Inferring Hierarchical Motifs from Execution Traces

Saba Alimadadi*
Northeastern University
Boston, MA, USA
saba@northeastern.edu

Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, BC, Canada
karthikp@ece.ubc.ca

ABSTRACT

Program comprehension is a necessary step for performing many software engineering tasks. Dynamic analysis is effective in producing execution traces that assist comprehension. Traces are rich sources of information regarding the behaviour of a program. However, it is challenging to gain insight from traces due to their overwhelming amount of data and complexity. We propose a generic technique for facilitating comprehension by inferring recurring execution motifs. Inspired by bioinformatics, motifs are patterns in traces that are flexible to small changes in execution, and are captured in a hierarchical model. The hierarchical nature of the model provides an overview of the behaviour at a high-level, while preserving the execution details and intermediate levels in a structured manner. We design a visualization that allows developers to observe and interact with the model. We implement our approach in an open-source tool, called Sabalan, and evaluate it through a user experiment. The results show that using Sabalan improves developers' accuracy in performing comprehension tasks by 54%.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

KEYWORDS

Program comprehension, behavioural model, hierarchical motifs

ACM Reference Format:
Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2018. Inferring Hierarchical Motifs from Execution Traces. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180216

1 INTRODUCTION

Program comprehension is an essential first step for many software engineering tasks. Developers spend a considerable amount of time understanding code. About 50% of maintenance effort is spent on comprehension alone [16]. Unfortunately, code understanding is challenging. To understand code, developers typically start by *searching* for clues in the code and the environment. Then they go back and forth on the incoming and outgoing dependencies to *relate* pieces of foraged information. Throughout the process, they *collect* information they find relevant for understanding the code on an "as-needed" basis [42]. However, developers often fail in searching and relating information, and lose track of relevant information when using such ad-hoc strategies [70]. Further, developers form mental models of code, that are often inaccurate [60]. Thus, there is a need for systematic and automated techniques for program comprehension [46].

Dynamic analysis, which collects and utilizes data (traces) from program execution [20], is a popular technique for program comprehension. However, due to the amount of information obtained during the execution, the traces tend to become complex and overwhelming, and thus difficult to understand [17, 80]. Existing techniques target this problem, e.g., by summarizing traces [32], structuring and visualizing collected data [2, 3, 29], or inferring system specifications [61]. However, the first technique loses some of the data that may still be valuable, and the rest become overwhelming for developers and are not flexible to small variations of data. The problem can also be approached by finding patterns in the execution. However, prior work in the area has predominantly focused on generic and predefined design patterns, low-level architectural relations between program artifacts, or visualizations of all details of execution [3, 13, 35, 43]. While useful, these approaches do not capture the behavioural patterns that are neither defined nor known prior to analysis, but form and recur (with small variations) throughout the execution of a program. Even in more traditional programming languages, patterns in execution do not repeat exactly in the same manner or same sequence. Further, presence of programming languages features such as dynamism, asynchrony and non-determinism in the execution makes the analysis more problematic and burdensome, and renders conventional techniques ineffective. Hence, program comprehension through dynamic analysis still remains challenging [20].

In this paper, we propose a novel technique for program comprehension by inferring a model of execution *motifs*. Motifs are abstract and flexible recurring patterns in program execution that serve a specific purpose in the functionality of the program. The term is inspired by *sequence motifs*, which are recurring patterns in DNA sequences that have a biological function [24]. Our approach discovers motifs from traces containing function executions and events. Our proposed algorithm compares a trace obtained from an interaction session against a database of previously-collected traces. It iteratively examines segments of traces for detecting sequences of function executions that may recur in execution. It is tolerant of small variations in different manifestations of each motif, allowing abstraction in inferred motifs. The algorithm discovers hierarchies

*Work performed while a PhD student at the University of British Columbia (UBC).

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180216

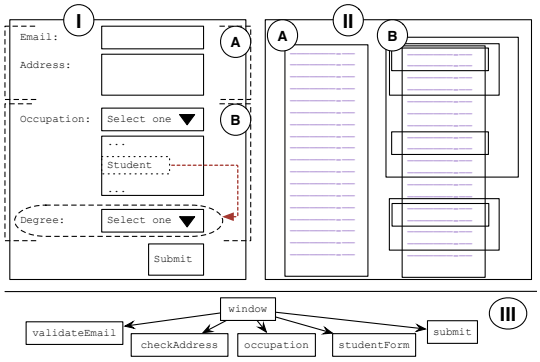

Figure 1: I: A sample registration form. II: A) Sample execution trace, and B) hierarchy of inferred motifs. III: Dynamic call graph of example

between motifs as they emerge from details of execution. The hierarchical structure of inferred motifs reveals how higher-level key points of execution are formed. It allows users to have an overview of the trace, while still having access to all execution details as well as all intermediate levels.

The main contributions of our work are as follows:

- We propose an automated approach for inferring a model of program behaviour, which encompasses hierarchies of abstract recurring motifs extracted from execution traces. Our approach is inspired by techniques from bioinformatics, where similar challenges arise in investigating similarities in large sequences of DNA. The motifs facilitate program comprehension by highlighting the main characteristics of behaviour, and abstracting the details and variations of execution.
- We design and build a visualization technique for presenting the motifs to developers, to provide assistance with program comprehension. Our method is complementary to existing tools and techniques, and is designed to be utilized alongside existing programming environments.
- We implement our approach in a tool called SABALAN that supports JavaScript-based web applications. Our tool is non intrusive for generating traces, and infers models of recurring motifs from execution trace in an automated manner.
- We evaluate our approach through a controlled experiment conducted with 14 participants, on a set of real-world program comprehension tasks. The results show that using SABALAN helps developers perform program comprehension tasks 54% more accurately than other tools.

2 CHALLENGES AND MOTIVATION

To assist the process of *searching*, *relating* and *collecting* information, many techniques collect execution traces, analyze them, and/or visualize the results for the developers. Despite providing the grounds for precise analyses, dynamic traces become very large and cause information overload. Further, they become very complex due to dynamism, asynchrony and non-determinism in program execution. These challenges render large traces ineffective in assisting program understanding.

```
<form>
  Email: <input type="email" id="email">
  Address: <input type="text" class="addr">
  Occupation: <div class="dropdown" id="occupation">
    <button class="dropbtn">
      Choose one</button>
    <div class="dropdown-content">
      <a href="#">Academic</a>
      <a href="#">Industry</a>
    </div>
  </div>
  <input type="submit" value="submit">Submit</input>
</form>
```

Figure 2: Initial DOM state of the running example.

```
$("#email").addEventListener("change", validateEmail, false);
$(".addr").click(checkAddress);
$(".dropdown-content").addEventListener("change", occupation, ↩
    false);
function validateEmail () {
  // do stuff
}
function checkAddress() {
  // do more stuff
}
```

Figure 3: [Partial] JavaScript code of the running example.

In this section, we use a simple example to illustrate these challenges (Figures 1–3). We selected JavaScript for the examples since it is the lingua franca of web development. It has been recently voted as the most popular programming language on StackOverflow [75], and is the most used language on GitHub [45]. JavaScript applications are highly dynamic, asynchronous and event driven, and heavily interact with the Document Object Model (DOM) and the server code [1]. These features can help demonstrate trace complexity within small code segments. While our approach is general, we use JavaScript in this paper to demonstrate it.

Overload by Information in Large Traces. The amount of information a trace carries matters due to the cognitive load that understanding the trace imposes on developers [19], e.g., a study found that one GB of trace data was generated for every two seconds of executed C/C++ code [68]. For modern applications, which are often distributed among many nodes with many components involved, the traces become incomprehensible for developers very quickly. Some techniques try to address the problem by reducing the trace during/after its collection [12, 32, 68] by focusing on more important entities, or filtering the details of the executions. These techniques have been able to make traces more useful by decreasing the information contained in the traces [37]. However, even with a technique that creates a smaller trace, the trace is still not necessarily understandable for developers, as some of the data might be lost or missed by developers.

Complex and Hidden Dependencies. Revealing abstract and higher-level patterns that highlight the key points of a program's behaviour can facilitate comprehension. The focus of the developer can be guided through a hierarchy of recurring patterns of execution, while all collected information are still preserved for further inquiry. However, extracting such patterns (motifs) is challenging due to the dynamism, asynchrony and non-determinism in program execution, expecially in JavaScript applications.

First, there are many complex and hidden dependencies between entities in the system, which can affect the execution. Understanding the impact of a user action or asynchronous communication with the server are examples of relations that are difficult, if not impossible, to capture from merely analyzing the code or the call graph. They act as media for connecting segments of execution together, that otherwise would not be related in the code itself. Further, for a part of behaviour to be distinguished as a motif, it should recur during the execution. Different executions of what is conceptually the same motif, may vary in details and thus may not converge to reveal the same motif. The alterations are intensified when programs have user interfaces, are distributed, or involve general dynamism and asynchrony. However, such variations should not prevent the analysis from recognizing the high-level blueprint of the behavioural motif they all manifest. An analysis that is overly dependent on execution details may not allow higher-level motifs to reveal themselves.

Example. Consider the example shown in Figure 1, showing a part of a form required for registering a user. Specific events on the input fields of the form have handlers that validate the input before the form can be submitted. `verifyEmail()` and `checkAddress()` (lines 1–2 of Figure 3) are handlers for `email` and `address` fields of part (A) of the form (lines 2–3 of Figure 2). The two function are always executed together in a successful registration scenario, and are a consistent part of the motif representing that scenario, due to their placement in the DOM. However, this relation cannot be inferred from the code (Figure 3) or the call graph (Figure 1.III).

Moreover, a successful submission requires proper submission of all the fields. However, the form can change in section (B) of Figure 1 based on the input of `occupation` (lines 4–11 of Figure 2). If the user chooses `Student`, a drop-down menu appears and the appearance, content, and functionality of the form changes based on user's input. However, the conceptual purpose of submitting the form, and hence the motif, remain the same. Should an analysis be too dependent on exact execution details, these two executions will be considered different. However, a more representative analysis should recognize that regardless of occupation of the user, the essence of the motif is the same and it should support both options. There is often neither prior knowledge nor templates of the application-specific motifs. Hence, a useful comprehension method should accomodate a degree of flexibility in inferring motifs, to allow abstract motifs to form independently from unimportant contextual details.

3 OVERVIEW OF THE METHODOLOGY

For execution traces, such as the one depicted in Figure 1.II.A, our goal is to infer a hierarchy of its recurring motifs, by utilizing the knowledge of previous executions of the application. The model of extracted motifs assists comprehension of the program behaviour by facilitating the cycle of *searching*, *relating*, and *collecting* information. Having our proposed approach, developers are able to gain an overall understanding of the highlights of execution, manifested as motifs, at a glance. Further, they would have the means to understand the details of such motifs, their hierarchies, and relations upon inquiry (Figure 1.II.B). Our approach takes advantage of precision of dynamic analysis, but prevents developers from being trapped and overwhelmed by the execution details.

Our proposed approach first instruments and intercepts the application on the fly, to obtain traces. Having a knowledge-base of previously collected traces and a query trace, our algorithm then extracts motifs of different lengths within traces, and infers hierarchies and relations of motifs. Our algorithm is inspired by bioinformatics algorithms for aligning biological sequences. Finally, our approach creates a behavioural model from the motifs and their relations, which we visualize for developers.

Execution Traces. To obtain the traces required for the algorithm, we *instrument* the applications and collect dynamic execution information automatically. Our instrumentation allows our technique to *intercept* all function executions and collect their context-sensitive information. It also intercepts all events that can occur during the execution. Their added knowledge can assist the algorithm in inferring conclusions about motifs and their causal and temporal relations with (a)synchronous events. Next, our approach eliminates low-level details included in the raw trace, such as auxiliary events, low-level and library method calls, and framework-specific details. The pruned trace is then used as the input to the algorithm. Note that our method is non intrusive, and preserves the original behaviour of the application under investigation.

4 ALGORITHM FOR INFERRING MOTIFS

In this section, we propose our algorithm for detecting motifs in order to create a higher-level model of behaviour. We define motifs as abstract and hierarchical sequences of function executions that recur throughout the lifetime of an application. While each motif eventually supports concrete sequences of function executions, it is by nature a composite element, and can represent more complicated structures. We define a motif as an ordered set of two or more members (m_0 to m_i), which include [sub-]motifs, abstract entities, and context-sensitive function executions. The confidence of a motif in each of its members is representative of the manner that member is observed within different executions of the motif, and is shown as c_0 to c_i for all motif's members, respectively.

$$M = \{\langle m_0, c_0\rangle \dots \langle m_i, c_i\rangle|_{i=1}^{\infty}\} \quad , \qquad \begin{array}{lcl} m_i & ::= & \text{function execution} \\ & | & \text{sub-motif} \\ & | & \text{abstract entity} \end{array}$$

Our approach draws the attention of developers to the main observed motifs, presumed to represent highlights of behaviour, preventing their view to be obstructed by low-level details. The underlying model still preserves details that can be demanded by users as necessary.

4.1 Inspiration from Analyzing Biological Sequences

In designing the algorithm, we were inspired by bioinformatics, where there is a constant need to explore, compare, and analyze large data sequences. Most relevant to our approach are *sequence alignment* algorithms, which find similarities in sequences of DNA, RNA, and protein by arranging and comparing them [31]. We begin our core algorithm by using a heuristic for finding exact matches between trace sequences. For comparison of the trace sequences, we adapt the idea of BLAST (Basic Local Alignment Search Tool) [4], a *local sequence alignment* algorithm which we modify to fit the

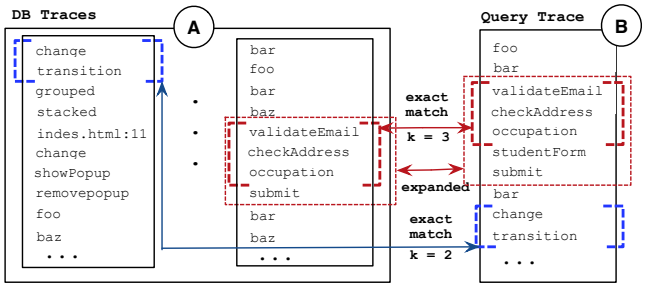

Figure 4: This figure depicts a DB of traces (A) and a sample query trace (B) of an application, on the left and right side, respectively. Exact matches of length 2 and 3 between the query trace and different DB traces are marked.

domain of execution traces. We then expand the matches by maximizing similarities in their neighbouring entities in the traces. This phase is accomplished using a dynamic programming algorithm in order to allow more flexible and more abstract motifs. Throughout this process, our method infers existing motifs and reveals their relations and hierarchies. In this section, we use Algorithms 1 – 2 and Figures 4 – 5 to explain our algorithm. Our algorithm takes the application (app) and its knowledge-base ($\Sigma\Sigma db$) as input and returns the extracted motifs as output. Please note that we have eliminated and/or merged many details for the sake of brevity. The details can be found in the repository of our open-source tool [71].

4.2 Forming a Knowledge Base

Our algorithm requires a knowledge base of multiple previous executions, i.e., a set of traces, named *database (DB) traces* ($\Sigma\Sigma db$), which can initially be collected by executing the test suite, crawling, or exploratory testing and exercising the application multiple times. During each interaction session, our approach collects a trace, called the *query trace* (Σq), which will be analyzed and compared against all DB traces for finding its motifs. Each query trace is itself added to the DB traces after the algorithm is finished. This part is depicted in lines 2–4 of Algorithm 1. Parts A and B of Figure 4 display sample DB traces and query trace of the running example (Figure 3).

4.3 Finding Exact Matches

Next, the algorithm finds all the exact matches of length k between the query trace and the DB traces. We start by matches of length 2 (function pairs). We then increment the length of exact matches iteratively and repeat the search at each iteration, until we have found all exact matches. Two sets of exact matches of length 2 and 3 are shown in Figure 4, between 2 DB traces (part A) and the query trace (part B). Lines 7–8 of Algorithm 1 iterate over the query trace for finding matches of length k, and increment k at each iteration. Lines 9–10 show that subsequences of length k are extracted from the query trace at each iteration, and are compared against all k-length subsequences of all DB traces to find matches.

4.4 Allowing Abstraction in Motifs

In the next step, we expand each match to progress towards finding flexible and abstract motifs, that are tolerant of small alterations. This technique decreases dependency on specific execution details,

Algorithm 1 Finding exact matches and expanding them

```
1: procedure ExtractPatterns(app, ΣΣdb)        ▷ app: application under analysis, ΣΣdb:
   algorithm knowledge base (DB traces)
2:     modifiedApp ← instrument(app)
3:     rawTrace ← intercept(modifiedApp)
4:     Σq ← prune(rawTrace)
5:     k ← 2
6:     motifs ← ϕ
7:     for i ← k; i ≤ Σq.length; i ++ do
8:         for j ← 0; j ≤ Σq.length - k; j ++ do
9:             subQ ← extractSubTrace(k, Σq, i, j)
10:            matches ← exactDbMatches(ΣΣdb, i, subQ)
11:            for m ← 0; matches.length; m ++ do
12:                for n ← 0; n < matches[m].length; n ++ do
13:                    dir ← initialExpansionDirection()
14:                    ΣI ← [ qI_start   qI_end
                              dbI_start  dbI_end ]
15:                    if expandable(Σq, ΣΣdb[m]), ΣI then
16:                        subDb ← matches[m][n]
17:                        expandedQ ← Σq.expand(sub_q, ΣI_q, dir)
18:                        expandedDb ← ΣΣdb[m].expand(sub_db, ΣI_db, dir)
19:                        dir ← dirExpansion(toggle: true, ΣΣdb[m], Σq, ΣI)
20:                        k.increment()
21:                        motif ← compare(expandedQ, expandedDb, k)
22:                        matches.push(motif, k)
23:                        motifs.add(motif)
24:                        ΣI ← adjustExpansionIndices(dir)
25:                    end if
26:                end for
27:            end for
28:        end for
29:    end for
       return motifs
30: end procedure
```

provides a higher-level overview of the semantics of the application, and permits flexible motifs of variable length that may include gaps.

At this stage, our algorithm iteratively performs the following steps. **First**, it selects two matches from existing matches, which were determined as the result of previous step (Figure 4). Then, it iteratively expands the query and DB matches from both directions, while gradually incrementing the length of the motif under investigation (Lines 11–27 of Algorithm 1). Figure 4 also shows an expanded match of two [partially] different sequences, for measuring their similarities (Figure 4. A). This phase continues until the accumulated penalty of gaps interrupts expansion of the motif.

Next, the algorithm finds a [sub-]sequence of those matches that have the maximum similarity. At this step, we adapt a dynamic programming algorithm called *Smith-Waterman* for finding patterns in two molecular sequences [74]. This algorithm quantifies the sequence alignment process by assigning scores for matches and mismatches, and penalties for gaps. Aligned sequences are then found by searching for the highest scores in scoring matrices. To adapt this algorithm to our domain and compare similarities of two traces, we propose a similarity matrix that determines the similarity of two members in traces. The similarity of the traces are determined by a combination of similarities of all their members, based on a dynamic programming heuristic.

Similarity Matrix. We propose a similarity measure for quantifying the similarities between function executions in traces. Our comparison is based on two metrics, function names and parameters. We devise three scores for comparison: strong match, weak match, and mismatch (leading to a carryover penalty). If two functions match in terms of names and parameter numbers they are a strong match. The match is weak if only the names are equal (and not parameter numbers). The reason for considering parameter

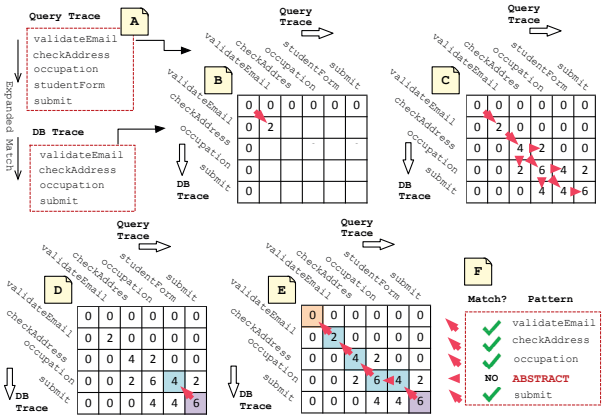

Figure 5: This figure briefly depicts (A) the process of taking two expanded trace subsets, (B, C) forming a scoring matrix based on similarities between sub-traces, and (D, E) finding a match in manner that maximizes the similarities between sub-traces. The final motif can be seen in (F).

count as a separate metric is the nature of JavaScript, where a function call does not need to be faithful to the function signature in terms of arguments (known as function variadicity). Two function executions do not match if both metrics are different. The strong and weak matches are assigned positive scores, while the mismatch is assigned a negative score as it can represent gaps in the motif, which can accumulate and disrupt a motif. The base scores for matches and penalties are determined using empirical data.

Moreover, our matrix needs to support comparison of motifs, to accommodate another extension of the original algorithm, which permits hierarchies between motifs. The function execution members of a motif are compared as explained above. Should a motif contain an abstract node, then all valid executions of the abstract node should be compared with the other sequence. Throughout the process of comparing motifs, our algorithm infers hierarchies and abstractions of motifs should they exist, as explained below.

Then, we perform our adaptation of the Smith-Waterman algorithm on the two expanded sequences, as shown in line 2 of Algorithm 2, which creates a scoring mechanism for comparing the two sequences based on a one-by-one comparison of all their entities, based on our similarity matrix. The result is a scoring matrix of overall scores of comparing two sequences ($M^{k+1,k+1}$). This process is shown for the two sequences of the running example from the previous step in Figure 5.A–C. To find the common motif in these sequences, we find the sub-sequences that hold the highest collective similarity as a group. We start by finding the highest score in the matrix (line 3 of Algorithm 2, Figure 5.D), and then trace the matrix back, determining the aligned motif at this stage (lines 4–10 & Figure 5.E). For navigating the motif back in the matrix, our dynamic programming algorithm chooses the maximum neighbouring score at each step (line 5). Based on the selected neighbour, the algorithm determines whether that motif member comes from one or both of sequences, and whether an abstract entity should be injected to show different alternatives of the motif. The motif's confidence in that member is then updated based on

Algorithm 2 Inferring motifs

```
1:  procedure COMPARE(S1, S2, k)          ▷ S1, S2: two trace sequences, k: sequence length
2:      M^{k+1,k+1} ← SMITHWATERMAN(S1, S − 2, k)
3:      ⟨i_max, j_max⟩ ← MAXSCORELOCATION(M^{k+1,k+1})
4:      while i_max > 0 AND j_max > 0 do
5:          dir ← MAXNEIGHBOUR(i_max, j_max)
6:          if dir == DIAG then MOTIF.INSERT(ABSTRACT(S1[i_max], S2[j_max]))
7:          elseMOTIF.INSERT(FUNCTION(S1[i_max] == S2[j_max]))
8:          end if
9:          ⟨i_max, j_max⟩ ← BACKTRACK(M^{k+1,k+1}, S1, S2, dir)
10:     end while
11:     if S2.type == MOTIF then BUILDHIERARCHY(S2, MOTIF)
12:     end if
        return MOTIF
13: end procedure
```

how it is selected (lines 6–7 of Algorithm 2). The inferred motif of the running example (Figure 5.F) has five members, one of which is abstract. The abstract member was advised to enable the motif to support both sequences shown in Figure 5.A, which have the same functionality (registration), but are executed in a slightly different manner. The abstract member demonstrates that observing the `studentForm` function in the motif is arbitrary, and the motif can either be observed either with five total members including the function, or with only four members which do not include function.

4.5 Inferring Hierarchies of Motifs

In the next step, we devise another extension to the original algorithm, which enables us to infer and reveal hierarchical relations between motifs. By definition, our motifs are composite entities, which can contain other motifs as their members. During the analysis, our algorithm may encounter cases where (1) the match that is being compared is a motif itself, and (2) the expansion leads to discovery of a new motif. In such cases, a hierarchical relation is added between the two motifs. This means that not only the [sub-]motif was observed independently within the execution, but it also contributes to the formation of the new and larger motif. Our analysis follows a bottom-up approach, starting with function executions as building blocks of the trace. It iteratively works the way up to higher-level and more abstract motifs that allow flexibility in execution. At each iteration of the algorithm, new motifs can be revealed which may have hierarchical relations with existing motifs. As such motifs emerge, our algorithm captures the process of their formation and hierarchies in a model (Section 5).

In the running example, we first find an exact match with $k = 3$, which is a motif itself, but with no abstraction (Figure 4). Later, during expansion, we find that this motif is a member of a larger motif (Figure 5.F) with two other members (an abstract member and a function execution). The algorithm creates a hierarchy (line 11 of 2), which then manifests in the model (Figure 6), as an edge from the new abstract motif (node 1) to the sub-motif (node 2).

5 CREATING AND VISUALIZING THE MODEL

In this section, we explain our methodology for inferring the hierarchical model of behavioural motifs and visualizing it.

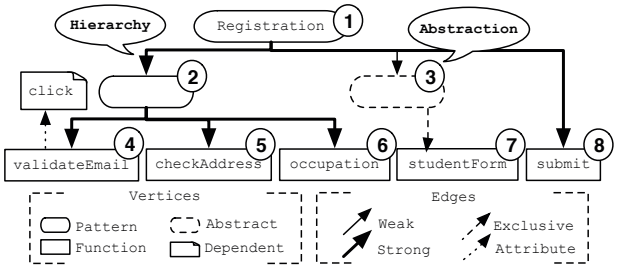

Figure 6: Sample model of the running example. The root node (1) is the highest-level inferred motif. Node 2 is a sub-motif of node (1), marked by the hierarchical edge between the two. Node 3 is abstract allowing variations of its child node to occur in the motif. The leaves of (nodes 4–8) are concrete function executions in the trace.

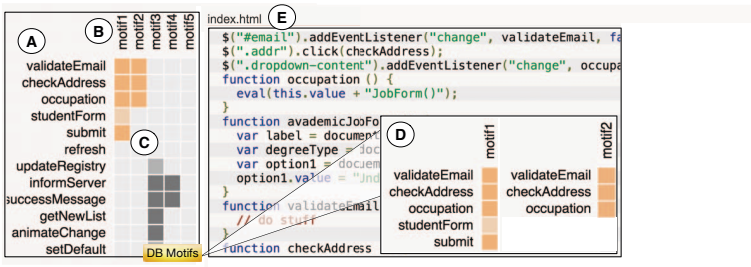

Figure 7: A [modified] screenshot of visualization. (A): Query trace. (B): Inferred motifs depicted on the table. (C): Motif hierarchies. (D):All motifs. (E): Code panel displaying selected function/motif code.

5.1 Creating the Motif Model

As mentioned above, during the process of extracting motifs, our approach infers the hierarchies and other potential relations between them. Such structural relations are preserved in a model, represented as a directed acyclic graph (DAG), which evolves as the algorithm proceeds, as explained below.

Vertices. Vertices of the graph can be functions, motifs, abstract entities, or dependent vertices. *Function* vertices are atomic nodes representing specific and context-sensitive function executions. *Motif* vertices are semantically composite classes. Each motif conceptually contains an ordered set of its members. *Abstract* vertices are the model's means for supporting flexibility in motifs. Should there be alterations in different observations of a motif, an abstract node is used for accommodating all valid cases. *Dependent* vertices hold additional attributes of other types of vertices. and exist only to provide more information about other vertices. e.g., an event contributing to a function execution is shown as a dependent vertex.

Edges. The vertices of the graph are connected through directed and ordered edges. The edges are responsible for connecting motifs to their members. The *direction* of an edge is from a motif node to its members, which are *ordered* based on the time they were observed in traces. The edges also represent the *confidence* of the algorithm in the respective member (strong or weak), based on the manner of observation of the member. Edges may contain other *special attributes*, depending on its type and the purpose it serves in the model. For instance, the "edge exclusion" property is used to show that only one of the variations of an abstract node is valid at a given time.

Figure 6 represents the model of the running example of Figure 1. The root of the DAG (node 1) is a motif representing registration. It consist of three members: a sub-motif (2), an abstract node (3), and a function execution (8). The first member is a motif itself, which contains a sequence of three functions from the trace, marked 4–6. The *strong* edge to the sub-motif (and from there to its children) shows the high confidence of the algorithm in the sub-motif. Node 3 is an abstract node, acting as a place holder for valid versions of the node, manifested in its children. This node exists due to the variation in two observations of the motif (Figure 5.A). In the case of our example, the *exclusive* type of the child (7) edge demonstrates that occurrence of this node is optional in the motif (`studentForm` is observed or not). Further, the *weak* edge type displayed the algorithm low confidence in this node. Node 8, the final member of 1, is an execution of function `submit`. All leaves of the DAG are concrete function executions from the trace. Nodes at higher levels of a graph involve abstractions and hierarchies, to represent the incremental process of emergence of motifs from details of trace.

Motif Relations. The hierarchies that form between motifs are one type of relations that are preserved through the model. The algorithm also discovers other types of relations between motifs, such as *temporal* or *causal* relations. The integration of all these relations depicts how semantics of the program are shaped from bottom (small specific motifs) to the top (larger and more abstract motifs representing key points of behaviour). The motifs may be semantically related in manners that are not quite obvious from the code. For instance, motif `m1` may *cause* motif `m2`, or they may be ordered (but not dependent) due to the design and the architecture of the system. Querying the model allows us to reveal patterns of motifs themselves and even discover patterns that are not known to the developer, are created unintentionally, or are imposed on the system by other factors such as third-party frameworks.

5.2 Visualizing the Model

Finally, we visualize the motifs to further assist program comprehension by taking advantage of information visualization techniques. Our web-based visualization provides two main views for displaying (1) the motifs recorded in a specific query trace, and (2) all motifs discovered in the behaviour (DB traces).

Trace Motifs. To allow developers to focus only on a part of behaviour that is of interest to them, this view displays motifs that are found within the query trace, freshly recorded from an interaction session (Figure 7, A and B). Section (A) of the figure displays the pruned query trace, where time proceeds from top to bottom. Section (B) displays the motifs, distinguished by colour and index. The saturation of each cell of a motif displays the motif's confidence in that member. Each motif may recur multiple times in the same trace, or may contain hierarchies of motifs (Figure 7C).

All Motifs. The second view is meant to provide a global overview of application behaviour by displaying all its motifs, extracted from all DB traces (Figure 7D). These motifs may display system use-cases, feature implementations, or other higher-level sequences

that somehow describe the functionality of the system. They do not conform to a single trace, and thus each motif has its own separate trace. In both views, hovering the mouse over each entity displays more information regarding that entity in the tooltip. Clicking on an entity displays its respective code (function or motif) on the code panel (Figure 7E). Displaying only the code relevant to the motif, allows developers to only focus on their specific task, without the added burden of understanding the whole application.

6 IMPLEMENTATION: SABALAN

We implemented our approach in an open-source tool, called Sabalan. The entire tool is implemented in JavaScript. We create our own Express.js server for implementing the algorithm, the instrumentation unit, and the visualization. We develop the bioinformatics-inspired algorithms from scratch for execution traces. We use a proxy to automatically inspect applications [38]. For instrumenting the code, we create an AST of the code, modify it, and serialize it back into JavaScript [25–27]. Sabalan is publicly available [71].

7 EVALUATION

We empirically evaluate our approach by investigating the characteristics of the extracted motifs, as well as the usefulness of our approach for developers and its overhead through the following research questions.

RQ1. What are the characteristics of typical motifs inferred by Sabalan from execution traces?

RQ2. Does using Sabalan improve developers' performance for common comprehension tasks?

7.1 Motif Characteristics

To address RQ1, we performed our analysis on seven JavaScript applications, listed in Table 1.

Design. We selected seven open-source JavaScript applications from GitHub (Table 1). These applications cover various software domains and were selected based on their popularity and usage. Based on each application's specifications, we selected a method for collecting its traces (running the test suite or designing scenarios for exploratory testing). We provided the traces (DB and query) as input, and analyzed their extracted motifs and investigated their main characteristics. We measured the number of unique motifs inferred from all traces for each application, calculated their lengths, and analyzed their hierarchies. We registered the size of DB traces in terms of number of different traces as well as the size of a trace.

Results and Discussion. The results of the analysis are depicted in Table 1. The second column displays the number of lines of code for each application, while the third column contains the number of motifs found in the applications. The next column represents the number of DB traces collected for each application. Column five, shows the average size of traces of each application, collected by Sabalan in one-minute interaction sessions. Note that our tool performs a level of filtering while logging execution details. Column six shows the average trace size collected by Google Chrome's JavaScript profiling and Timeline. It can be seen that the average trace size using Sabalan is 77 KBs, while without Sabalan there is an average of 96 MBs of data for the same interaction session. These values emphasize the extent of the information contained in the raw traces, even for modest-sized applications, which makes it challenging for developers to analyze them. However, using our approach, developers have an average of 8 recurring high-level motifs for each interaction session, each with an average length of 4 (columns 7–9), to guide them through the understanding of the behaviour. The last column displays the number of unique hierarchical relations between unique inferred motifs. The numbers show the existence of hierarchies of motifs. Further assessment of the structures of model graphs depict the bottom-up formation of higher-level key points of behaviour based on smaller motifs through such hierarchies.

Table 1: Characteristics of traces and inferred motifs

						Motif Length			
Application	**LOC**	**# of M.**	**#of DB**	**Trace size (KB)**	**Raw Trace (MB)**	Avg	Min	Max	**# of unq H.**
Phormer	6000	13	20	84	86	4	2	11	4
same-game	229	4	7	255	143	3	2	4	0
simple-cart	9238	4	19	45	67	4	2	8	3
browserQuest	36206	17	15	67	125	5	3	9	2
adarkroom	15612	6	15	41	40	4	2	6	2
doctored.js	3534	4	10	16	102	3	2	5	1
hextrix	5154	7	16	30	110	4	2	6	2
Average	10853	8	14.5	77	96	4	2	7	2

There are a few cases in the results where the algorithm was not able to find many (meaningful) motifs, or any hierarchies. Upon further investigation, we found that these applications rely heavily on external and graphic libraries, which we had disabled in our analysis. These features can be activated in the future if needed.

An important factor that can impact the efficacy of the algorithm is the *requirements of the DB traces*. The number of initial traces in the knowledge base, their coverage of the application's functionality, and their similarities (or differences), are factors that can impact the quantity and quality of the final motifs. We aimed to maximize the features we covered with the DB traces. We stopped collecting new DB traces when we observed that adding a trace did not affect the inferred motifs (average of 14.5 DB traces per application).

7.2 Controlled Experiment

Next, we conducted a controlled experiment to assess the effectiveness of our technique for developers in practice and address RQ2. We divided the participants into control and experimental groups. The experimental group used our approach, while the control group used the tool of their choice. The participants accomplished a set of comprehension tasks, and their performance was measured. The tasks were designed based on common software comprehension activities [63]. We defined the performance of a developer by the combination of time and accuracy of completing the tasks. Our hypothesis was that using our approach would enhance developers' performance in understanding the overall behaviour, main use-cases, and recurring motifs of a web application.

7.2.1 Experiment Planning. The goal of our experiment is to investigate the following research questions.

RQ2.1. Does using Sabalan decrease task completion *duration* for common comprehension tasks?

RQ2.2. Does using Sabalan increase task completion *accuracy* for common comprehension tasks?

RQ2.3. Is Sabalan better suited for certain types of comprehension tasks?

Table 2: Comprehension tasks used in the study.

Task	Description	Activity
T1.A	Understanding all common usecases	A1, A7, A9
T1.B	Determining the most used scenarios	A6, A7
T2.A	Locating the implementation of a feature for reuse	A1, A3
T2.B	Estimating the quality of said implementation	A4, A5, A8
T3	Understanding the addition of a new feature	A1, A2, A3

Variable Selection. Our design involved one *independent variable (IV)*, the variable we controlled, which was the type of tool used, i.e., a nominal variable with two levels. We refer to the first level as SABALAN, since they had to use our tool. The second level represented usage of other tools, which we refer to as OTHER. Our goal was to measure developers' performance in completing the tasks. We quantified it using two variables, task completion *duration* and *accuracy* (both continuous), that were our *dependent variables (DV)*.

Selection of Object. We chose Phormer photo gallery application as our object [66], which has about 6,000 lines of code and over 43,000 downloads. It is an open-source PHP-based application that allows users to store photos, categorize and rate them, and view a slideshow. Since we had allocated limited time for each session, we had to choose an application that is simple, and yet exhibits realistic motifs in its behaviour - these criteria are met by Phromer.

Selection of Subjects. We recruited 14 participants for our experiment. They were all graduate students in computer science and engineering, and many of them had professional software development experience. The participants consisted of 2 females and 12 males, aged between 23 and 35. Knowledge of programming and familiarity with web development (and particularly JavaScript) were our only requirements for picking the participants. Overall, our participants had 1–10 years of web development and 1–18 years of software development experience, respectively.

Experimental Design. Our experiment had a "between-subject" design. To avoid the carryover effect, we divided our participants into two groups. The *experimental* group were given access to SABALAN for performing the tasks, while the *control* group used Google Chrome's Developer Tools (i.e., DevTools) for completing the session. All our participants were familiar with DevTools according to their answers to the pre-questionnaire form and chose to use it during the experiment. No member of the experimental group were familiar with SABALAN prior to the study session. To avoid bias in favour of one of the groups in terms of their proficiency levels, we collected historical data about our participants prior to scheduling the sessions. We assigned each participant a proficiency score, based on a combination of metrics, including their experience with software development, knowledge of JavaScript, and how they perceived their own expertise. We balanced the proficiency levels in both experimental and control groups.

Experiment Tasks. We designed five comprehension tasks, outlined in Table 2, based on common program comprehension activities proposed by Pacione et al. [63]. As the name suggests, these activities represent fine-grained activities that developers need to perform for understanding software, regardless of the language and the platform used. Table 2 shows how each of our tasks covers one or more activities - all activities are covered in our design. Moreover, each task also included a mini questionnaire, which asked about how participants perceived the difficulty of the task, the required time, and the required expertise level for accomplishing the task. We have made all the tasks and datasets publicly available [71].

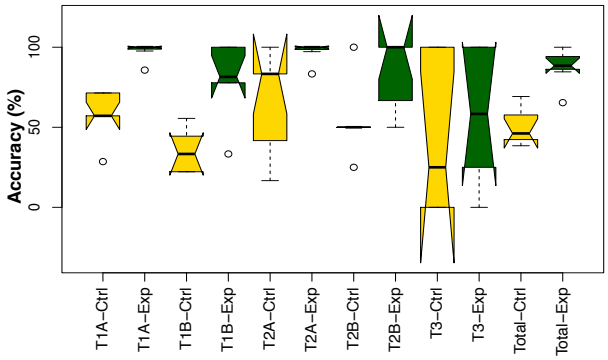

Figure 8: Notched box plots of accuracy results. Green plots display experimental (SABALAN) group, and gold plots display the control group. Higher values are better.

7.2.2 Experimental Procedure. The procedure of the experimental sessions consisted of three main phases.

- **Pre-study.** We required all participant to fill a pre-questionnaire form to gather some demographic data about them. Further, we used the data regarding their experience, programming habits, and self-perceived expertise level, to assign participants' expertise scores. The score allowed us to fairly balance the expertise levels in both experimental and control groups.
- **Training.** At this step, the experimental group were given a tutorial on SABALAN, which they were encountering for the first time. Then both groups were given some time to familiarize themselves with the setting of the experiment. We then started the tasks when the participants were ready.
- **Tasks.** During this phase, the participants completed the five comprehension tasks summarized in Table 2. Based on our design, we wanted to measure both *duration* and *accuracy* of completing the tasks. To measure time, we prepared each task on a separate sheet of paper. We started a timer when we handed a task sheet to a participant, and asked her to return it to us (with the answer) as soon as she had completed the task, which is when we stopped the timer. This allowed us to record the time they spent on each task. We evaluated the accuracy of each task later, based on rubrics we had prepared prior to conducting the experiment.
  Moreover, we wanted to gather some data regarding how the participants perceived the tasks. Thus, we provided them with a set of meta tasks, that questioned them about the perceived difficulty, time-consumption, and required expertise level for each task. Finally, the participants filled a post-questionnaire form regarding their experience in the study.

7.2.3 Results. We first ran the Shapiro-Wilk test on all collected data sets, to determine if they were normally distributed. For normally distributed data of accuracy we used *two-sample t-tests*. The duration data did not pass the normality test and thus we used the *Mann-Whitney U test* for it.

For the accuracy, the results of running the tests showed a significant difference, with a high confidence, for the experimental

group using Sabalan (Mean=87.8%, STDDev=11.6%), compared to the control group using Chrome DevTools (Mean=50.5%, STDDev=11.6%); ($p-value = 6.2e-05$). The accuracy results are shown in Figure 8. *Overall, using Sabalan increased developers' accuracy in performing comprehension tasks by an average of 54%, over other tools.* (RQ2.1). We further analyzed the impact of using Sabalan in accuracy of individual tasks. The results of running the statistical tests showed significant difference in favour of Sabalan for all tasks, expect T3. The accuracy of results of T1A through T2B were significantly higher using Sabalan. The results for task T3, although not statistically significant, were on average 23% more accurate when participants used Sabalan.

For the times, the collected task completion duration data were comparable for participants of both experimental and control groups. Running the tests did not reveal any statistically significant difference in task duration between the two groups (RQ2.2). Finally, we analyzed the collected data from the questionnaire form participants filled regarding each task. They perceived the difficulty of tasks from 2.15 to 2.54, based on a 5-point Likert scale, which shows an average level of difficulty for all tasks. We compared the difficulty of each task as perceived by participants with the results of duration and accuracy of the same task. We found no correlation between the perceived difficulty of a task by participants and how they perform the task (using the Pearson correlation coefficient).

7.3 Discussion

The results of the experiment revealed that Sabalan improves developers' performance in comprehension by significantly increasing their accuracy by 54% (RQ2.1). The results however did not show a significant difference for duration (RQ2.2).

Domain Knowledge and Use-cases. One of the first steps towards program understanding is general understanding of its domain and overall dynamic behaviour by identifying the components that provide a solution to the domain. Our results show that Sabalan significantly increases the accuracy of such tasks (T1). This task consisted of two main parts, understanding the overall behaviour and use-cases of the experimental object (T1.A), and deciding on their importance (T2.B). Using Sabalan significantly improved the accuracy of these two tasks by 49% and 78%, respectively. The results show that using Sabalan not only provides a more accurate overview of an application's behaviour compared to ad-hoc approaches, but also helps developers obtain a better understanding of the importance and usage of the main system components and their interactions (RQ2.3).

Feature Location. Feature location is one of the main tasks performed during program comprehension, and has many applications, such as reuse and testing. Our results show that using Sabalan significantly improved the accuracy of feature location (RQ2.1, RQ2.3). The experimental group were able to find components involved in the implementation of a feature and infer their relations 42% more accurately than the control group (T2.A). They were also 42% more accurate in estimating the quality of the implementation of the said feature (T2.B). Investigating the answers revealed that the control group missed many connections in the code that lead to discovery of different parts of the implementation and thus failed to create a complete and accurate model of the involved code. Due to their incomplete understanding of the feature, the control group was not able to estimate and measure the *quality* of the respective part of application as well. The experimental group, however, could assess the quality based on the more accurate model of the behaviour that extracted the feature as a behavioural motif (RQ2.3).

Software Change and Root-Cause Detection. The last task (T3) involved understanding the system in order to make a change, by finding the root cause of a particular observed behaviour. The experimental group were able to perform the task 23% more accurately with Sabalan, although the results were not statistically sigifnificant (RQ2.1). Using Sabalan, they were able to focus on a much smaller part of the code that was relevant to the feature that needed change. However, because we do not have debugger support within Sabalan, using common debugging techniques such as setting breakpoints and watching variables in such tasks required the participants to frequently switch between the visualization and the application. We plan to integrate our prototype with a debugger such as Chrome DevTools in the future.

Accuracy over Speed. The results did not show any significant difference for task completion duration in favour of Sabalan. We believe this is not a significant issue due to three reasons. *First*, accuracy of performing a task is more important than its speed [2]. The significant improvement of task completion accuracy with Sabalan (54%), and the test's high confidence in the result, emphasize the challenges of comprehending traces, as well as the usefulness of Sabalan in improving developers' performance for completing said tasks. Investigating the answers further, we found that many participants in the control group had finished the tasks, assuming they had the right answers. While in fact, they were not even aware that they are missing crucial parts of the answer, which resulted in them having lower accuracy than Sabalan users. *Next*, we believe that the unfamiliarity of our participants with Sabalan might have caused them to spend more time trying to use it. This theory is strengthened when we analyze individual tasks results. We observed that the experimental group had the worst speed ratio compared to the control group for the first task (T1.A), after which they quickly improve and surpass the control group in later tasks. *Finally*, dividing the locus of attention may have also played a role in the results. While the control group only focused on the browser, the Sabalan group had to switch back and forth between the application and the tool. We believe this can be solved by extending the tool or integrating it into an existing programming environment.

Participants' Perception of Tasks vs. Performance Reality. There were no correlations between the difficulty of a task as perceived by participants, and their measured performance scores. All participants deemed all tasks to be of moderate difficulty. However, the control group scored significantly lower accuracy marks for all tasks. This shows that their interpretation of the task requirements did not match the reality of the task they had just performed. The results confirm the challenging nature of trace comprehension.

Performance Overhead. We used the experimental object of our user study, Phormer, to obtain data regarding the additional overhead of our approach, in 10 one-minute interaction sessions. We measured three sources of potential performance overhead into account. The overhead caused by *instrumentation* phase, the imposed overhead on *execution* of instrumented code and data collection,

and the overhead of *analysis* of traces and motif extraction, which were respectively measured as 1.2, 0.1, and 2.1 seconds on average. This is negligible for all practical purposes, and is barely noticeable during the interaction with the application. Thus, the performance overhead of Sabalan was entirely acceptable for this application.

7.4 Threats to Validity

The **external threats** of conducting an experiment such as ours, typically arise from representativeness of tasks, participants, and object selected for the experiment. We mitigated the threat of task selection by designing our tasks so that they covered all common comprehension activities in Pacione et. al. [63], which are representative of routine comprehension tasks. A valid concern is regarding the representativeness of the participants of the developer population, since we recruited students. We tried to address this concern by recruiting only graduate students who had prior experience with JavaScript, many of whom had experience working in industry. To address the threat of representativeness of the experimental object, we chose an open source JavaScript application Phormer, [66] with about 6,000 lines of code and over 43,000 downloads at the time of conducting the study. An **internal threat** that concerns our method is the bias towards assigning participants into control and experimental groups, or the population-selection problem, which we addressed by balancing the expertise levels between the two groups. Other threats can arise due to the possible bias of the examiner (us) regarding the measurement of both duration and accuracy of task completion. We mitigated the time measurement threat by designing a method for time measurement that both the participant and the examiner agreed upon, namely physically exchanging the task sheet between the participant and the examiner. We mitigated the bias towards measuring accuracy by creating a rubric prior to conducting the experiments, and abiding by the rubric for marking the tasks in order to address this threat. The final threat we address is the tool used in the experiment. We chose Google Chrome's Developer Tools, which is very popular for client-side web development, and all our participants were previously familiar with it (based on the pre-questionaire they filled out).

8 RELATED WORK

Trace Analysis and Visualization. Several papers assist program comprehension through dynamic analysis and visualization. Their proposed techniques allow users to explore large traces [69], or perform reduction, compaction and pruning techniques on traces [12, 32–34, 68]. A popular trend is using standard visual protocols, such as UML diagrams [13, 22, 76]. Others propose more customized visualization techniques through synchronized views [17], provide program's landscape focusing on communications [29], allow user interactions with the visualization [65], visualize similarities in traces [18], or present many other techniques for representing the traces [10, 21, 28, 39–41, 48, 69, 72, 77]. Extravis [17] is the first such technique that was quantitatively measured in a user study [19], followed by others [11, 61]. Another group of methods capture and analyze low-level information in traces using techniques such as extracting behavioural units described in usecase scenarios [78], profiling [43], dividing the trace into segments [67], identifying feature-level phases by defining an optimization problem [7], or similar methods [15, 23, 64]. Heuzeroth et al. [35, 36] propose to find patterns in execution. Others aim at providing higher-level representations of trace [2, 3, 80]. However, unlike our approach, these approaches do not infer a hierarchical behavioural model, which reveals the key points of behaviour, while still preserving the details, and permitting users to navigate them on demand. Further, our approach infers abstract motifs, which reduces the dependency of our analysis on details, without losing the data.

Feature Location, Capture & Replay. Many papers have focused on feature location [55, 57, 81]. Record and replay tools aid understanding and debugging [6, 14, 28, 56, 58, 59, 79] by providing a deterministic replay of UI events without capturing their consequences. Tracing techniques [5, 37, 62] collect traces of JavaScript selectively. Some papers focus on visualization for helping program understanding [5, 62, 81]. However, these methods are committed to preserving the exact sequence of events and replaying them. Their analyses do not permit abstraction of implementation details and inferring higher-level motifs. They are not concerned with creating a behavioural model and providing a hierarchical overview of execution for assisting comprehension, unlike our work.

Specification Mining. Assisting comprehension by mining software specifications from traces has been well explored [11]. Many methods have assessed [49] or improved the performance of miners, by pruning and clustering traces [50], supporting equivalent states [53, 73], or finding inconsistencies in resource utilizations [61]. Others compare different model inference techniques [44], synergize or combine them [47, 52, 54], or facilitate declaration of algorithms [8, 9]. Other work uses Markov models and analyzes them using model checking [30]. Unlike our approach, these techniques do not provide a hierarchical model of abstract recurring motifs of program execution. Specification mining has been improved with novel use of object hierarchies [51]. However, this technique only supports existing hierarchies between Java classes and packages, and not application-specific motifs, which are not defined or specified prior to analysis. Unlike these approaches, our algorithm allows our analysis to abstract out the low-level details and tolerate small changes, which allows recurring motifs of behaviour to emerge.

9 CONCLUDING REMARKS

In this paper, we proposed a generic technique for inferring a hierarchical model of application-specific motifs from execution traces. Our motifs, inspired by bioinformatics algorithms, are recurring abstract patterns of execution that abstract out alterations and are closer to the higher-level features of a system. We designed a visualization for our technique that allows users to observe and query the motifs for program understanding. Our technique is implemented in a tool called Sabalan, which is publicly available. The results of our user experiment showed that using the systematic analysis of Sabalan enabled participants to perform comprehension tasks 54% more accurately than other state of art tools.

10 ACKNOWLEDGEMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council (NSERC) and a research gift from Intel Corporation. We are grateful to all of the participants in our experiment.

REFERENCES

[1] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2015. Hybrid DOM-Sensitive Change Impact Analysis for JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. LIPIcs, 321–345.

[2] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding Asynchronous Interactions in Full-Stack JavaScript. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 1169–1180.

[3] Saba Alimadadi, Sheldon Sequeira, Ali Mesbah, and Karthik Pattabiraman. 2016. Understanding JavaScript Event-Based Interactions with Clematis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2016), 17 pages. http://salt.ece.ubc.ca/publications/docs/tosem16.pdf

[4] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3 (1990), 403 – 410.

[5] Domenico Amalfitano, AnnaRita Fasolino, Armando Polcaro, and Porfirio Tramontana. 2014. The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering* 10, 1 (2014), 41–57.

[6] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *Proceedings of the International Conference on Dependable Systems & Networks (DSN)*. IEEE Computer Society, 403–410.

[7] Omar Benomar, Houari Sahraoui, and Pierre Poulin. 2014. Detecting program execution phases using heuristic search. In *International Symposium on Search Based Software Engineering*. Springer, 16–30.

[8] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2013. Unifying FSM-inference algorithms through declarative specification. In *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 252–261.

[9] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering* 41, 4 (2015), 408–428.

[10] Ivan Beschastnikh, Yuriy Brun, Michael D. Ernst, and Arvind Krishnamurthy. 2014. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, 468–479.

[11] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 267–277.

[12] Johannes Bohnet, Martin Koeleman, and Jürgen Döllner. 2009. Visualizing massively pruned execution traces to facilitate trace exploration. In *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 57–64.

[13] Lionel C Briand, Yvan Labiche, and Johanne Leduc. 2006. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering* 32, 9 (2006), 642–663.

[14] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the Symposium on User Interface Software and Technology (UIST)*. ACM, 473–484.

[15] Jonathan E Cook and Zhidian Du. 2005. Discovering thread interactions in a concurrent system. *Journal of Systems and Software* 77, 3 (2005), 285–297.

[16] Thomas A. Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[17] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. 2007. Understanding execution traces using massive sequence and circular bundle views. In *IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 49–58.

[18] Bas Cornelissen and Leon Moonen. 2007. Visualizing similarities in execution traces. In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*. 6–10.

[19] Bas Cornelissen, Andy Zaidman, and Arie van Deursen. 2011. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering* 37, 3 (2011), 341–355.

[20] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

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

[22] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the Execution of Java Programs. In *Software Visualization: International Seminar*, Stephan Diehl (Ed.). Springer Berlin Heidelberg, 151–162.

[23] Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. 2010. Modeling features at runtime. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 138–152.

[24] Patrik D'haeseleer. 2006. What are DNA sequence motifs? *Nature biotechnology* 24, 4 (2006), 423–425.

[25] escodegen 2017. Escodegen. https://github.com/estools/escodegen. (2017).

[26] esprima 2017. Esprima. http://esprima.org/. (2017).

[27] estraverse 2017. Estraverse. https://github.com/estools/estraverse. (2017).

[28] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proceedings of International Conference on Software Engineering (ICSE)*. IEEE, 752–761.

[29] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. 2013. Live trace visualization for comprehending large software landscapes: The ExplorViz approach. In *IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.

[30] Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. 2014. Mining behavior models from user-intensive web applications. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 277–287.

[31] Martin Gollery. 2005. Bioinformatics: Sequence and Genome Analysis, 2nd ed. *Clinical Chemistry* 51, 11 (2005), 2219–2219.

[32] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. 2006. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 181–190.

[33] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. 2004. A survey of trace exploration tools and techniques. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 42–55.

[34] Abdelwahab Hamou-Lhadj, Timothy C Lethbridge, and Lianjiang Fu. 2004. Challenges and requirements for an effective trace exploration tool. In *Proceedings of the IEEE International Workshop on Program Comprehension*. IEEE, 70–78.

[35] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. 2003. Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 94–103.

[36] Dirk Heuzeroth, Thomas Holl, and Welf Löwe. 2001. *Combining static and dynamic analyses to detect interaction patterns*. Univ., Fak. für Informatik, Bibliothek.

[37] Joshua Hibschman and Haoqi Zhang. 2015. Unravel: Rapid Web Application Reverse Engineering via Interaction Recording, Source Tracing, and Library Detection. In *Proceedings of ACM User Interface Software and Technology Symposium (UIST)*. ACM, 270–279.

[38] hoxy 2017. Hoxy Proxy. greim.github.io/hoxy/. (2017).

[39] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. 2014. State of the art of performance visualization. *EuroVis* (2014).

[40] S Jayaraman, B Jayaraman, and D Lessa. 2016. Compact visualization of Java program execution. *Software: Practice and Experience* (2016).

[41] Benjamin Karran, Jonas Trumper, and Jurgen Dollner. 2013. Synctrace: Visual thread-interplay analysis. In *IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–10.

[42] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (2006), 971–987.

[43] Johannes Koskinen, Markus Kettunen, and Tarja Systa. 2006. Profile-based approach to support comprehension of software behavior. In *IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 212–224.

[44] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 178–189.

[45] Alyson La. 2015. Language Trends on GitHub. https://github.com/blog/2047-language-trends-on-github. (2015).

[46] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 492–501.

[47] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (t). In *International Conference on Automated Software Engineering (ASE)*. IEEE, 115–125.

[48] Shen Lin, François Taïani, Thomas C Ormerod, and Linden J Ball. 2010. Towards anomaly comprehension: using structural compression to navigate profiling call-trees. In *Proceedings of the 5th international symposium on Software visualization*. ACM, 103–112.

[49] David Lo and Siau-Cheng Khoo. 2006. QUARK: Empirical assessment of automaton-based specification miners. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 51–60.

[50] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 265–275.

[51] David Lo and Shahar Maoz. 2009. Mining hierarchical scenario-based specifications. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 359–370.

[52] David Lo and Shahar Maoz. 2012. Scenario-based and value-based specification mining: better together. *Automated Software Engineering* 19, 4 (2012), 423–458.
[53] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 345–354.
[54] David Lo, Leonardo Mariani, and Mauro Santoro. 2012. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software* 85, 9 (2012), 2063–2076.
[55] James Lo, Eric Wohlstadter, and Ali Mesbah. 2013. Imagen: Runtime Migration of Browser Sessions for JavaScript Web Applications. In *Proceedings of the International World Wide Web Conference (WWW)*. ACM, 815–825.
[56] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static Analysis of Event-Driven Node.js JavaScript Applications. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 505–519.
[57] Josip Maras, Maja Stula, and Jan Carlson. 2013. Generating Feature Usage Scenarios in Client-side Web Applications. In *Proceeding of the International Conference on Web Engineering (ICWE)*. Springer, 186–200.
[58] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, 159–174.
[59] Paula Montoto, Alberto Pan, Juan Raposo, Fernando Bellas, and Javier López. 2009. Automating navigation sequences in AJAX websites. In *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer, 166–180.
[60] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-level Models. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT)*. ACM, 18–28.
[61] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. 2014. Behavioral resource-aware model inference. In *Proceedings of the ACM/IEEE international conference on Automated software engineering*. ACM, 19–30.
[62] Stephen Oney and Brad Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE Computer Society, 105–108.
[63] Michael J Pacione, Marc Roper, and Murray Wood. 2004. A novel software visualisation model to support software comprehension. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, IEEE, 70–79.
[64] Vijay Krishna Palepu and James A. Jones. 2013. Visualizing constituent behaviors within executions. In *IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 1–4.
[65] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. 2010. Immediacy through interactivity: online analysis of run-time behavior. In *Working Conference on Reverse Engineering (WCRE)*. IEEE, 77–86.
[66] Phormer 2017. Phormer PHP Photo Gallery. http://p.horm.org/er/. (2017).
[67] Heidar Pirzadeh and Abdelwahab Hamou-Lhadj. 2011. A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension. In *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 221–230.
[68] Steven P. Reiss and Manos Renieris. 2001. Encoding program executions. In *proceedings of the International Conference on Software Engineering*. IEEE Computer Society, 221–230.
[69] Manos Renieris and Steven P. Reiss. 1999. ALMOST: exploring program traces. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management*. ACM, 70–77.
[70] M. P. Robillard, W. Coelho, and G. C. Murphy. 2004. How effective developers investigate source code: an exploratory study. *IEEE Transactions on Software Engineering* 30, 12 (2004), 889–903.
[71] Sabalan 2017. Sabalan. https://github.com/saltlab/sabalan. (2017).
[72] Raja R Sambasivan, Ilari Shafer, Michelle L Mazurek, and Gregory R Ganger. 2013. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE transactions on visualization and computer graphics* 19, 12 (2013), 2466–2475.
[73] Matthias Schur, Andreas Roth, and Andreas Zeller. 2013. Mining behavior models from enterprise web applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 422–432.
[74] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195 – 197.
[75] Stack Overflow. 2017. Developer Survey. http://stackoverflow.com/research/developer-survey-2017. (2017).
[76] Tarja Systä, Kai Koskimies, and Hausi Müller. 2001. Shimba–an environment for reverse engineering Java software systems. *Software: Practice and Experience* 31, 4 (2001), 371–394.
[77] Jonas Trümper, Johannes Bohnet, and Jürgen Döllner. 2010. Understanding complex multithreaded software systems by using trace visualization. In *Proceedings of the international symposium on Software visualization*. ACM, 133–142.
[78] Yui Watanabe, Takashi Ishio, and Katsuro Inoue. 2008. Feature-level phase detection for execution trace using object cache. In *Proceedings of the international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 8–14.
[79] Shiyi Wei and Barbara G Ryder. 2014. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*. Springer, 1–26.
[80] Andy Zaidman and Serge Demeyer. 2004. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 329–338.
[81] Andy Zaidman, Nick Matthijssen, Margaret-Anne Storey, and Arie van Deursen. 2013. Understanding Ajax applications by connecting client and server-side execution traces. *Empirical Software Engineering* 18, 2 (2013), 181–218.