Hierarchical Abstraction of Execution Traces for Program Comprehension

Yang Feng†, Kaj Dreef†, James A. Jones†, Arie van Deursen⋆
†University of California, Irvine, Irvine, USA
⋆Delft University of Technology, Delft, The Netherlands
{yang.feng,kdreef,jajones}@uci.edu
arie.vandeursen@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 in order 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, we propose a fully automatic approach to present a semantic abstraction with different levels of functional granularity from full execution traces. Our approach builds multi-level abstractions and identifies frequent behaviors at each level based on a number of execution traces, and then, it labels phases within individual execution traces according to the identified major functional behaviors of the system. To validate our approach, 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 approach is capable of presenting users a high-level comprehensible abstraction of execution behavior. Based on a real world subject program the participants in our user study were able to achieve a mean accuracy of 70%.

ACM Reference Format:
Yang Feng†, Kaj Dreef†, James A. Jones†, Arie van Deursen⋆. 2018. Hierarchical Abstraction of Execution Traces for Program Comprehension. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3196321.3196343

1 INTRODUCTION

With the increasing size and complexity of modern software, comprehending the source code has become one of the most expensive tasks in all phases of software life cycle because software must be sufficiently understood before it can be properly modified or enhanced [10, 18]. 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 development tasks, such as adding features, debugging, or optimization [10, 21, 40].

Unfortunately, the complexity of modern software makes it difficult for developers to build up this cognitive mapping manually, especially given the overwhelmingly large size of source code. 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.,* [34, 34, 39, 42]) is to summarize or aggregate executions into "phases", *i.e.,* groups of execution events that constitute functionality behaviors within the execution. Another approach (*e.g.,* [8, 17, 31, 34]) is to visualize executions, allowing developers to observe interesting internal behaviors of the software execution.

These approaches are promising but are limited in a number of ways that might be improved upon: (1) execution abstractions are limited to a single level of granularity and thus might not support well a variety of maintenance tasks that require comprehension at either a coarse or fine grain, or both (*e.g.,* [34, 39, 42]); (2) executions events are filtered to support understanding of overall behavior but may omit important events (*e.g.,* [9, 22, 39]); 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.,* [8, 31]).

In this work, we present Sage — a novel approach 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, and (3) labels the hierarchical execution representation comprehensibly.

The hierarchy provides representations of execution functionality across 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 approach provides two main advantages: (1) the output is a hierarchical structure that provides the developers a view across multiple comprehension levels. This view can guide developers in locating the functionalities of interests; (2) on each level, identified

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196343

functionalities are organized sequentially. In this view, all of identified functionalities are abstracted and presented to include multiple frequent patterns of behavior to help identify significant behaviors.

With our prototype implementation of Sage, we demonstrate its ability to create hierarchical representations of execution behaviors that allow for user investigation of both high-level execution behavior and lower-level constituent behaviors within high-level behaviors of interest. Further, we evaluate (1) its ability to reduce the overwhelming number of events within an execution trace to a comprehensible number for developers; (2) the efficiency of building these representations; and (3) the comprehensibility of the identified functionality phases.

We present an evaluation that includes: (1) a case study that demonstrates how the approach works for a large program (Javac); (2) a quantitative evaluation of the size reduction of the execution trace to a more comprehensible abstraction, with its computational costs; and (3) a user study that assesses users' ability to understand the approach's generated behavior abstractions. 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 paper are as follows:

- We present our approach 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 method to generate a hierarchical abstraction of execution behavior for an execution under investigation. This abstraction enables developers to trace and locate the functionalities of interests and corresponding context.
- With this hierarchical representation of an execution under investigation, we present our approach for abstracting, modeling, and labeling it for developer inspection and exploration.

2 MOTIVATION AND CHALLENGES

Software must be sufficiently understood before it can be properly modified. Especially, for a given development task, developers or maintainers must map the high-level software functionality with the source code to gain a sufficient level of understanding. However, modern software is large in its codebase and its runtime behavior is complex, which makes bridging the cognitive gap between the high-level software functionality with low-level implementation difficult for developers. The expensive cost of manually building this cognitive analysis naturally motivates the needs of automated methods for assisting developers in understanding the programs. Under this circumstance, developers collect execution traces (*i.e.,* a log of internal execution events) to observe and reason about low-level software execution 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.

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 [34] 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.* [8] observed execution phases with their Extravis visualization. They observed that for a program that 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:

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 create 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.

Challenge 2: Behaviors contain sub-behaviors. As observed by Cornelissen *et al.* [8], 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.

Challenge 3: Identifying functionality on demand. Because software developers are often interested in the software behavior with different granularities for completing various development tasks [10], identifying functionality phases on demand become challenging in practice. Multi-level representation can help users to access any arbitrary area of a trace with support for exploration, panning, searching focusing, and zooming. Hence, our approach seeks to provide a hierarchical structure and enable developers to analyze and explore execution traces.

3 APPROACH

In this section, we introduce our approach for hierarchically abstracting an execution trace and addressing the challenges mentioned in Section 2. In Figure 1, we present the framework of our approach 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 [30] as the instrumenter to collect execution traces.

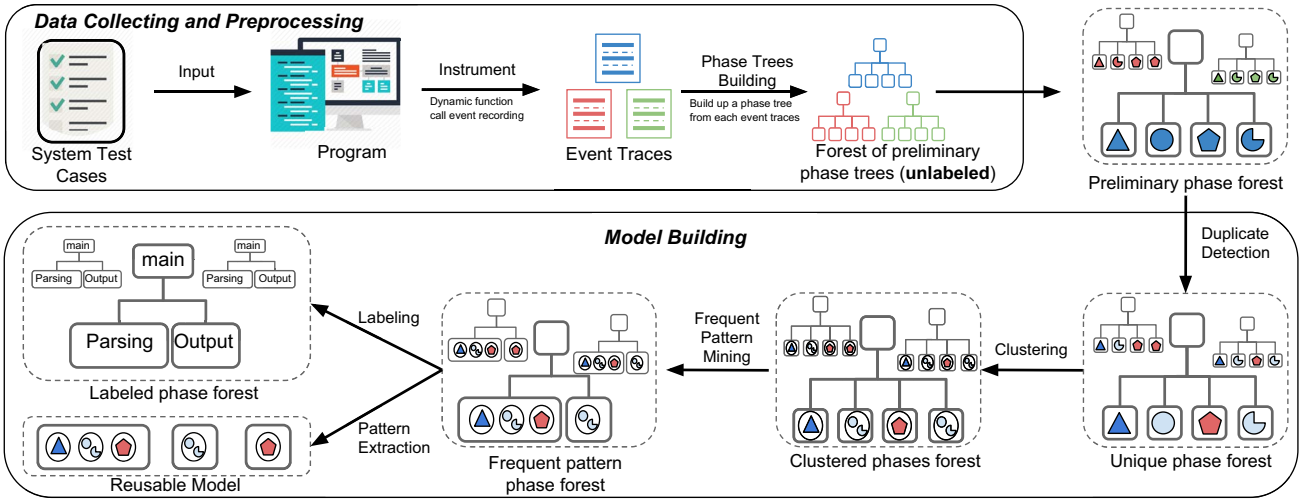


Figure 1: The framework for abstracting and labeling execution events, *i.e.,* the model building stage for the Sage approach

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

3.1 Data Collecting

In our approach, execution traces are obtained using the dynamic instrumenter, Blinky [30]. 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 [33] and Salah *et al.* [36], 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) method ID, (2) method signature, and (3) call depth. For multi-threaded programs, each thread is analyzed as a separate trace.

3.2 Preliminary Phase Tree Building

The next step of the approach is to reproduce a dynamic call tree from each method trace. Ammons *et al.* [2] noted that dynamic call trees provide a hierarchical abstraction that is the most precise but space-inefficient data structure to present the calling context of the execution. As such, our approach uses this data structure as the basis for further abstraction.

Dynamic call trees were chosen as a basis on which we will abstract the execution and build our hierarchical execution representations because it naturally presents a hierarchical task-based representation of the execution behaviors that were performed. The methods themselves abstract the behavior of the entirety of their called methods. For example, a method `readFile()` may repeatedly call `readLine()`, which may in turn repeatedly call `readByte()`. In this example, the call to `readFile()` provides an abstraction of the functionality and behavior the sequence of `readLine()` and `readByte()` method calls.

Dynamic call trees are composed of ordered nodes, structured by an edge from each caller to callee method [2]. Given the sequence of method-invocation events in a trace, along with each method invocation's call depth, we reverse engineer the calling structure from the flat sequential trace. The process for recreating the dynamic call tree from the flat trace is relatively straightforward—we derive the caller-callee structure by assessing the increases and decreases in call depth in the method trace. Although relatively straightforward, we account for situations such as exception handling and the invocation of non-instrumented methods (which may, for example, call back to instrumented methods). In both cases, the resulting traces may result in call depths that jump in non-incremental ways.

In Figure 2, we present a simple example to demonstrate a classic dynamic call tree for a partial execution trace of NanoXML. The vertical axis represents the call depth. The order of method-enter events are represented on the horizontal axis. Each node represents one method-invocation event, and the directed edges represent a call from a caller method to a called method. We will use this example to demonstrate some of the following steps of our approach.

Using the dynamic call tree, we then create a set of "preliminary phases" to build a "preliminary phase tree" for each execution. A preliminary phase is a subtree of dynamic method calls of increasing call depth, and preliminary phases may contain other preliminary phases. We identify preliminary phases by analyzing the shape of the dynamic call tree — at locations where there is a local minima in the call stack depth (*i.e.,* the call stack decreases then immediately increases), we identify a phase boundary. Figure 3 depicts our example dynamic call tree from Figure 2, labeled with the identified phases. Notice that phase P1 includes method calls from `scanData` to `processSchema`, with no sub-phases, because the calling depths are strictly increasing—this represents that a single behavior is being performed. The set of all preliminary phase trees are referred to as the *preliminary phase forest.*

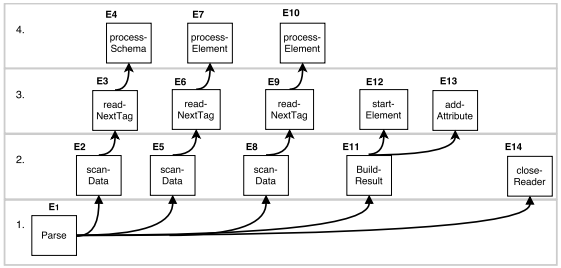


Figure 2: Dynamic call tree of an execution trace with call depth

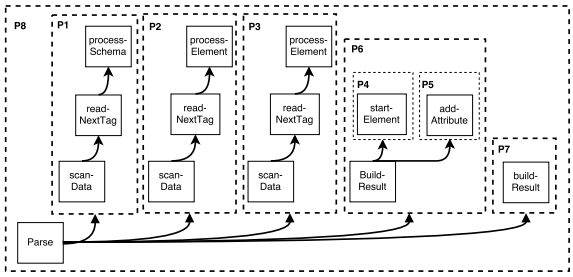


Figure 3: Phase tree of an execution trace

3.3 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 steps:

(1) **Duplicate phase detection**: Based on the *preliminary phase forest*, we identify duplicate *preliminary phases* on each level of the hierarchical structure. In our approach, we create a global key for the nodes of the *preliminary phase trees* based on the level 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 3.3.1.
(2) **Phase clustering**: Some *unique phases* perform similar functionality by invoking a similar set of methods. Our approach 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 3.3.2.
(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 level of the hierarchical structure of clustered phases. In our approach, 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 3.3.3.
(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 3.3.4.

3.3.1 Duplicate Phase Detection. Because our preliminary tree building approach processes the execution traces one by one, some identical *preliminary phases* may occur multiple times throughout forest. As shown in Figure 3, two phases, P_2 and P_3 are identical. In the figure, we simply show method names, but the approach actually uses fully qualified method signatures. The identification of duplicate phases is important in two ways: (1) identifying identical behavior can help developer understanding within and across executions, and (2) reducing the set of unique behaviors can make subsequent steps of the approach more efficient.

To identify duplicate preliminary phases, we create a *global key* as the identity for each phase of the forest. In our implementation, the global key is comprised of: (1) the hierarchical level, *i.e.,* the distance from the root node to the phase node, and (2) the invoked method list, ordered by their first invocation in the phase. For example, a phase that is rooted at hierarchical level 2 that contains the list of methods $\langle m_a, m_b, m_c, m_b, m_c \rangle$ 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. That means, we can replace the phases that have the same global key by one *unique phase*.

3.3.2 Phase Clustering. The prior step identified the strictly identical *preliminary phases* across the whole forest. However, the executions may still contain similar behaviors that have only slight variations on the method invocation profiles. As shown in Figure 3, P_2 and P_3 read the next tags and processed the XML elements in the exact same behavior, whereas P_1 encountered the header element of XML file that should be processed by the method `processSchema`. As such, the main functionalities of $P1$, $P2$, and $P3$—the reading the next tags and processing the elements—may be considered similar, although not identical.

To identify the similar phases revealing the similar functionalities, we applied *agglomerative hierarchical clustering* (AHC) [24] on the *unique phases* set. In this step, there are three fundamental parameters: (1) the threshold to stop the agglomerating; (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.

To measure the distance between phases, we use the *Jaccard Distance*, which is widely used for comparing the similarity and diversity of data sets, to measure the similarity between invoked methods among the *unique phases*. We defined *Jaccard Distance* in Equation 1, in which, 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. The clustering process iteratively clusters the two most similar phases, as measured by the Jaccard distance, until a maximum distance threshold *maxDis* is reached.

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

For example, in Figure 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 methods are invoked in P_1 or 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 a result of the clustering, we mark the unique phases with corresponding cluster identifiers. The clustered phases are referred to as *clustered phases*, and the resulting phase trees with the clustered phases are referred to as *clustered phase trees*. Moreover, as a final stage of producing the clustered phase trees, we aggregate sequentially contiguous phases that share the same cluster identifiers into one larger clustered phase to further abstract a series of similar behavior into a single larger and longer-running behavior.

3.3.3 Frequent Pattern Mining. Because developers often invoke methods in patterns to implement functionalities (as described in [1, 6, 38]), our approach 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) [4], 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 [4]. 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 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**. *minSup* assists in identifying the patterns with frequencies higher than the threshold. The second maximum gap between the items of the patterns, denoted as *maxGap*, specifies if gaps are allowed in frequent patterns and the allowed size of those gaps. For example, if *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 *maxGap* is set to N, a gap of $N - 1$ items is allowed between two consecutive items of a pattern.

$$support(Pattern_i) = \frac{|\{t \in T; Pattern_i \subseteq T\}|}{|T|} \quad (2)$$

We found $maxDis = 0.2$, $minSup = 0.3$, $maxGap = 1$ to provide a well balanced level of abstraction, and thus use these for our studies and experiments in the next section.

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.

3.3.4 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.,* [11, 14, 25, 27]) have found that source code contains valuable and meaningful clues for describing functionalities. Our approach 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 set of labels that could make it 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 *TF-IDF* (term frequency-inverse document frequency) [37] metric as the weight of our labels. Our approach presents the most highly weighted labels to describe the phases.

In the implementation, we treat the method signatures 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. After we calculate the weight for each of the labels, we sort them and extract the top 20 terms as the final label set for each frequent pattern phase.

$$\text{tf-idf}_{t,d} = (1 + \log(\text{tf}_{t,d})) \cdot \log(1 + \frac{N}{\text{df}_t}) \quad (3)$$

3.4 Execution Abstraction

Once the Sage approach completes the model building stage of all prior steps for any set of training executions, a model of all behaviors is available for abstracting any execution (whether seen before or new) so it can be provided to a developer for inspection

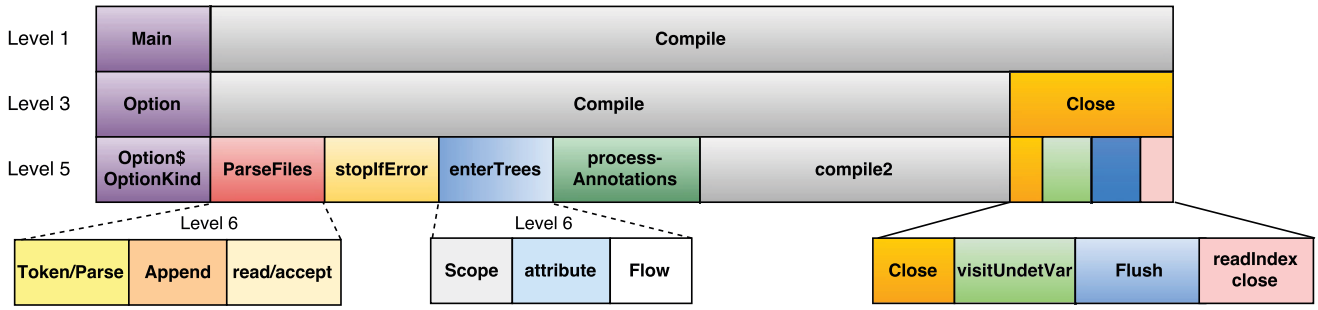


Figure 4: Hierarchical execution phase abstraction for an execution of Javac

and comprehension. For the execution under inspection, the phase detection step is first performed, and all subsequent steps are performed by utilizing the set of phases identified in the trained model of mined phases. Moreover, the semantic labels are also precomputed in the model building stage. It should be noted that the model building 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. Figure 4 provides a visual representation of our output. Section 4.2 provides a quantitative evaluation that includes model building- and application-stage timings.

4 EVALUATION

We implemented our approach in a tool, called Sage, to evaluate the effectiveness of our approach. With Sage, we conducted three studies. First, we present a case study to demonstrate the approach and the way in which it is capable of abstracting a large execution trace into a hierarchical structure. In this case study, we present the horizontal view as well as the vertical view contained in the hierarchical structure. Next, we present a quantitative evaluation on three real programs to assess the benefits and costs of our approach on abstracting functionality phases for those subjects and executions. Finally, we present a user study to evaluate the ability of our approach to generate comprehensible functionality phases from execution traces.

These evaluations are motivated by three research questions:

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

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

RQ3: To what extent are the labeled phases of our approach comprehensible for developers?

4.1 Case Study on Javac

In this case study we applied our approach on Javac, an open-source Java compiler, to demonstrate our approach. We chose Javac as a subject program for this case study because the functionality of a compiler is relatively familiar to software-engineering researchers with a relatively understood and well defined pipeline of phases.

To train the model, we used the 19 test cases provided by the Open JDK project[1] that target the "Warning," "Assert," and "Enum" features. The Blinky instrumenter [30] was used to collect the method call event traces for these test cases. The combined size of the 19 trace files was 995 megabytes. In total, the trace files contained more than 20 million trace events—far too large for developer comprehension.

Using this trained model, we now demonstrate the output for an execution that we wish to abstract. The test case `Enum2.java` is a file for Javac to compile. We produced the execution trace for Javac when compiling the `Enum2.java` source-code file. The execution trace for this execution contains 1,146,572 events.

Figure 4 depicts three complete levels of our phase hierarchy and one partial level for some sub-phases for Javac on the `Enum2.java` test-case execution. In this figure, we visualize the identified phases for Levels 1, 3, and 5 (and partial Level 6) as vertically stacked layers. We omitted level 2 and 4, because they were similar to Level 1 and 3 respectively, and due to limited space in the paper. Each rectangle in each level represents an execution phase, and the width of each phase indicates the length of that phase from the execution trace. In this figure, we included only the top method-name label due to space limitations.

Our hierarchical structure presents users both horizontal and vertical views of a large trace. The horizontal view enables users to analyze all events and their relationship on the same level, including going back and forth in the trace, inspecting events, and processing the data for understanding the trace based on one abstraction level. The vertical view enables users to analyze and locate software behaviors with different functionality granularities. Based on these two views, our approach alleviates the third challenge we discussed in the Section 2.

We observe that the Javac execution for compiling `Enum2.java` is unsurprisingly dominated by a "compile" phase in the top, most-abstracted level. However, when we look into the deeper levels of the hierarchy, we observe that the "compile" phase can be decomposed into sub-phases that account for constituent behaviors. Many of these phases have labels that are interpretable by developers without in depth knowledge of Javac, such as "parseFiles," whereas others would probably be more meaningful for the developers of Javac (for whom we intend the visualization to be useful), such as "visitUndetVar." An interactive visualization of the output can be found on our website [2]. Overall, we observe a drastic abstraction of the trace from over 1 million events in the execution trace to only 10 functionality phases at Level 5 of our hierarchical phase model.

[1] http://openjdk.java.net/

[2] https://spideruci.github.io/sagevis-demo/

Table 1: Size abstraction and time cost for each step of the model building procedure

Projects	|Traces|	|Events|	|Methods|	Phase Detection		Duplicate Detection		Clustering		Pattern Mining		Labeling
				time (s)	|phases|	time (s)	|phases|	time (s)	|phases|	time (s)	|phases|	time (s)
NanoXML	235	247,471	100	3.5	225,430	0.1	786	0.5	160	50.8	47	1.7
Javac	19	22,439,965	3919	247.8	51,878	0.0	1605	20.1	650	428.9	57	1.0
JEdit	18	10,129,771	5431	143.6	692,286	0.4	17,088	1994.3	3838	5846.0	84	20.7

4.2 Quantitative Evaluation

To evaluate the computational efficiency of Sage, as well as its effectiveness at reducing the information overload of large execution traces, we conducted a quantitative evaluation. For this evaluation, we used three software subjects: Javac, JEdit, and NanoXML. For each of these three software subjects, we computed the hierarchical phase model for all training test-case executions and report the reduction in model size and time costs for each step in our model building phase. Then, we also provided 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. These experiments were performed on a 1600MHz CPU with 8 cores, with 64-bit Ubuntu 14.04, with 12GB of RAM.

4.2.1 Model Building Results. We present our results for the model building stage of our approach for these three subjects in Table 1. By examining the results in Table 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 are much smaller, which may not be surprising given the smaller size of the subject and the task it performs. Javac and JEdit have a relatively small number of execution traces, but their program and execution size is much larger in comparison to NanoXML.

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. For the same reason, the number of phases for JEdit is almost 41 times as that of NanoXML, and the time cost of JEdit's phase detection is 41 times as that of NanoXML. As a result, the phase detection step identified 225430, 51878, and 692286 preliminary phases in the top-8 levels of NanoXML, Javac, and JEdit, respectively.

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. This expense is especially noticed for JEdit, which requires around 33 minutes to cluster the unique phases for the model building stage.

For the frequent pattern mining step, we observe that this step is the most time-consuming of the model building 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 phases).

It is worth noting that the model building 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.

4.2.2 Application Results. In Table 2, we present our results for applying our Sage approach on 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.

The "time" column of Table 2 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 building procedure is an offline 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 model building results, the efficiency bottleneck of the approach is in the steps of clustering and pattern mining, which consume more than 95% of the total time cost.

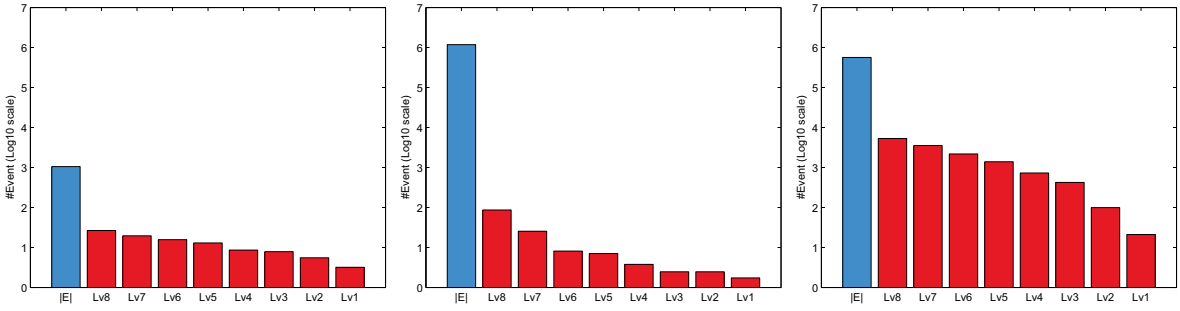
Based on the Table 2, it is clear that the top level of our hierarchical structure is capable of highly abstracting the execution traces: the NanoXML execution abstractions contain an average of 3.2 phases, the Javac execution abstractions contain an average of 1.7 phases, and the JEdit execution abstractions contain an average of 21.1 phases. Figure 5 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 studied levels of the hierarchical execution phase model.

4.3 User Study

To evaluate the degree to which our approach provides comprehensible meanings of actual behavior of execution phases, we conducted a user study. Because understanding the software program is expensive for developers but critical for software development, automatically presenting comprehensible high-level functionality phases from execution traces for developers to reduce this cost is helpful for various software engineering tasks. The goal of this study was to determine whether our approach can build up the cognitive mapping between the high-level software behavior and the low-level implementation. Thus, we evaluate the comprehensibility of the labeled phases with a number of participants.

Table 2: Average number of phases at each level and total time cost for execution abstraction

Project	|trace events|	Level								time
		1	2	3	4	5	6	7	8	
NanoXML	1053	3.2	5.5	7.9	8.6	13.0	15.7	19.6	26.5	0 s
Javac	1,181,050	1.7	2.5	2.5	3.8	7.1	8.1	25.5	86.8	13 s
JEdit	562,765	21.1	99.1	424.4	729.1	1392.8	2182.1	3547.7	5317.8	9.1s



(a) NanoXML **(b) Javac** **(c) JEdit**

Figure 5: Trace size (in blue) and average number of phases at each top-8 levels of the hierarchy (in red). Note log scale.

For this study, we used JEdit as our subject program because it allowed us to invoke high-level functionalities that any participants should understand, despite being unaware of how it was implemented in code.

The study setup involves the following steps. We identified nine typical functionalities of the editor software to represent the high-level functionalities, which are "New File," "Change Font," "Word Count," "Close&Exit," "KeywordHighlight," "Open File," "Typing," "Search," and "Indent." And then, we executed the instrumented JEdit with a combination of several operations drawn from the basic functionalities that we defined. Thus, each of these compound-functionality executions simulate the real usage scenarios, and each of them contains multiple basic functionalities. For example, one execution may include "Open File," then "Typing," then "Change Font," then "Close&Exit."

We totally produced 18 compound-functionality execution traces and input all of them into Sage to produce the labeled phases. Based on our knowledge of the basic functionalities that we invoked to create the execution traces and our familiarity with JEdit, we manually labeled the ground truth for each phase of the execution.

Based on this information, we designed each of the phases into a multiple-choice question. Each question gave one phase identified by Sage, presented as the textual labels produced by the Semantic Labeling step, and five options to describe the high-level functionality of this phase. The five options contain one correct option (*i.e.,* the ground truth), three wrong choices that were randomly selected from the nine high-level functionalities, plus one "Unknown" option.[3] We recruited 28 computer science graduate students as participants. All participants had at least one year of industrial internship experience. None of the participants had previously used JEdit nor read its source code. Each participant was required to label 15 randomly selected questions within 30 minutes.

[3]Sample questions can be found here: https://spideruci.github.io/sagevis-demo/questionnaire

Table 3: The user study results for JEdit

Functionality	|*Questions*|	|*Correct*|	Accuracy
New File	56	46	0.82
Change Font	28	21	0.75
Word Count	56	28	0.50
Close&Exit	70	56	0.80
Keyword Highlight	28	11	0.39
Open File	28	22	0.79
Typing	70	52	0.74
Search	56	36	0.64
Indent	28	24	0.86
Total	420	296	0.70

In Table 3, we present the comprehension accuracy of each of the basic functionalities. 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, respectively. The fourth column represents the accuracy. Based on these results, we observe that the accuracy values range widely, *i.e.,* from 0.39 to 0.86, and the average accuracy is 0.70.

We further investigated the results by talking with some of the participants face to face. Overall, the participants stated that they thought the labels of "New File," "Close&Exit," "Open File," and "Indent" functionalities were for the most part clear and comprehensible. They observed that the top-5 labels of the identified phases contained the keywords revealing the functionality, such as "NewFile," "Close," "Open," and "shiftIndentRight. For the relatively low accuracy of the "Keyword Highlight" and "Word Count" functionalities, we found the most meaningful label of the "Keyword Highlight" functionality is the "KeywordMap." However, we discovered that many 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 that 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.

5 DISCUSSION

In this section, we discuss the evaluation results and answer the three research questions introduced in Section 4.

Addressing Information Overload. In order to address research question **RQ1**, we looked at if our approach was successful in abstracting the overwhelming amount of information presented in the execution trace, thus reducing the information overload a developer would face if attempting to understand an execution trace. In Table 1 we found that the number of meaningful execution events was reduced in each step during the model building stage of the approach. Moreover, when processing an execution for developer inspection (results shown in Table 2), we found multiple orders of magnitude reduction in trace sizes, regardless of subject program and hierarchical level for the top-8 levels. As such, we posit that the approach substantially alleviates the information-overload difficulty for developer comprehension of execution traces.

Addressing Behavior Subsumption. In order 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 Table 2 and in Figure 5. In practice, different maintenance tasks require different understandings of the program. 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 level 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, when exploring deeper in the hierarchy, its sub-behaviors are revealed.

In the results of the Quantitative Study (Section 4.2), in Table 2 and Figure 5, 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 3.2, whereas the average number of behavior phases at Level 8 is 26.5. For Javac, the average number of identified behavior phases at Level 1 is 1.7, whereas the average number of behavior phases at Level 8 is 86.8. For JEdit, the average number of behavior phases at Level 1 is 21.1, whereas the average number of behavior phases at Level 8 is 5317.8.

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.

Addressing Comprehensibility of Execution Traces. In order to address **RQ3**, we need to determine if the approach can help developers understand the executed behaviors based on the labels provided by the approach. In the user study, we demonstrated that the labeled phases enabled developers to comprehend the behaviors of the execution. For 6 out of the 9 studied functionalities, the users achieved accuracy scores of 70% or greater. Moreover, across all 9 studied functionalities, users achieved an overall accuracy of 70%. Hence, we posit that the approach assists with the comprehensibility of understanding execution trace events; however, for some behaviors, the approach produced better results than for others, and thus there may be room for future improvement.

6 THREATS TO VALIDITY

In terms of threats to validity, we cannot claim that our results will generalize to all programs, all traces, and all programming languages. Our evaluations were conducted on three programs that were all written in Java. Although Sage is not language-specific to Java, it is widely known that Java programs often are written with more descriptive method names than programs written in languages such as C. Because the labels applied to the execution phases are based on method names, more descriptive method names will likely produce better comprehension among users, however this is true with or without such tool support.

Another possible threat to validity is that the activities and functionalities identified in the user study may not be representative of those that would be of interest to actual users conducting maintenance tasks. 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 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. Moreover, none of our participants had any knowledge of the implementation of the subject program, which provided some assurance that such knowledge did not influence their choices. However, by controlling for such an external influence, the participants may have less implementation knowledge than the developers and maintainers that we envision to utilize such a system in practice.

7 RELATED WORK

Various researchers have proposed different ways to help developers efficiently gain a better understanding of the program's behavior via automatically mapping the high-level functionalities with source code based on program execution traces.

Pattern Mining for Program Comprehension. Because pattern mining techniques can aggregate trace events, researchers employ them to provide a high-level model of a program execution and ease its comprehension [3, 5, 16, 19, 20, 29, 32].

Fadel *et al.* [19] employed pattern matching techniques to aggregate events of execution traces of Linux kernel. They focused on identifying the entry and exit event pairs to form functional phases, such as function calls, system calls or interrupts. Jivan *et al.* [16] tracked similar execution patterns to generate high-level generic patterns. Their approach employed these patterns to save the storage and computation for the analysis of multi-core systems. Benomar *et al.* [5] presented a search-based method to identify the feature-level patterns from execution traces. Their method is able to form a set of phases that minimizes coupling while maximizing cohesion. To identify features from execution traces for improving program comprehension, Asadi *et al.* [3] proposed a search-based approach to identify conceptually cohesive segments in execution traces. Medini *et al.* [29] presented a dynamic-programming-based algorithm to improve the efficiency of feature localization.

All of these techniques focused on identifying high-level patterns over only a single execution trace. In contrast, our approach applies frequent pattern mining over multiple execution traces to find patterns that exist among multiple executions. Moreover, our approach provides multiple levels of abstraction to allow exploration and comprehension at different levels of granularity.

Trace Reduction and Abstraction Techniques. Due to the massive size of execution traces, scalability issues inevitably arise. To improve the comprehensibility of execution traces, researchers have proposed many trace compression and abstraction techniques to reduce their size. Chan *et al.* [7] used sampling to reduce the size of the traces. Watanabe *et al.* [39] and Zaidman *et al.* [42] 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. Reiss *et al.* [35] encoded repeating events to reduce the size of the execution traces. Additionally, Reiss [34] abstracted execution traces comparing parts of the trace with certain intervals, if they are similar they would be part of the same phase, if not, a new phase was found. However, each approach provides only one level of granularity for the developers. In contrast, we do not omit execution information (*i.e.,* sample), and we add abstractions to the detailed execution trace — at multiple levels of granularity to assist developer inspection and exploration of execution behaviors.

Hamou-Lhadj and Lethbridge [23] compressed the execution traces while retaining the full call tree. However, in our approach we identify similar recurring patterns thus reducing and abstracting the execution trace for comprehension.

A different approach is to use metrics-based filters to reduce the size of an execution trace. Hamou-Lhadj *et al.* [22] 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.

Multi-level Trace Abstraction. Multi-level abstractions of executions enable developers to access arbitrary areas of the execution [18]. Thus, researchers have sought to build multi-level abstractions to assist developers in analyzing and comprehending execution traces. Yang *et al.* [41] build a hierarchical tree that represents low-level dynamic control flow. The goal of this work is to improve the efficiency of path tracing and path querying, whereas with our approach analyzes and abstracts method invocations to assist with developer comprehension.

Jivan *et al.* [15, 17] visualized execution traces in a hierarchical way also using a timeline. By zooming in the user can zoom in on the current level, and when zooming in lower levels will be presented. This work focuses on a visualization that can be applied to unspecified trace abstraction approaches. However, in earlier work Jivan *et al.* [16] provided such a trace abstraction technique targeting kernel-level events (*e.g.,* system calls, disk usage). In contrast, our approach operates on higher-level events such as application-level methods, and thus are likely closer to the level of functionality that application developers would understand.

Lo *et al.* [26] mined an execution trace in a hierarchical fashion to find different levels of granularity and visualize those in live sequence charts to the user. 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. In addition, Medini *et al.* [28] presents SCAN, an automated tool that is able to automatically divide executions and label execution segments. SCAN uses Formal Concept Analysis (FCA) to form the lattice structures over segments of execution traces. The lattice structure maintains the relations between the execution segments of the same execution to find common execution patterns. In contrast, our approach finds commonalities across multiple executions to find behaviors that may be infrequent within an execution but frequent among executions. De Pauw *et al.* [12, 13, 31] visualizes patterns in the executions and present them in a more compact way by clustering recurrent patterns together. Our approach also finds recurrent patterns, but provide further abstraction of the abstraction trace to more compact and comprehensible representations.

8 CONCLUSION

In this paper, we presented Sage, an approach that creates a hierarchical abstraction and multiple levels of granularity of an execution trace for developer inspection, exploration, and comprehension. The approach is composed of a model building stage and an application stage. The model building 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 provide an evaluation that consists of a case-study demonstration, a quantitative evaluation of the computational costs and size reduction of execution traces, and a user study that assessed the understanding of execution behaviors. 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 quantitative evaluation revealed two things: (1) the approach 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 model building costs are not inconsequential, but these costs can be incurred infrequently and offline. Moreover, the application to execution traces under investigation occurs efficiently—for all studied cases, in less than a second. Finally, 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.

Additionally, future work can address other system features. Although our approach is applicable to multi-threaded programs (by analyzing each thread independently), further gains may be possible by analyzing correlations and communications between threads to identify phases. Finally, we will be evaluating Sage on more and larger software subjects.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under award CAREER CCF-1350837.

REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[3] F. Asadi, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc. A heuristic-based approach to identify concepts in execution traces. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 31–40. IEEE, 2010.

[4] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435. ACM, 2002.

[5] O. Benomar, H. Sahraoui, and P. Poulin. Detecting program execution phases using heuristic search. In *International Symposium on Search Based Software Engineering*, pages 16–30. Springer, 2014.

[6] R. P. Buse and W. Weimer. Synthesizing API usage examples. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 782–792. IEEE, 2012.

[7] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 237–244, May 2003.

[8] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 49–58, Washington, DC, USA, 2007. IEEE Computer Society.

[9] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213–222, March 2007.

[10] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[11] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using IR methods for labeling source code artifacts: Is it worthwhile? In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 193–202. IEEE, 2012.

[12] W. De Pauw and S. Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th international symposium on Software visualization*, pages 143–152. ACM, 2010.

[13] W. De Pauw, D. H. Lorenz, J. M. Vlissides, and M. N. Wegman. Execution patterns in object-oriented visualization. In *COOTS*, volume 98, pages 16–16, 1998.

[14] N. DiGiuseppe and J. A. Jones. Semantic fault diagnosis: Automatic natural-language fault descriptions. In *International Symposium on Foundations of Software Engineering*, 2012.

[15] N. Ezzati Jivan. *Multi-Level Trace Abstraction, Linking and Display.* PhD thesis, École Polytechnique de Montréal, 2014.

[16] N. Ezzati-Jivan and M. R. Dagenais. An efficient analysis approach for multi-core system tracing data. In *Proceedings of the 16th IASTED International Conference on Software Engineering and Applications (SEA 2012)*, 2012.

[17] N. Ezzati-Jivan and M. R. Dagenais. Multiscale navigation in large trace data. In *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–7, May 2014.

[18] N. Ezzati-Jivan and M. R. Dagenais. Multi-scale navigation of large trace data: A survey. *Concurrency and Computation: Practice and Experience*, 29(10), 2017.

[19] W. Fadel. Techniques for the abstraction of system call traces to facilitate the understanding of the behavioural aspects of the linux kernel. Master's thesis, Concordia University, 2010.

[20] R. L. C. Fonseca. *Improving visibility of distributed systems through execution tracing.* University of California, Berkeley, 2009.

[21] D. J. Gilmore. Models of debugging. *Acta psychologica*, 78(1-3):151–172, 1991.

[22] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 112–121, March 2005.

[23] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 159–168. IEEE, 2002.

[24] A. K. Jain and R. C. Dubes. Algorithms for clustering data. 1988.

[25] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[26] D. Lo and S. Maoz. Mining hierarchical scenario-based specifications. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 359–370, Nov 2009.

[27] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society, 2001.

[28] S. Medini, V. Arnaoudova, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, and P. Tonella. SCAN: an approach to label and relate execution trace segments. *Journal of Software: Evolution and Process*, 26(11):962–995, 2014.

[29] S. Medini, P. Galinier, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. A fast algorithm to locate concepts in execution traces. In *International Symposium on Search Based Software Engineering*, pages 252–266. Springer, 2011.

[30] V. K. Palepu and J. A. Jones. Revealing runtime features and constituent behaviors within software. In *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, pages 86–95, Sept 2015.

[31] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang. Visualizing the execution of Java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, UK, 2002. Springer-Verlag.

[32] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6), 2007.

[33] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 371–382. IEEE, 2009.

[34] S. P. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the Third International Workshop on Dynamic Analysis*, WODA '05, pages 1–6, New York, NY, USA, 2005. ACM.

[35] S. P. Reiss and M. Renieris. Encoding program executions. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 221–230, May 2001.

[36] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, and F. I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 155–164. IEEE, 2005.

[37] G. Salton and J. Michael. Mcgill. 1983. *Introduction to modern information retrieval*, 1983.

[38] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage API usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 319–328. IEEE Press, 2013.

[39] Y. Watanabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 8–14, New York, NY, USA, 2008. ACM.

[40] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 808–819. ACM, 2016.

[41] C. Yang, S. Wu, and W. K. Chan. Hierarchical program paths. *ACM Trans. Softw. Eng. Methodol.*, 25(3):27:1–27:44, Aug. 2016.

[42] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 329–338, March 2004.