>instance</td>
</tr>
<tr>
<td>Label 2</td>
<td>JavaTokenizer</td>
<td>read</td>
<td>visitClassType</td>
<td>getTag</td>
</tr>
<tr>
<td>Label 3</td>
<td>UnicodeReader</td>
<td>complete</td>
<td>visitBlock</td>
<td>accept</td>
</tr>
<tr>
<td>Label 4</td>
<td>JavacParser</td>
<td>getTag</td>
<td>visitApply</td>
<td>visitClassType</td>
</tr>
<tr>
<td>Label 5</td>
<td>block</td>
<td>accept</td>
<td>translate</td>
<td>accepts</td>
</tr>
<tr>
<td>Label 6</td>
<td>modifiersOpt</td>
<td>toString</td>
<td>visitExec</td>
<td>toString</td>
</tr>
<tr>
<td>Label 7</td>
<td>parseType</td>
<td>visitType</td>
<td>write</td>
<td>visitMethodDef</td>
</tr>
<tr>
<td>Label 8</td>
<td>toP</td>
<td>accepts</td>
<td>analyzeTree</td>
<td>visitClassDef</td>
</tr>
<tr>
<td>Label 9</td>
<td>nextToken</td>
<td>resolveQualifiedMethod</td>
<td>visitMethodDef</td>
<td>read</td>
</tr>
<tr>
<td>Label 10</td>
<td>Errorous</td>
<td>isAccessible</td>
<td>accept</td>
<td>visitType</td>
</tr>
<tr>
<td>Label 11</td>
<td>literal</td>
<td>visitMethodDef</td>
<td>visitClassDef</td>
<td>hasNext</td>
</tr>
<tr>
<td>Label 12</td>
<td>term</td>
<td>isErrorous</td>
<td>toString</td>
<td>get</td>
</tr>
<tr>
<td>Label 13</td>
<td>storeEnd</td>
<td>accessS$000</td>
<td>visitVarDef</td>
<td>next</td>
</tr>
<tr>
<td>Label 14</td>
<td>token</td>
<td>visitClassDef</td>
<td>reset</td>
<td>complete</td>
</tr>
<tr>
<td>Label 15</td>
<td>to</td>
<td>enterAnnotation</td>
<td>getTag</td>
<td>visitMethodType</td>
</tr>
</tbody>
</table>

5.5 Impact of the Frequent Pattern Mining

In this section, we take a close look at how the visualization with and without frequent pattern mining differ from each other and examine the usefulness of the visualization (RQ5). We will focus on three different aspects of the visualization, namely: navigation, detecting behaviors, and abstracting patterns. Based on these three points, we will look at the pros and cons of frequent pattern mining. Finally, we present the difference using visualizations with and without frequent pattern mining for multiple projects.

5.5.1 Navigation

SAGEVIS enables the developer to explore the execution trace on different levels of granularity. Without frequent pattern mining, it is possible to dive deeper into a specific part of the trace, as presented in Figure 5.3a. The left part of the execution trace is on a coarse-grained level of granularity, while the right side has zoomed in to lower levels. With frequent pattern mining this is not possible, as seen in Figure 5.3b. The moment we look at finer-grained levels, we have to do that for the complete level.

Although losing the ability to pick and choose at what granularity which part of the trace is presented is unfortunate, the frequent pattern mining does help to keep the number of phases that are added per level to a minimum. When comparing Figure 5.3a and 5.3b, we see that phases on the right side in Figure 5.3b some phases merged together as a frequent pattern. As a result, the developer is presented less phases directly, making the timeline less cluttered.

An interesting thing to note is that the largest frequent phase in Figure 5.3b has a different head method then the phase around the same moment in the tree structure, see Figure 5.3a. The first phase contained in the frequent pattern is a phase with the head method checkNotNull; in the Gson library, this is a precondition to check if a variable is not null and is often used within toJson. So, due to looking at a lower level of granularity, the frequent pattern phase describes an internal behavior of the phase. In Figure 5.4b we present
5.5. Impact of the Frequent Pattern Mining

(a) Gson without frequent pattern mining and fine-grained details on right side.

(b) Gson visualization with frequent pattern mining at finer-grained level.

Figure 5.3: Gson visualization demonstrating navigating

same visualization with frequent pattern visualization at a higher level, where it directly becomes clear that the actual behavior toJson is.

5.5.2 Behavior Detection

In Figure 5.4 we present two visualizations of a Gson execution trace, one with and one without frequent pattern mining. The reason for choosing Gson is that the program is an open-source, well-documented, and well-tested library. The goal of the visualization is to assist developers in giving them an understanding about what is going in the code. When looking at the original test, it was performing the following main behaviors:

1. Convert a Java object to JSON string;
2. Parse a JSON string to a Java object;
3. Parse a Java object to a JSON element object.

When looking at the hierarchical structure, see Figure 5.4a the three main behaviors can easily be picked out: (1) toJson, (2) fromJson, and (3) ToJsonTree. However, for the visualization with frequent pattern mining, see Figure 5.4b this is not that simple. The top level does show us one behaviors in the top level, namely behavior (1) toJson. The other two main behaviors fromJson and ToJsonTree seem to have been merged together with toJson as a frequent pattern. As a result, only one big phase is presented even though it contains three separate behaviors.
5. Evaluation

(a) Visualization without frequent pattern mining.

(b) Visualization with frequent pattern mining.

Figure 5.4: Gson visualization presenting the three high-level behaviors

However, even though the individual behaviors are not directly presented to the developer, it is clear in both visualization that at some form of data is transformed into JSON format.

5.5.3 Visualizing Patterns

Sometimes programs have recurring patterns of phases. Visualizing this will cause the timeline of SAGEVIS to be cluttered. In SAGE, the frequent pattern mining step was introduced to counter this problem by detecting frequent patterns and abstract them away as a frequent pattern phase. In this section, we will take a look at how frequent pattern mining impacts the visualization.

To determine the impact of frequent pattern mining on the visualization, we will look at an execution trace of NANOXML. This project was chosen, because the main goal of the library is to read, write, and manipulate XML files. These operations are often done in a repetitive manner, so we expect to see patterns of phases to occur. In Figure 5.5, we present, at the top, the tree structure visualization and, at the bottom of the figure, the visualization using frequent pattern mining. First, we can note that the bottom visualization contains less phases than the top level. This is because of the frequent pattern mining, which clusters phases together as one phase when they occur as a frequent pattern.

Second, the frequent pattern mining is able to detect the pattern and create a frequent pattern phase. This simplifies the view considerably and gives a clearer representation of what happens in the execution, which can help developers to comprehend the execution better. In the top visualization, it becomes clear that there is some form of recurrent pattern of *scanSomeTag* and then a shorter phase. However, the overall phase is unclear from the
5.5. Impact of the Frequent Pattern Mining

Figure 5.5: NanoXML visualization with (top) and without (bottom) frequent pattern mining.

top visualization. The frequent pattern version does give directly an indication of what is happening in the execution, because it clusters the recurrent pattern of \textit{scanSomeTag} and a short phase as a larger phase, \textit{i.e.}, \textit{read}. However, even though it is correct to detect the recurrent pattern of \textit{scanSomeTag}'s phases as a phase which reads a XML file, misses to add the next set of phases to it as well. The \textit{addPCData} phase is also still part of reading the XML file and should have been part of the \textit{read} phase.

Ultimately, detecting patterns in the visualization can give the developer a quick idea of the high-level behaviors of the program. However, the frequent pattern mining did fail to detect the complete pattern as a phase, because it left out the \textit{addPCData} phase and the phases before that.
Chapter 6

Discussion

In this chapter, we discuss the evaluation results and answer the five research questions introduced in Chapter 5.

6.1 Addressing Information Overload

To address research question RQ1, we assessed if our technique was successful in abstracting the overwhelming amount of information presented in the execution trace, and thus reducing the information overload that a developer would face if attempting to understand an execution trace. In Table 5.1, we found that the number of meaningful execution events was reduced in each step during the training stage of the approach. Moreover, when processing an execution for developer inspection (results shown in Table 5.4, 5.5, and 5.6), we found multiple orders of magnitude reduction in trace sizes, regardless of subject program and hierarchical level for the top 8 levels. Next to the reduction in size, we also see that the phases discovered in the execution trace of JAVAC do present the behaviors that you would expect to occur. As such, we posit that the approach substantially alleviates the information-overload difficulty for developer comprehension of execution traces.

6.2 Addressing Behavior Subsumption

To address research question RQ2, we observe the execution behavior phases in the JAVAC case study, as well as differences in the number of phases found at each of the top 8 hierarchical levels of Tables 5.4, 5.5, and 5.6, and in Figure 5.1. In practice, different maintenance tasks require different understanding of the program. However, existing work in source-code summarization and execution-trace abstraction are typically inflexible to produce phase descriptions at multiple levels of abstraction. As shown in the JAVAC case study, SAGE is able to build up the hierarchical structure and present the comprehensible functionality units at multiple levels of granularity. The top layer of the JAVAC’s hierarchical structure presents an overview of the compiling procedure, which consists of only two highly abstract functionality units. The labels in the top level are not descriptive enough to
describe all of its functionalities. However, exploring the hierarchy further will reveal its sub-behaviors.

In the results of the Quantitative Study (Section 5.2), in Tables 5.4, 5.5, and 5.6 and Figure 5.1, we see the average number of execution phases varies substantially from Level 1 to Level 8. For NANOXML, the average number of identified behavior phases at Level 1 is 3.5, whereas the average number of behavior phases at Level 8 is 43.5. For JAVAC, the average number of identified behavior phases at Level 1 is 2.0, whereas the average number of behavior phases at Level 8 is 93.6. For JEDIT, the average number of behavior phases at Level 1 is 19.7, whereas the average number of behavior phases at Level 8 is 6955.8.

As such, based on both the case study and the quantitative results, we see a substantial difference of levels of abstraction for behaviors and their sub-behaviors at the varying levels of the hierarchy, which would allow developers to investigate and explore features and functionalities to support their maintenance tasks.

6.3 Addressing Comprehensibility of Execution Traces

To address RQ4, we need to determine if the technique can help developers understand the executed behaviors based on the labels provided by the technique.

Execution traces are often incomprehensible, because of their massive size and fine-grained events. In the case study, we demonstrated how a user could use the technique to help them gain a better understanding of an execution trace by utilizing the multiple levels of granularity.

Still, even with the different levels of granularity it may be still difficult for developers to determine what behavior actually occurred on that level of abstraction. In the user study, we demonstrated that the labeled phases enabled developers to comprehend what the behaviors of the execution. For 6 out of the 9 studied functionalities, the users achieved accuracy scores of 70% or greater. Moreover, overall across all 9 studied functionalities, users achieved accuracy of 70%. Hence, we posit that the approach assists with the comprehensibility of understanding execution trace events, however for some behaviors the technique produced better results than for others.

6.4 Computational Efficiency

To address RQ3, we timed for three projects the time cost to train the model and to abstract a single execution after the model has been trained. In Table 5.2, we present the time cost for the training phase and notice that the most time is spent in the frequent pattern mining phase. The training stage can take a long time to complete; however, the time cost to abstract a single execution trace is done in matter of seconds, see Table 5.3. The training stage can be performed “offline”—computational costs can be incurred during off hours (e.g., an overnight build) and can be trained infrequently.
6.5 Advantages and Limitations of Frequent Pattern Mining

To address RQ5, we needed to compare SAGEVIS with and without frequent pattern mining to determine the impact it has on breaking the hierarchical tree structure.

SAGEVIS is a intuitive way to explore an execution trace by utilizing the different levels of granularity. By using zooming and dragging a user can go through large execution traces and still get the amount of details they are interested in. The frequent pattern mining provides two ways of presenting the data to the user, each of them with their own pros and cons. Without frequent pattern mining, the visualization can be build around the tree structure. This gives the user the ability to choose which part of the visualization they want to explore in more depth, while maintaining the level of granularity for the other parts of the visualization. However, because of the lack of the frequent pattern mining, it can happen that one phase has many sub-phases, even though among these sub-phases patterns can be found. This can result in a cluttered timeline that is difficult to comprehend.

The frequent pattern visualizations solves this problem by detecting those patterns can and cluster them together. As a result, we reduce the information directly presented to the developer, making the timeline easier to understand. The downside for detecting the frequent phases is that the frequent pattern mining does break the tree structure. As a result, the developer ability to view parts different parts of the execution trace different levels of granularity is lost. They can only switch between the complete levels of granularity and not cherry-pick.

Ultimately, visualizing the execution trace with and without frequent pattern mining each has their own pros and cons depending on the system under observation. The frequent pattern mining can help if a phase contains many sub-phases which contains patterns, e.g., loops. However, the cost of this is losing the tree structure, making navigating through the different levels of granularity less intuitive.

6.6 Limitations of Model

The frequent pattern mining performance highly depends on the model that is created in the training stage. For the training stage, a set of execution traces are needed to find the frequent pattern, e.g., test suite of the program. However, if a certain behavior is not covered by the execution traces of the training stage, then the model for SAGE will be unable to abstract the behavior further then the results obtained after phase detection and duplicate detection.

6.7 Threats to Validity

Threats to External Validity

Bad method naming, i.e., arbitrary names, cryptic abbreviations, or some generic names, is one of the main threats to external validation. Because we label the phases solely based on the method signatures, it does not come as a surprise that the labeling results highly depend on the quality of the method naming. This fact makes our results are not generalizable to any software system, because only the combination of a good code naming convention
and good labeling strategies can yield good results. However, given the fact that good naming convention has become one of fundamental quality metrics for modern software system, especially for the large-scale program, good naming convention is one necessity for project’s success, this threats to the validity of our labeling technique is weak.

Furthermore, in this thesis, all of the studies are conducted based on Java programs. This fact seems another threat to the external validity. However, as we can see, the two primary steps of our technique framework did not employ any peculiar features of Java. In the preliminary phase tree recovering step, for the programs that are implemented by other than Java, we can employ other instrumenters to record the method-enter events and call depths. And the whole model building stage is based on the preliminary phase forest but not on any programming language characteristics. Given these facts, we can see this threat is not strong to the generalization of SAGE.

Finally, the studied projects are limited to a small set of open-source system, which do not per se represent accurately all systems. Especially, because the systems under study are java systems and heavily rely on object-oriented design principles we cannot conclude that the conclusions based on the results can be generalized.

**Threats to Internal Validity**

One internal threat is the activities used in the user study may not be representative. To address this threat, we designed the user study based on the widely used open source text editor and selected the typical features of this kind of program. Similarly, another threat to internal validity is that the participants involved in the experiment may not be representative of general developers. To address this threat, we employed the senior graduate students with at least one year industrial internship experience, and none of the participants have any knowledge of using or of maintaining the subject programs.
Chapter 7

Related Work

Various researchers have proposed different ways to help developers gain a better understanding about the program’s behavior by using dynamic analysis.

7.1 Execution Trace Visualization

Software developers often use visualizations to gain a better understanding of their programs. In the field of dynamic analysis multiple visualizations have been created to convey certain properties of the system.

Both Cornelissen et al. [10] and Palepu et al. [22] visualized an execution trace. In both visualizations recurring patterns or phases can be noticed. In the former paper the authors even see smaller sub-phases within phases. This informed our intuition of phases within the execution and that those phases contain sub-phases.

In comparison to the fine-grained visualization there are tools that focus on the high-level functionality of the program. Alimadadi et al. [2] looked at the runtime behavior of event-driven web applications and mapped the low-level event-based behavior to a higher-level behavioral model. JIVE [26] is a visualization tool that determines the high-level phases in each thread and presents them to the user.

David Lo and Shahar Maoz also mine an execution trace in a hierarchical fashion to find different levels of granularity and visualize those in live sequence charts to the user [18]. However, the hierarchical structure is not based on detecting phases and its sub-phases, but it is derived from the package structure and live sequence charts.

Naser Ezzati-Jivan and Michel R.Dagenais visualized execution trace in a hierarchical way also using a timeline [14]. By zooming in the user can first physically zoom in on the current level, when zooming in further it will start presenting the lower level to you. In this work there is no tree structure, meaning there is no direct link between the layers. As a result, the user can look at parts of one layer and parts of other layers.
7. Related Work

7.2 Execution Mining

Execution trace mining techniques have been widely adopted to assist various software engineering tasks, including bug detection [13, 19], functionality identification [27] and specification identification [17, 4].

To improve the precision of mining-based bug detection techniques, Eichinger et al. [13] create weighted call graphs from execution traces and proposed a combination of structural and numerical techniques to mine them. Lo et al. [19] applied mining techniques on the program execution traces to discover a set of discriminative features, based on which they perform feature selection to build up a classifier to capture failures and anomalies. Safyallah and Sartipi [27] extracted execution patterns from feature-specific task scenarios. Those patterns are later used to detect execution patterns in the execution trace. Lee et al. [17] presented a tool, namely JMINER, to discover the parametric specification from execution traces. Ammons et al. [4] uncovered frequent interaction patterns from the execution trace as state machines to capture both temporal and data dependences, based on which, they proposed a method to discover formal specifications of code interaction. Each such technique mines execution trace information to help developers with a software-engineering task. In contrast, we support general developer comprehension of execution trace comprehension.

7.3 Trace Reduction and Abstraction Techniques

Due to the massive size of execution traces, scaleability issues inevitably arise. Multiple techniques have been proposed for reducing the execution trace size. Chan et al. [8] used sampling to reduce the size of the traces. In our approach, we utilize an entire trace to abstract it into higher levels of functionality.

Watanabe et al. [32] and Zaidman et al. [34] both propose abstract execution traces. The former proposed to find phases within executions based on the creation and destruction of objects in object-oriented programs. The latter used a heuristic approach to divide the trace into recurring event clusters. Additionally, Reiss and Renieris [25] encoded repeating events to reduce the size of the execution traces. However, each approach provides only one level of granularity for the developers. In this thesis, we add abstraction to the execution trace, but at multiple levels of granularity to assist developer inspection and exploration of execution behaviors.

Yang et al. [33] build similar to us a hierarchical tree that represents the control flow in the execution. In our approach on the other hand, we abstract over the control flow and focus on the actual functional behavior.

A different approach is to use metric based filters to reduce the size of an execution trace. For example, Hamou-Lhadj et al. [15] filtered out the utility components based on fan-in/fan-out to only keep the high-level components, resulting in a smaller execution trace. Also Cornelissen et al. [9] showed that limiting the stack depth can be an effective way to reduce the size of the execution trace. In these approaches parts of the execution are filtered out. In our approach we keep all events and use them to inform our abstractions.
Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we presented SAGE, a technique that creates a hierarchical abstraction and multiple levels of granularity of an execution trace for developer inspection, exploration, and comprehension. The technique is composed of a training stage and an application stage. The training stage can be informed by any number of executions (typically automated test cases). The application stage can be initiated by a developer and their execution trace can be abstracted and labeled within seconds.

We also introduced an interactive visualization, SAGEVIS, that enables developers to explore the execution trace at different levels of granularity. By utilizing the clustering and duplicate detection, developers identify recurring behaviors throughout the execution.

We provide an evaluation that consists of a quantitative evaluation of the computational costs and size reduction of execution traces, a user study that assessed the understanding of execution behaviors, a case-study demonstration of how SAGE and SAGEVIS can help developers identify the high-level behaviors in an execution trace, and finally a comparison of the visualizations with and without frequent pattern mining.

The quantitative evaluation revealed two things: (1) the technique is successful at substantially abstracting the execution trace and reducing the amount of data to be inspected by the developer to understand execution behavior; and (2) the training costs are not inconsequential, but these costs can be incurred infrequently and offline.

The user study showed that for most functionalities, the users were able to achieve a 70% or greater accuracy; for a few functionalities, there is room for improvement in future work.

The case study on the JAVAC compiler demonstrates SAGE’s ability to reveal the primary behavior phases within a large execution trace and its ability to provide behavior information at multiple levels of granularity.

The comparison of the visualization with and without frequent pattern mining demonstrated that both visualizations have their pros and cons. Frequent pattern mining can help reduce the number of phases presented to the developer. However, this is at the cost of easily navigating the hierarchical structure.
8. CONCLUSIONS AND FUTURE WORK

Finally, SAGE and SAGEVis enable developers to abstract and explore execution traces at different levels of granularity, which can help them gain a better understanding of the program.

8.2 Future Work

8.2.1 SAGE

In the future, we want to further improve the technique to address some of the limitations that it now faces. One of the main limitations of SAGE is that it focuses solely on one thread of the application. As a result, the developer is limited in gaining an understanding of behaviors that run concurrently to each other. Addressing this problem is twofold. First, research needs to be done to determine what information is necessary to help developers comprehend the dynamic concurrent behaviors. Second, we need to rethink how to present the obtained information to developers that is also scalable. Currently, the visualization scales well in terms of trace size, but it becomes crowded when many threads are running concurrently.

Finally, we want to evaluate SAGE on more and larger systems.

8.2.2 SAGEVis

Multithread Visualization

SAGEVis is currently limited to single-threaded applications. A future direction would be to extend the capabilities of SAGEVis to support multithreaded application. At the moment, multithreaded support can cause scalability issues due to the amount of threads being presented simultaneously. For example, in Figure 8.1a we present a mock of how the current version of multithreaded support would work. Developers are directly presented with 5 threads, which may or may not be of any use to the developer. However, it does directly force a lot more information on the user. Also for a few threads, it is still doable to visualize each thread as its own bar. However, when the number of threads simultaneously running increases and each thread is visualized on its own bar the visualization will become large and difficult to comprehend.

Our vision is to take the approach a step further. Instead of only clustering events with the same thread together, would it be possible to cluster complete threads together that perform similar behavior. For example, if multiple threads are running at the same time and perform a similar behavior we could collapse them into each other. In Figure 8.1b we present a mock of how the collapsed threads would make the overview a lot simpler. The developer can directly see via the label what happened within the block, but the developer can choose themselves to explore the inside further or not.

Dynamically Inferred State Machine

In the future, we want to explore other forms of visualizing the hierarchical phase structure. One possibility would be to visualize the hierarchical tree structure as a state machines
8.2. Future Work

based on the dynamic information, resulting in a \textit{dynamically inferred state machine}. The transitions between states would be based on which phases can follow each preceding phase. A visualization like this would be able to present a single threaded execution trace in a more dense visualization in comparison to the interactive timeline visualization of \textsc{SAGEVIS}.

In Figure 8.2, we present an mock visualization of how we envision the dynamically inferred state machine to work. Due to repeating phases at each level, we could present a state machine based on the phases and which phases are capable of following each other based on the dynamically obtained information. Also because of the different levels of granularity, we can show the dynamically inferred state machine at different levels of granularity.

Finally, we want to evaluate the visualization on multiple large software systems with a user group to determine where the visualization excels and where it can be improved.
Bibliography


Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

**DCT:** Dynamic Call Tree

**Execution event:** An event that happened during the execution of the program, *e.g.*, a method enter.

**Execution trace:** A file that contains all the events of a single execution of a program.

**FPM:** Frequent pattern mining

**Phase:** A group of execution events that constitute a functionality behavior within the execution.

**Preliminary phase tree:** A tree structure of phases without duplicate detection and clustering.

**SAGE:** Our approach for hierarchically abstracting an execution trace.

**SAGE VIS:** Our tool to visualize the abstracted execution trace.
Appendix B

Ranking of Labels for JEdit Behaviors
### B. Ranking of Labels for JEdit Behaviors

#### Table B.1: Behavior and the corresponding labels in order of rank

<table>
<thead>
<tr>
<th>Rank</th>
<th>New File</th>
<th>Change Font</th>
<th>Word Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>newFile</td>
<td>handlePropertiesChanged</td>
<td>eval</td>
</tr>
<tr>
<td>2</td>
<td>eval</td>
<td>removeLayoutComponent</td>
<td>getSelectedText</td>
</tr>
<tr>
<td>3</td>
<td>setBuffer</td>
<td>FontSelectorDialog</td>
<td>getNameSpace</td>
</tr>
<tr>
<td>4</td>
<td>reset</td>
<td>addOptionGroup</td>
<td>toObject</td>
</tr>
<tr>
<td>5</td>
<td>getToolTipText</td>
<td>FontSelector</td>
<td>get</td>
</tr>
<tr>
<td>6</td>
<td>setMode</td>
<td>invalidateScreenLineCounts</td>
<td>invokeAction</td>
</tr>
<tr>
<td>7</td>
<td>saveBackup</td>
<td>getRoot</td>
<td>invoke</td>
</tr>
<tr>
<td>8</td>
<td>closeBuffer</td>
<td>getActionSets</td>
<td>getAction</td>
</tr>
<tr>
<td>9</td>
<td>getFoldHandler</td>
<td>selectPane</td>
<td>visit</td>
</tr>
<tr>
<td>10</td>
<td>setSelection</td>
<td>valueChanged</td>
<td>reset</td>
</tr>
<tr>
<td>11</td>
<td>visit</td>
<td>setAntiAliasEnabled</td>
<td>getStructureHighlightColor</td>
</tr>
<tr>
<td>12</td>
<td>commitTemporary</td>
<td>GlobalOptionGroup</td>
<td>getOffsets</td>
</tr>
<tr>
<td>13</td>
<td>bufferRemoved</td>
<td>initKeyBindings</td>
<td>setVariable</td>
</tr>
<tr>
<td>14</td>
<td>removeBuffer</td>
<td>save</td>
<td>isPublic</td>
</tr>
<tr>
<td>15</td>
<td>IntegerArray</td>
<td>addKeyBinding</td>
<td>getModel</td>
</tr>
<tr>
<td>16</td>
<td>addBuffer</td>
<td>addTextListener</td>
<td>paintValidLine</td>
</tr>
<tr>
<td>17</td>
<td>toObject</td>
<td>addOptionPane</td>
<td>message</td>
</tr>
<tr>
<td>18</td>
<td>getNameSpace</td>
<td>removeNotify</td>
<td>showWordCountDialog</td>
</tr>
<tr>
<td>19</td>
<td>compare</td>
<td>eval</td>
<td>doWordCount</td>
</tr>
<tr>
<td>20</td>
<td>get</td>
<td>addNotify</td>
<td>getThis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Close &amp; Exit</th>
<th>Keyword Highlight</th>
<th>Open File</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>removeNotify</td>
<td>createRegExpEOLSpanRule</td>
<td>getSelectedFiles</td>
</tr>
<tr>
<td>2</td>
<td>close</td>
<td>createEOLSpanRule</td>
<td>getExtendedAttribute</td>
</tr>
<tr>
<td>3</td>
<td>save</td>
<td>setMode</td>
<td>addAction</td>
</tr>
<tr>
<td>4</td>
<td>stop</td>
<td>add</td>
<td>isMiddleButton</td>
</tr>
<tr>
<td>5</td>
<td>closeView</td>
<td>getTokenMarker</td>
<td>filesSelected</td>
</tr>
<tr>
<td>6</td>
<td>dispose</td>
<td>KeywordMap</td>
<td>mouseClicked</td>
</tr>
<tr>
<td>7</td>
<td>setBuffer</td>
<td>doAWTRequest</td>
<td>filesActivated</td>
</tr>
<tr>
<td>8</td>
<td>saveBackup</td>
<td>loadMode</td>
<td>eval</td>
</tr>
<tr>
<td>9</td>
<td>setFirstPhysicalLine</td>
<td>addRuleSet</td>
<td>waiting</td>
</tr>
<tr>
<td>10</td>
<td>setSelection</td>
<td>compare</td>
<td>visitEnd</td>
</tr>
<tr>
<td>11</td>
<td>removeFromBus</td>
<td>updatePosition</td>
<td>compare</td>
</tr>
<tr>
<td>12</td>
<td>reset</td>
<td>handleMessage</td>
<td>FileCellRenderer</td>
</tr>
<tr>
<td>13</td>
<td>saveGeometry</td>
<td>visit</td>
<td>getTableCellRendererComponent</td>
</tr>
<tr>
<td>14</td>
<td>isUndoInProgress</td>
<td>ActionSet</td>
<td>transactionComplete</td>
</tr>
<tr>
<td>15</td>
<td>unsetProperty</td>
<td>finishSaving</td>
<td>setDirectory</td>
</tr>
<tr>
<td>16</td>
<td>charsToEscapes</td>
<td>resetClearDirty</td>
<td>getIconForFile</td>
</tr>
<tr>
<td>17</td>
<td>writeXMLDeclaration</td>
<td>setTokenMarker</td>
<td>createBeanShellAction</td>
</tr>
<tr>
<td>18</td>
<td>getToolTipText</td>
<td>parseBufferLocalProperties</td>
<td>directoryLoaded</td>
</tr>
<tr>
<td>19</td>
<td>removePluginJAR</td>
<td>sort</td>
<td>VFSDirectoryEntryTable</td>
</tr>
<tr>
<td>20</td>
<td>runInDispatchThreadAndWait</td>
<td>bufferSetSorted</td>
<td>createMenuInsert</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>Typing</th>
<th>Search</th>
<th>Indent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>getWidget</td>
<td>SearchDialog</td>
<td>preContentRemoved</td>
</tr>
<tr>
<td>2</td>
<td>getComponent</td>
<td>eval</td>
<td>shiftIndentRight</td>
</tr>
<tr>
<td>3</td>
<td>userInput</td>
<td>HistoryTextField</td>
<td>getSelectedLines</td>
</tr>
<tr>
<td>4</td>
<td>contentInserted</td>
<td>setInstantPopups</td>
<td>createWhiteSpace</td>
</tr>
<tr>
<td>5</td>
<td>preContentInserted</td>
<td>createReplaceLabelAndField</td>
<td>preContentInserted</td>
</tr>
<tr>
<td>6</td>
<td>insert</td>
<td>HistoryTextArea</td>
<td>contentInserted</td>
</tr>
<tr>
<td>7</td>
<td>eval</td>
<td>updateEnabled</td>
<td>contentRemoved</td>
</tr>
<tr>
<td>8</td>
<td>doPaintContents</td>
<td>createFieldPanel</td>
<td>transactionComplete</td>
</tr>
<tr>
<td>9</td>
<td>getBuild</td>
<td>initFocusOrder</td>
<td>insert</td>
</tr>
<tr>
<td>10</td>
<td>getVersion</td>
<td>createSearchSettingsPanel</td>
<td>beginCompoundEdit</td>
</tr>
<tr>
<td>11</td>
<td>buildToVersion</td>
<td>createButtonsPanel</td>
<td>endCompoundEdit</td>
</tr>
<tr>
<td>12</td>
<td>paintString</td>
<td>showSearchDialog</td>
<td>eval</td>
</tr>
<tr>
<td>13</td>
<td>propertiesChanged</td>
<td>createFindLabelAndField</td>
<td>getIndentSize</td>
</tr>
<tr>
<td>14</td>
<td>visit</td>
<td>createMultiFilePanel</td>
<td>setLineEndOffset</td>
</tr>
<tr>
<td>15</td>
<td>newFile</td>
<td>loadGeometry</td>
<td>getCompressedReplaceFromReplaceReplace</td>
</tr>
<tr>
<td>16</td>
<td>isElectricKey</td>
<td>componentRemoved</td>
<td>getLeadingWhiteSpace</td>
</tr>
<tr>
<td>17</td>
<td>isPrefixActive</td>
<td>componentAdded</td>
<td>getLeadingWhiteSpaceWith</td>
</tr>
<tr>
<td>18</td>
<td>addNotify</td>
<td>VariableGridLayout</td>
<td>getReplaceFromRemoveInsert</td>
</tr>
<tr>
<td>19</td>
<td>reset</td>
<td>getSearchDialog</td>
<td>setDirty</td>
</tr>
<tr>
<td>20</td>
<td>setBuffer</td>
<td>getSelectedText</td>
<td>get</td>
</tr>
</tbody>
</table>
Appendix C

Questionnaire
1 Information

Could you please provide the following information:

- Name,
- Occupation,
- E-mail

2 Introduction

The goal of this research project is to automatically detect the phases in the execution traces of a program. These phases will be clustered and labeled. If similar phases are executed multiple times in a row that will be grouped together and presented as one phase.

The questions that will be asked in this survey are focused on seeing if the detected phases and corresponding labels are effective to describe what happened in the program.

3 Multiple Choice Questions

The following questions are all focused around the program jEdit. This program is a text editor for programmers, see Figure 1. It contains the general features that most text editors have, but also extra features that are useful for software development:

- Typing, indent;
- New file, Save, Save As, Close;
- Copy, Paste, Cut;
- Undo, Redo;
- Change Case;
- Delete Lines;
- Change Style of Text Area, Change Font Style;
- Find & Replace, Word Count;
- keyword highlighting;

In the following questions phases will be presented to you. Every phase can be described by one of the previous mentioned features OR is an Undefined or Unknown feature. Please identify the feature that matches the phase as close as possible. Per question there is only one answer allowed.
### 3.1 Question 1

Listing 1: List of phases belonging to Question 1

**Key Method Signatures:**
- `preContentRemoved`
- `undo`
- `userInput`
- `getTransferable`
- `findWordStart`
- `resizeSelection`
- `RichTextTransferable`
- `getTransferDataFlavors`
- `setSelectedText`
- `preContentInserted`
- `insert`
- `contentInserted`
- `contentRemoved`
- `transactionComplete`
- `xyToOffset`
- `setRegister`
- `isElectricKey`
- `findWordEnd`
- `xToOffset`
- `isPrefixActive`
- `replaceSelection`
- `beginCompoundEdit`
- `endCompoundEdit`
- `getSelectionAtOffset`
- `eval`
- `toLowerCase`
- `fireBeginUndo`
- `mouseDragged`
- `exchangeLastElement`

(A). Change Font  
(B). Close&Exit  
(C). User Typing  
(D). Open File  
(E). Undefined or Unknown

### 3.2 Question 2

Listing 2: List of phases belonging to Question 2

**Key Method Signatures:**
- `eval`
- `getSelectedText`
- `getNameSpace`
- `ToObject`
- `get`
- `invokeAction`
- `invoke`
- `getAction`
- `visit`
- `reset`
- `getStructureHighlightColor`
- `getOffsets`
- `setVariable`
- `isPublic`
- `getModel`
- `paintValidLine`
- `message`
- `showWordCountDialog`
- `doWordCount`
- `getThis`
- `NameSpace`
- `cacheBlock`
- `castObject`
- `Primitive`
- `runCachedBlock`
- `getMethod`
- `BshMethod`
- `invokeMethod`
- `Variable`
- `handleMessage`

(A). Change Case  
(B). New File  
(C). Open File  
(D). WordCount  
(E). Undefined or Unknown

### 3.3 Question 3

Listing 3: List of phases belonging to Question 3

**Key Method Signatures:**
- `removeNotify`
- `close`
- `save`
- `stop`
- `closeView`
- `dispose`
- `setBuffer`
- `saveBackup`
- `setFirstPhysicalLine`
- `setSelection`
- `removeFromBus`
- `reset`
- `saveGeometry`
- `visit`
- `unsetProperty`
- `charsToEscapes`
- `writeXMLDeclaration`
- `getToolTipText`
- `removePluginJAR`
- `runInDispatchThreadAndWait`
- `EditorExitRequested`
- `getPluginJARs`
- `removeBuffer`
- `textAreaDisposed`
- `exit`
- `EditorExiting`
- `windowClosing`
- `errorOccurred`
- `waitForIoTasks`
- `closeStream`

(A). Close&Exit  
(B). New File  
(C). Change Font  
(D). WordCount  
(E). Undefined or Unknown
### 3.4 Question 4

**Listing 4**: List of phases belonging to Question 4

| Key Method Signatures: | preContentRemoved, shiftIndentRight, getSelectedLines, createWhiteSpace, preContentInserted, contentInserted, contentRemoved, transactionComplete, insert, beginCompoundEdit, endCompoundEdit, eval, getIndentSize, setLineEndOffset, getCompressedReplaceFromReplaceReplace, getLeadingWhiteSpace, getLeadingWhiteSpaceWidth, getReplaceFromRemoveInsert, setDirty, add, getNameSpace, toObject, reset, remove, fireTransactionComplete, doDelayedUpdate, handleTokenWithSpaces, getReadyToBreakFold, expandFold |

1. WordCount
2. New File
3. Search
4. Indent
5. Undefined or Unknown

### 3.5 Question 5

**Listing 5**: List of phases belonging to Question 5

| Key Method Signatures: | preContentRemoved, undo, userInput, getTransferable, findWordStart, resizeSelection, RichTextTransferable, getTransferDataFlavors, setSelectedText, preContentInserted, insert, contentInserted, contentRemoved, transactionComplete, xyToOffset, setRegister, isElectricKey, findWordEnd, xToOffset, isPrefixActive, replaceSelection, beginCompoundEdit, endCompoundEdit, getSelectionAtOffset, eval, toLowerCase, fireBeginUndo, mouseDragged, exchangeLastElement, cu |

1. Indent
2. Close&Exit
3. User Typing
4. Keyword Highlight
5. Undefined or Unknown

### 3.6 Question 6

**Listing 6**: List of phases belonging to Question 6

| Key Method Signatures: | getWidget, getComponent, userInput, contentInserted, preContentInserted, insert, eval, doPaintContents, getBuild, getVersion, buildToVersion, paintString, propertiesChanged, transactionComplete, newFile, isElectricKey, isPrefixActive, addNotify, reset, setBuffer, getLineContext, markTokens, recalculateVisibleLines, setMode, addBuffer, init, get, saveBackup, getResourceAsStream, detectEncoding |

1. Keyword Highlight
2. New File
3. Search
4. WordCount
5. Undefined or Unknown
3.7 Question 7

Listing 7: List of phases belonging to Question 7

**Key Method Signatures:** createRegexpEOLSpanRule, createEOLSpanRule, setMode, add, getTokenMarker, KeywordMap, doAWTRequest, loadMode, addRuleSet, compare, updatePosition, handleMessage, visit, isUndoInProgress, finishSaving, resetClearDirty, setTokenMarker, parseBufferLocalProperties, sort, bufferSetSorted, getFoldHandler, access, propertiesChanged, setPerformingIO, getBooleanProperty, run, maybeReload, handleVFSUpdate, getJEditHome, resolveSymlink

(A). New File  
(B). Search  
(C). Keyword Highlight  
(D). Undo  
(E). Undefined or Unknown

3.8 Question 8

Listing 8: List of phases belonging to Question 8

**Key Method Signatures:** preContentRemoved, undo, userInput, getTransferable, findWordStart, resizeSelection, RichTextTransferable, getTransferDataFlavors, setSelectedText, preContentInserted, insert, contentInserted, contentRemoved, transactionComplete, xyToOffset, setRegister, isElectricKey, findWordEnd, xToOffset, isPrefixActive, replaceSelection, beginCompoundEdit, endCompoundEdit, getSelectionAtOffset, eval, toLowerCase, fireBeginUndo, mouseDragged, exchangeLastElement, cu

(A). Open File  
(B). New File  
(C). User Typing  
(D). Close&Exit  
(E). Undefined or Unknown

3.9 Question 9

Listing 9: List of phases belonging to Question 9

**Key Method Signatures:** SearchDialog, eval, HistoryTextField, setInstantPopups, createReplaceLabelAndField, HistoryTextArea, updateEnabled, createFieldPanel, initFocusOrder, createSearchSettingsPanel, createButtonsPanel, showSearchDialog, createFindLabelAndField, createMultiFilePanel, loadGeometry, componentRemoved, componentAdded, VariableGridLayout, getSearchDialog, getSelectedText, getToolTipText, getNameSpace, toObject, getModel, get, setVisible, invokeAction, invoke, getAction, getResourceAsStream

(A). User Typing  
(B). Search  
(C). WordCount  
(D). New File  
(E). Undefined or Unknown
3.10 Question 10

Listing 10: List of phases belonging to Question 10

**Key Method Signatures:** SearchDialog, eval, add, visit, setInstantPopups, HistoryTextField, createReplaceLabelAndField, HistoryTextArea, updateEnabled, createFieldPanel, initFocusOrder, createMultiFilePanel, createRegExpEOLSpanRule, createEOLSpanRule, createSearchSettingsPanel, createButtonsPanel, showSearchDialog, createFindLabelAndField, handleMessage, setMode, getTokenMarker, getBooleanProperty, loadGeometry, KeywordMap, loadMode, doAWTRequest, addRuleSet, paintHighlight, componentRemoved, acces

(A). Undo
(B). Change Case
(C). User Typing
(D). Search
(E). Undefined or Unknown

3.11 Question 11

Listing 11: List of phases belonging to Question 11

**Key Method Signatures:** eval, paintValidLine, getSelectedText, getNameSpace, toObject, get, invokeAction, invoke, getAction, visit, reset, getStructureHighlightColor, getOffsets, setVariable, isPublic, getModel, paintScreenLineRange, message, showWordCountDialog, doWordCount, getThis, NameSpace, cacheBlock, castObject, Primitive, runCachedBlock, getMethod, BshMethod, invokeMethod, Variabl

(A). Keyword Highlight
(B). Indent
(C). WordCount
(D). Change Font
(E). Undefined or Unknown

3.12 Question 12

Listing 12: List of phases belonging to Question 12

**Key Method Signatures:** handlePropertiesChanged, removeLayoutComponent, FontSelectorDialog, addOptionGroup, FontSelector, invalidateScreenLineCounts, getRoot, getActionSets, selectPane, valueChanged, setAntiAliasEnabled, GlobalOptionGroup, initKeyBindings, save, addKeyBinding, addTextListener, addOptionPane, removeNotify, eval, addNotify, save, init, addComponent, propertiesChanged, ok, removeToolBar, parseStyle, addToolBar, combinedOptions, addTreeModelListener

(A). Close&Exit
(B). Search
(C). Change Case
(D). Change Font
(E). Undefined or Unknown
### 3.13 Question 13

Listing 13: List of phases belonging to Question 13

**Key Method Signatures:**
- `newFile`
- `eval`
- `setBuffer`
- `reset`
- `getToolTipText`
- `setMode`
- `saveBackup`
- `closeBuffer`
- `getFoldHandler`
- `setSelection`
- `visit`
- `commitTemporary`
- `bufferRemoved`
- `removeBuffer`
- `IntegerArray`
- `addBuffer`
- `toObject`
- `getNameSpace`
- `compare`
- `contentRemoved`
- `createUntitledBuffer`
- `getNextUntitledBufferId`
- `getOwner`
- `invokeAction`
- `getAction`
- `getResourceAsStream`
- `invoke`
- `access`
- `getName`

(A). Open File  
(B). WordCount  
(C). Keyword Highlight  
(D). New File  
(E). Undefined or Unknown

### 3.14 Question 14

Listing 14: List of phases belonging to Question 14

**Key Method Signatures:**
- `removeNotify`
- `close`
- `dispose`
- `save`
- `stop`
- `closeView`
- `setBuffer`
- `saveBackup`
- `setFirstPhysicalLine`
- `setSelection`
- `removeFromBus`
- `reset`
- `saveGeometry`
- `visit`
- `unsetProperty`
- `charsToEscapes`
- `writeXMLDeclaration`
- `removePluginJAR`
- `runInDispatchThreadAndWait`
- `EditorExitRequested`
- `getPluginJARs`
- `removeBuffer`
- `textAreaDisposed`
- `exit`
- `EditorExiting`
- `windowClosing`
- `errorOccurred`
- `waitForIoTasks`
- `closeStream`
- `closeAllBuffer`

(A). WordCount  
(B). Close&Exit  
(C). User Typing  
(D). Search  
(E). Undefined or Unknown

### 3.15 Question 15

Listing 15: List of phases belonging to Question 15

**Key Method Signatures:**
- `getSelectedFiles`
- `getExtendedAttribute`
- `addAction`
- `isMiddleButton`
- `filesSelected`
- `mouseClicked`
- `filesActivated`
- `eval`
- `waiting`
- `visitEnd`
- `compare`
- `FileCellRenderer`
- `getCellRendererComponent`
- `ActionSet`
- `setDirectory`
- `setIconForFile`
- `createBeanShellAction`
- `directoryLoaded`
- `VFSDirectoryEntryTable`
- `createMenuItem`
- `setTemporaryProperty`
- `VFSBrowser`
- `VFSFileChooserDialog`
- `setBuffer`
- `visit`
- `openFile`
- `setInstantPopups`
- `HistoryTextField`
- `put`
- `saveBackup`

(A). Indent  
(B). User Typing  
(C). Change Case  
(D). Open File  
(E). Undefined or Unknown