Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events

Markus Ermuth
Department of Computer Science
TU Darmstadt, Germany
markus.ermuth@gmail.com

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

ABSTRACT

Automated testing is an important part of validating the behavior of software with complex graphical user interfaces, such as web, mobile, and desktop applications. Despite recent advances in UI-level test generation, existing approaches often fail to create complex sequences of events that represent realistic user interactions. As a result, these approaches cannot reach particular parts of the application under test, which then remain untested. This paper presents a UI-level test generation approach that exploits execution traces of human users to automatically create complex sequences of events that go beyond the recorded traces. The key idea is to infer so-called macro events, i.e., sequences of low-level UI events that correspond to a single logical step of interaction, such as choosing an item of a drop-down menu or filling and submitting a form. The approach builds upon and adapts well-known data mining techniques, in particular frequent subsequence mining and inference of finite state machines. We implement the approach for client-side web applications and apply it to four real-world applications. Our results show that macro-based test generation reaches more pages, exercises more usage scenarios, and covers more code within a fixed testing budget than a purely random test generator.

CCS Concepts

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

Keywords

GUI testing, test generation, JavaScript, web applications

1. INTRODUCTION

Many programs, such as client-side web applications, mobile applications, and classical desktop applications, interact with users through a graphical user interface (GUI). It is important to test such programs at the UI-level by triggering sequences of UI events, such as clicking, moving the mouse, and filling text into a form. However, the complexity of many GUI applications makes manual UI-level testing difficult. For example, a complex client-side web application may consist of dozens of pages that each provide hundreds of events that a tester may trigger. Because exploring such programs manually is difficult, automated test generation approaches have been proposed [26, 24, 27, 11, 8, 42, 17, 35]. The basic idea is to generate sequences of UI events that achieve high coverage or that trigger a particular kind of problem. Existing approaches include black-box approaches, such as the popular Monkey runner for Android [1], which triggers random UI events, and white-box approaches, which, e.g., symbolically analyze the programs code to find events worth triggering.

Despite recent advances in UI-level test generation, two important challenges remain. First, deeply exploring a program often requires *complex sequences of events*. For example, consider a program that uses a drop-down menu to connect pages to each other. To reach another page, a test generator must move the mouse into the menu, wait until the menu appears, and then click on one of the menu items, without interleaving other events that would hide the menu again. Existing black-box approaches are unlikely to create such complex sequences because they are unaware of the semantics of the individual UI events. One approach would be to enhance black-box approaches with domain knowledge about the most common complex sequences of events. However, the large number of such sequences and their different implementations makes this approach difficult in practice. Existing white-box approaches are, in principle, able to identify complex sequences of events that lead to not yet covered behavior, but do not scale well to complex programs.

Second, effective testing requires both *realistic sequences of events*, to explore the most common paths that users will take, and unusual sequences of events, to uncover corner case errors. For example, consider a program that asks the user to fill and submit a form. Most users will fill data into each part of the form and then click the submit button. Since most existing test generators are oblivious of how realistic a sequence of events is, they do not recognize this common way of using such a form, and instead spend significant effort on exploring behavior that may not be relevant in practice, such as filling in some data without ever submitting the form.

We identify an important reason why existing test generation approaches do not fully address these challenges: The granularity of events as seen by the test generator does not

ISSTA'16, July 18–20, 2016, Saarbrücken, Germany
ACM. 978-1-4503-4390-9/16/07...$15.00
http://dx.doi.org/10.1145/2931037.2931053

[1] http://developer.android.com/tools/help/monkey.html

match the logical steps perceived by a user. For example, a user considers "go to another page via a drop-down menu" or "fill and submit a form" as one logical event, whereas a test generator treats them as sequences of various UI events. Test generators that are unaware of such logical steps spend lots of effort in generating sequences of events that a user might never trigger and that may not reach particular states of the program.

This paper presents a UI-level test generation approach that exploits execution traces of human users to automatically create complex sequences of events that represent logical steps as perceived by a user. We call such sequences of UI events *macro events*. The key components of the presented approach infer macro events from given usage traces and apply the inferred macro events during test generation to automatically create tests that trigger complex, realistic sequences of events. Because the approach can be combined with other test generation approaches, such as random testing or guided testing, it preserves the benefits of these approaches while addressing the challenge of creating complex, realistic sequences of events.

A major challenge is to infer and represent macro events in a way that abstracts from the recorded usage trace. This abstraction helps the approach to summarize multiple slightly different usages into a single macro event and to apply a macro event beyond the situation from which it is inferred. We address this challenge by combining and extending two data mining techniques, frequent subsequence mining and inference of finite state machines (FSMs). For example, these techniques may infer a macro event that represents "go to another page via a drop-down menu" from usage traces that use the menu to reach some pages and apply the macro event to reach other pages.

We implement the approach into an automated test generator for client-side web applications and evaluate it with four widely used programs. Our results show that the approach can effectively infer macro events from usage traces and that macro-based test generation improves upon random test generation. In particular, we find that the approach increases the number of pages visited within a fixed testing budget by 69.6%, on average, and that it increases the number of covered branches for three of the four programs. Furthermore, the approach is able to cover several usage scenarios that random testing misses.

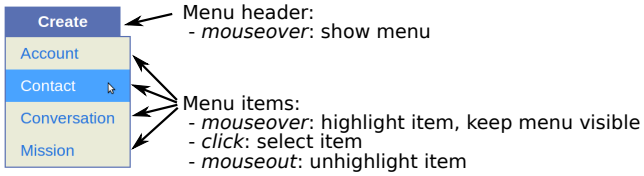
In summary, this paper contributes the following:

- We introduce macro events, which summarize sequences of UI events into logical steps that users commonly perform.
- We present algorithms for inferring macro events from usage traces and for applying them to generate UI-level tests that trigger complex, realistic sequences of events.
- We implement the idea into a practical tool and provide empirical evidence that it improves the effectiveness of random test generation for widely used programs.

2. EXAMPLE AND OVERVIEW

This section motivates our approach with an example and outlines its key components. Figure 1a shows a drop-down menu that enables a user to navigate to different functionalities of the program. The example is a simplified version

(a) Screenshot:



(b) Execution traces:

Trace 1		Trace 2		Trace 3	
Event	Element	Event	Element	Event	Element
mouseover	header	mouseover	header	mouseover	header
mouseover	item	mouseover	item	mouseover	item
click	item	mouseout	item	click	item
mouseout	item	mouseover	item	mouseout	item
		click	item		
		mouseover	item		

(c) Inferred macro event:

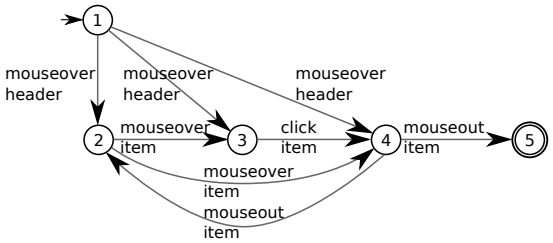


Figure 1: Motivating example.

of a menu in the Zurmo web application. Correctly using the menu is crucial for effectively using the program. Even though choosing an item from the menu can be seen as one logical step, it consists of multiple independent UI events: Revealing the menu corresponds to a *mouseover* event on the menu header. Moving the mouse through the menu corresponds to *mouseover* and *mouseout* events on the menu item list, which highlight the corresponding item and keep the menu visible. Selecting a menu item corresponds to a *click* event on the item.

Usage scenarios that require complex sequences of events, such as the example, pose a challenge for automated test generators. For example, consider a generator that creates sequences of events by randomly selecting among the set of available events. Since the menu in Figure 1a is part of a complex web site with hundreds of available events, the probability to trigger a sequence of events that opens the menu, keeps it visible by moving the mouse over it, and selects a menu item is very small. As a result, such random testing is unlikely to reach all pages that users may access through the menu and therefore cannot thoroughly test parts of the program.

The approach presented in this paper addresses the challenge of generating complex and realistic sequences of events in three steps. First, the approach gathers execution traces from the interactions of human users with a program. Such traces can be gathered, e.g., during manual pre-deployment testing or via lightweight, distributed sampling of user interactions with a deployed program. Second, the approach identifies recurring patterns in the usage traces and summarizes them into an FSM-based description. Each such pattern, called a macro event, represents one logical step

of using the program. Third, the approach automatically generates tests, i.e., sequences of UI events, through a combination of random testing and applying the inferred macro events. Section 3 presents these steps in detail. The following illustrates them with the motivating example.

Recording usage traces. To illustrate the first step consider three usages of the menu. In the first and the last usage, the user selects the first item. In the second usage, the user moves the mouse over the first item and selects the second item. Figure 1b shows the execution traces for these usages. The traces summarize the events triggered by the user and record the event type, e.g., *mouseover*, and a description of the DOM element on which the event is triggered, e.g., *header*. Note that in practice, execution traces typically comprise thousands of events. Section 3.1 describes the details of recording usage traces.

Inferring macro events. The key challenges for the second step of the approach are twofold. First, to be applicable to real-world programs, the approach must efficiently deal with large execution traces. Second, to infer macro events that are applicable beyond simply replaying exactly the sequences of events observed in a usage trace, the approach must abstract events and the order in which events occur. We address these challenges by using and extending two existing data mining techniques. At first, frequent subsequence mining identifies recurring sequences of events that are candidates for describing a single logical step of interacting with the program. For the example, the approach identifies five such subsequences, which correspond to Trace 1 and Trace 2 and three subsequences of them. For realistic traces, the approach typically identifies thousands of subtraces that each contain a subset of all trace events. Then, the approach clusters related subsequences and applies a variant of the k-tails algorithm [6] to summarize a set of subsequences into an FSM that represents a macro event. Figure 1c shows the macro event that the approach infers for the running example. Each transition represents a UI event, and a sequence of UI events accepted by the FSM corresponds to the logical step described by the macro event. Section 3.2 describes inferring macro events in detail.

Generating sequences of UI events. Based on the inferred macro events, the final step of the approach creates tests by generating sequences of UI events. The test generation algorithm probabilistically chooses between applying a macro event (if there is an applicable macro event) and randomly picking the next event to trigger. For the example, the test generator may try to apply the macro event whenever it reaches a page with a matching drop-down menu. Because the macro events generalizes from the observed traces, applying the macro may skip an arbitrary number of menu items before clicking on an item. We combine macro event-based testing with random testing because not all events of a program are part of a macro event. Section 3.3 describes the test generation part of the approach in detail.

3. APPROACH

This section explains the three steps outlined in the previous section in detail. Users interact with web applications through events:

Definition 1. A *micro event* e, or short event, is a tuple (t, d), where

- t is the event's type, and
- d is the kind of DOM element that the event is triggered on (also called event target).

For example, a click on a button is represented as a micro event $(click, button)$. The goal of our approach is to summarize logical steps of a user's interaction with a program into macro events:

Definition 2. A *macro event*, or short macro, describes a sequential event pattern consisting of two or more micro events. The pattern is represented by a non-deterministic FSM $(\Sigma, \mathcal{A}, s_0, \delta, \rho, \mathcal{F})$, where:

- Σ is the set of all micro events used within the pattern,
- $\mathcal{A}$ is the set of states,
- $s_0 \in \mathcal{A}$ is the initial state,
- $\delta : \mathcal{A} \times \Sigma \to \mathcal{P}(\mathcal{A})$ is the transition function that returns the set of destination states that can be reached when triggering a particular micro event in a particular state,
- $\rho : \Sigma \to \{same, ancestor, descendant, family, none\}$ assigns to each micro event its structural relation to the preceding micro event, and
- $\mathcal{F} \subseteq \mathcal{A}$ is the set of final states, characterized by the lack of outgoing transitions.

For example, Figure 1c shows a macro event that summarizes the sequences of micro events that a user may trigger to select an item from a drop-down menu.

3.1 Recording User Actions

While a user interacts with the program under test, the user logging component of our approach records all UI events that the user triggers. Since the ultimate goal is to exercise code triggered by user events, the approach ignores events that do not trigger any JavaScript code.

The approach stores the recorded events into an event trace, which is a sequence of trace entries.

Definition 3. A *trace entry* t is a tuple (e, p, s), where:

- e is the recorded event,
- p is an abstraction of the path in the DOM tree that contains the event target, and
- s is the state of the program when the recorded event is triggered.

Recording the path p is important to identify events that belong to the same UI component, e.g., events that belong to different parts of a menu. We use the XPath of a DOM element as the abstraction p because it captures the path within the DOM tree that contains the DOM element. The XPath is a unique reference for the DOM element, as long as the application stays on one page and does not change its DOM tree.

We record the state s of the program because the approach limits the scope of macro events to sequences of events that happen on a single page. To represent the program's state, we use the URL and the title of the current page. Richer representations of the state [42] can be plugged into our approach.

3.2 Inference of Macro Events

Based on the recorded event traces, the second step of the approach is to infer macro events. This step builds on two existing machine learning approaches, frequent subsequence mining and finite-state machine inference, and has four substeps. First, we pre-process the event traces by splitting them into per-page sequences (Section 3.2.1). Second, we use frequent subsequence mining to identify recurring patterns within the traces (Section 3.2.2). Third, we cluster similar subsequences that belong to the same macro event. Finally, we infer a FSM from each cluster of subsequences. Each such FSM represents one macro event (Section 3.2.4).

3.2.1 Pre-Processing of Event Traces

We limit the scope of the macro events to a single page. The motivation for this pre-processing is two-fold. First, page transitions are a natural border for macro events. Second, cutting the recorded traces into smaller sequences of events reduces the computational effort of the later steps of inferring macro events. To split the trace, our approach compares the URL of each trace entry with the directly preceding entry. If the URLs differ, the approach splits the trace in-between the two entries. To remove sequences that are too short to infer recurring macro events, we drop all sequences with a length below a threshold $length_{min}$ (= 2 for all experiments).

3.2.2 Mining Frequent Subsequences

First, we use frequent subsequence mining to identify recurring patterns within the recorded event traces. By searching for subsequences, the approach gains robustness against noisy events that interleave with the actual pattern. For example, suppose a trace that contains the following sequence of events ten times:

$$(keydown, input)$$
$$(keypress, input)$$
$$(keyup, input)$$

One occurrence of this sequence is interleaved by the following event:

$$(mouseover, h2)$$

The approach identifies this event as noise and infers the remaining sequence of events without the noisy event as a pattern.

Background: Closed sequential pattern mining. Given a set of sequences of items, closed sequential pattern mining (CloSpan) [45] extracts frequent patterns, also known as frequent subsequences. A subsequence is considered *frequent* if the number of sequences that contain the subsequence (i.e., the support) is beyond a particular threshold. CloSpan differs from other subsequence mining approaches such as PrefixSpan [30], because it searches for *closed* frequent subsequences. A frequent subsequence is considered closed if there is no other frequent subsequence that contains all items of the subsequence in the same order, while both subsequences have the same support.

We build upon and extend the CloSpan approach to identify frequent subsequences of trace entries. The original algorithm is not directly applicable for three reasons. First, it searches for subsequences of arbitrary length, which limits its scalability. To enable our approach to scale to event traces of real-world programs, we adapt the algorithm to search subsequences within a configurable minimum and maximum length. This change enables us to control the computational cost of the mining process and to exclude patterns that are too short to represent meaningful macro events. Second, the original algorithm considers items as black boxes. In contrast, our approach reasons about relations between trace entries based on the structural relations between their corresponding DOM elements. Considering such relations is crucial to identify macro events that contain semantically related micro events. We adapt the CloSpan algorithm to consider relations during the mining process. Third, the original algorithm reasons not only about sequences of items but also about sequences of itemsets. Since the order of events is critical for generating UI-level tests, we do not use itemsets in our approach and omit all itemset-related parts of the algorithm.

Algorithm 1 Mining closed frequent subsequences

Input: Set $\mathcal{S}_{rec}$ of recorded sequences
Output: Set $\mathcal{S}$ of closed frequent sequences

1: remove items with support $\leq supp_{min}$ from $\mathcal{S}_{rec}$
2: remove sequences with length $\leq length_{min}$ from $\mathcal{S}_{rec}$
3: $\mathcal{S}^1 \leftarrow$ all frequent 1-item sequences in $\mathcal{S}_{rec}$
4: $\mathcal{S} \leftarrow \mathcal{S}^1$
5: **for each** $\sigma \in \mathcal{S}^1$ **do**
6: $\quad \mathcal{S} \leftarrow$ CloSpan(S);
7: eliminate non-closed sequences from $\mathcal{S}$

8: **function** CloSpan(σ)
9: $\quad$ **if** $\sigma' \in \mathcal{S}$ exists s.tt. $\sigma \sqsubseteq \sigma'$ or $\sigma' \sqsubseteq \sigma$ **then**
10: $\quad\quad$ **return**
11: $\quad$ **if** $|\sigma| \geq length_{min}$ **then**
12: $\quad\quad$ $\mathcal{S} \leftarrow \mathcal{S} \cup \{\sigma\}$
13: $\quad$ **if** $|\sigma| == length_{max}$ **then**
14: $\quad\quad$ **return** ;
15: $\quad$ **for each** $\sigma'' \in \mathcal{S}_{rec}$ s.tt. $rel(lastEntry(\sigma), firstEntry(\sigma'')) \neq$ "none" **do**
16: $\quad\quad$ CloSpan($\sigma \diamond \sigma''$);

Algorithm 1 shows the main steps of the subsequence mining, adapted from the original CloSpan algorithm [45]. The underlined parts are our modifications. The algorithm takes the set $\mathcal{S}_{rec}$ of recorded and pre-processed sequences of trace entries as its input and builds a set $\mathcal{S}$ of closed frequent subsequences. At first, the algorithm prunes trace entries that occur less often than a configurable minimum support and sequences that are shorter than the minimum length. The main part of the algorithm identifies all single-item sequences and then iteratively extends them into longer subsequences using the CloSpan function. Finally, the algorithm scans the resulting set of sequences to remove all non-closed sequences, i.e., sequences that are contained in another sequence with the same support.

Lines 8 to 16 show the CloSpan function, which grows the sequences in $\mathcal{S}$ based on a sequence σ. The function checks whether any already found frequent sequence in $\mathcal{S}$ contains the currently grown sequence σ or any sequence is contained in σ. When such a contained sequence is found, the algorithm stops growing sequences based on σ because it already found all extensions of the current sequences. Next (lines 11 to 14), the algorithm adds σ to the set of subsequences. In contrast to the original algorithm, our approach adds a sequence only if its length exceeds $length_{min}$ (= 2 for all ex-

periments). Furthermore, the approach stops the growth of a sequence when its length reaches $length_{max}$ (= 10 for all experiments). These two checks avoid unnecessary growing and processing of sequences that do not match our length requirements, reducing the overall computational effort of the mining process. Section 5.4 shows the effectiveness of these checks in practice.

The final part of the CloSpan function (lines 15 to 16) recursively calls the function to extend all sequences in $\mathcal{S}$ with the new sequence σ. We extend this step of the original algorithm with an additional filter. The filter ensures that the events that will eventually be part of an inferred macro are semantically related to each other. To illustrate the motivation for this filtering, consider a page that contains multiple forms and an inferred macro that describes the order of micro events required for filling a form. If the macro event describes only the event type and the kind of DOM element of each event (Definition 1), then the test generator may switch between the different forms while applying the macro event. As a result, the approach would partly fill multiple forms instead of fully filling and submitting one form. Instead, we want the test generator to focus on one form at a time, which corresponds to the usage that the macro is inferred from. The challenge is how to distinguish between the different forms, or more generally, how to identify elements that are semantically related.

To address this challenge, the approach considers the structural relations between the target DOM elements of events when inferring and applying macros. The approach reasons about the relations of DOM elements by comparing their XPaths and identifies five kinds of relations:

Definition 4. The relation $rel(t_1, t_2)$ of two trace entries $t_1 = (e_1, p_1, s_1)$ and $t_2 = (e_2, p_2, s_2)$ is one of the following:

- *same* if $p_1 = p_2$, i.e., if the recorded events are triggered in the same DOM element
- *ancestor* if the XPath p_1 contains the XPath p_2, i.e., one event is triggered on a DOM element that is an ancestor of the other DOM element
- *descendant* if the XPath p_2 contains the XPath p_2
- *family* if the XPaths p_1 and p_2 share a common ancestor DOM element d_{anc} that is different from *html* and *body*, and if the distance of the elements described by p_1 and p_2 to d_{anc} is smaller than a configurable threshold
- *none* otherwise

Based on these relations, Algorithm 1 ensures that only related trace entries are extracted into a subsequence. Line 15 checks whether the last entry of the currently grown sequence σ and the first entry of a possible suffix σ'' are in a non-trivial relation. Only if such a relation exists, the algorithm concatenates both sequences, $\sigma \diamond \sigma''$, and recursively passes them to the CloSpan function for further extension. This process continues until the algorithm has identified all frequent subsequences, where each entry is related to the next entry and that fit our length requirements.

3.2.3 Clustering

We use a simple prefix clustering to bundle similar frequent subsequences. For this purpose, the approach compares the first event entry of each found sequence and clusters together those sequences that share the same event.

Algorithm 2 Adapted k-tails

Input: Set $\mathcal{S}_{cluster}$ of closed frequent sequences , k
Output: Merged FSM $M = (\Sigma, \mathcal{A}, s_o, \delta, \mathcal{F})$

```
1: for each σ ∈ S_cluster do
2:     add σ to M
3:     merged ← true
4:     while merged do
5:         merged ← false
6:         for each s1, s2 ∈ A do
7:             if s1, s2 are k-equivalent then
8:                 merge s1, s2
9:                 merged ← true
```

That is, we bundle sequences that share the same entry point. These entry points will serve as the initial state of the FSMs that result when the bundled sequences are merged.

3.2.4 Summarizing Subsequences into FSMs

We define an equivalence relation for events to be able to distinguish between them:

Definition 5. Given a trace $[..., t_1, t_2, ..., t_3, t_4, ...]$, where $t_i = (e_i, p_i, s_i)$, the trace entries t_2 and t_4 are equivalent, if and only if

- $e_2 = e_4$ and
- one of the following is true:
  - $rel(t_1, t_2) = rel(t_3, t_4)$,
  - $rel(t_1, t_2)$ is *none*, or
  - $rel(t_3, t_4)$ is *none*.

That is, we consider two trace entries as equivalent when they have the same event type and the relations to their preceding trace element is the same or non-existent.

For summarizing the subsequence clusters, we adapt the k-tails algorithm [6], which infers a FSM from sequences of events. To obtain a FSM, k-tails creates an initial FSM out of the given sequences and then iteratively merges k-equivalent states of the FSM, until no more states can be merged. Two states are considered k-equivalent if the paths of length k, which start from these states, are equal.

We adapt the k-tails algorithm, as described by Beschastnikh et al. [5] to limit the search space for k-equal states as well as to reduce the total number of states. Algorithm 2 shows our variant of k-tails. Again, we highlight those parts that we changed in comparison to the original algorithm.

The algorithm initializes the FSM by adding sequences to it. But instead of first adding all sequences and then merging all equivalent states, our algorithm adds only one sequence at a time to the FSM and merges the existing FSM with the added sequence before adding another sequence. This approach reduces the search space for equal states compared to the original approach, because we reduce the number of states by merging the FSM before adding new sequences.

Next, the algorithm searches for k-equivalent states (defined below) by comparing all states with each other, until it finds two states that are k-equivalent. If the algorithm finds two k-equivalent states, it merges them. The algorithm repeats the searching and merging until there are no more k-equivalent states.

While our approach merges two states, it also merges transitions that are equivalent except for the relation that the

Algorithm 3 TriggerNextEvent

Input: Set $\mathcal{M}$ of macros, set $\mathcal{E}$ of available events
Output: Triggers the next event

```
1: randNumb ← a random integer value between 0 and 1
2: if randNumb < p_macro then
3:     TriggerRandomEvent(E)
4: else
5:     UseMacroEvent(M, E)
6:     if found no applicable macro then
7:         TriggerRandomEvent(E)
```

attached event entries have to the preceding event in their sequence. For this purpose, the approach keeps the less restrictive relation of the compared transitions. That is, the resulting transition satisfies the requirements of both merged transitions. For example, if one transition has the *same* relation and the other transition has the *family* relation, then the approach assigns the *family* relation to the merged transition.

To merge all final states and therefore reduce the total total number of states, we redefine the original k-equivalence relation as follows. The definition builds upon Definition 5 to compare trace entries.

Definition 6. Two states are k-equivalent if at least one of the following conditions holds:

- both states have equivalent k-tails,
- both states are final states, i.e., they do not have any tails, or
- both states have tails with an equal length $<$ k that end with a final state.

3.3 Test Generation

The final step of our approach is to augment random test generation by applying the inferred macro events. Algorithm 3 shows the test generation algorithm. The algorithm has two modes: macro-based exploration and random exploration. During random exploration, the algorithm switches to macro-based exploration with a configurable probability p_{macro} (= 50% for all experiments). Once it is in macro-based exploration mode, the algorithm tries to execute a macro until reaching a final macro state. The functions *TriggerRandomEvent* and *UseMacroEvent* represent these two modes. *TriggerRandomEvent* simply chooses among all currently available events with a uniform probability distribution. Algorithm 4 summarizes the *UseMacroEvent* function.

One challenge is to choose which macro to use in a particular state of the program. We call the macro that is used to trigger subsequent events, the *active macro*. The choice of the active macro depends on the available events, i.e., events that can be triggered in the current state. A naive approach that selects a macro only if all events that are contained within the macro are available would be ineffective for two reasons. First, a single macro may produce different sequences of events that use different subsets of the macro's events. That is, applying the macro may not need all events of the macro. Second, since a web application is capable of modifying its UI on the fly without actually loading another page, the set of available events may change while applying a macro. In particular, the program may add components

Algorithm 4 UseMacroEvent

Input: Set $\mathcal{M}$ of macros, set $\mathcal{E}$ of available events
Output: Triggers events according to a macro

```
1: if no active macro or page changed then
2:     SelectNewMacroEvent(M, E)
3:     M_active ← selected macro
4:     e ← selected event
5:     TriggerEvent(e)
6:     return ;
7: else
8:     SelectMacroEventContinuation(M_active, E)
9:     if continuation found then
10:        e ← selected event
11:        TriggerEvent(e)
12:        return ;
13:    else
14:        M_active ← none
15:        TriggerNextEvent(M, E)
```

to its UI in reaction to a previously triggered event, thereby making new events available.

To avoid rejecting macros unnecessarily, our approach incrementally checks the applicability of a macro, immediately before triggering an event. For this purpose, we split the task of selecting an applicable macro into two subtasks: selecting a new macro when there is no active macro, and choosing the next transition within the active macro. Algorithm 4 shows how the approach chooses between selecting a new macro and continuing an active one. The algorithm first checks if there is an active macro. If there is none, *SelectNewMacroEvent* is called to select a new macro. Otherwise, the algorithm checks if the active macro can be continued by using the *SelectMacroEventContinuation* function. The algorithm may fail to find a micro event that is applicable according to the current macro and available in the current state of the program, e.g., because the program non-deterministically changed its state. In this case, the approach returns to Algorithm 3, which will choose between triggering a random event and selecting a new macro again.

In the *SelectNewMacroEvent* function, our approach checks if there is a macro that is both applicable in the particular state of the program: Each available event is compared with the event that belongs to an initial transition of each macro. If a match is found, the function chooses the matching available event as selected event that should be triggered as well as the according macro as the active macro. Otherwise, the search for a matching event continues. If the search cannot find any matching event, the algorithm concludes that there is no macro that can be used in the current state of the program and returns without selecting an event or an active macro.

Within the *SelectMacroEventContinuation* function, our approach first checks if the active macro has reached a final state. If so, then the function returns without selecting an event. Otherwise, the algorithm takes a transition from the current state of the macro and searches the available events for an event that matches the transition's event. In case of a match, the matching event is returned as selected event. Otherwise, the algorithm returns without a selected event.

For comparing events, our approach uses Definition 5. For comparing the event target's relation, the approach determines the relation of the candidate's event target with the

event target of the previously triggered event. To match, the relation does not have to be an exact match, but the relation specified in the transition serves as the minimum requirement. Therefore, our approach also accepts events with a target DOM element that have a stricter relation to the previously triggered event.

The combination of random and macro-based exploration is effective for two reasons. First, the algorithm falls back to random event triggering when the test generation cannot find any macro that matches the particular state of the program. The fallback is especially useful, when the test generation gets stuck in a particular state of the program. Without the fallback, the test generator would either repeatedly apply the same macros, while the state of the program remains unchanged, or would not use any macro at all. Instead, a random event helps escape from a dead end. Second, random exploration enables the approach to test those parts of the program that are not covered by any inferred macro event. For example, this may include parts of the program not exercised in the recorded trace.

4. IMPLEMENTATION

We implement the approach as a Firefox add-on and a node.js application. The browser add-on records all events triggered by a user into a file. To this end, the browser add-on keeps track of all available events that a user may trigger and attaches to each of them an additional event handler that records the triggered event. The trace files obtained from users are then analyzed by the node.js application to infer macro events. The test generation part of the approach is also implemented in the browser add-on. The browser add-on part of our implementation builds upon an existing test generation framework for web applications[2] that has been used in previous work [35].

5. EVALUATION

We evaluate the approach with four real-world JavaScript-based web applications. The evaluation addresses the following research questions:

- *How effective is macro-based testing compared to random testing?* Compared to random testing, macro-based testing reaches more pages, achieves higher coverage, and covers more usage scenarios of a program. (Section 5.2)
- *Does the inference approach effectively summarize large traces into macros?* The approach reduces large event traces into a small number of recurring macros. (Section 5.3)
- *Are the inferred macros applicable during test generation?* The macro-based test generator successfully applies most of the inferred macros (87%). (Section 5.3)
- *How efficient is the approach?* The approach is efficient enough to analyze large programs: Inferring macros takes between 13 seconds and 85 minutes per program. The macro-based test generator is only 8% slower than a purely random test generator. (Section 5.4)

5.1 Experimental Setup

[2]https://github.com/michaelpradel/WebAppWalker/

Table 1: Overview of the benchmark programs.

Program	Version	Description	LoC
Drupal	7.38.1	CMS	8,371
MODX	3.0.4	CMS	45,620
Moodle	2.9.1	E-Learning	412,596
Zurmo	2.3.5pl	CRM	42,206

Table 1 lists the benchmark programs along with their number of lines of JavaScript code.[3] We locally install these programs and use a system with an Intel Core i7-860 with 8 cores (4 physical and 4 virtual cores) and 2.8 GHz clock speed. The system has 8 GB RAM and runs Ubuntu 15.04.

To gather event traces, we ask a group of users to exercise the benchmark programs, while our approach captures their interactions. All participants are computer science students (one PhD student, two master students, one bachelor student) that are not involved in the project beyond participating in the experiment and that had no prior knowledge about the benchmark programs before the experiment. We instruct each participant to use each program for five minutes in a way that they image the program to be used during productive work. We further instruct participants that repeatedly using any part of the program is acceptable, whereas they do not have to exhaustive explore every detail of the programs. We do not give any additional instructions to influence the participants' exploration of the programs as little as possible. In total, the experiment yields 16 execution traces.

We set $supp_{min}$ to 8, i.e., two times the number of traces for each program. The rationale is that a subsequence should be contained at least twice in each recorded trace to be considered as frequent. We set k for the k-tails algorithm to 2. To evaluate the test generation, we limit the generation process to trigger 500 events. We repeat each test generation experiment ten times because it depends on non-deterministic decisions.

5.2 Effectiveness of Generated Tests

To evaluate the effectiveness of macro-based test generation, we compare the approach to purely random testing. The random test generator repeatedly picks an event from the set of currently available events, which is equivalent to setting $p_{macro} = 1$ in Algorithm 3. For a fair comparison, we implement the random test generator based on the same infrastructure (Section 4) as the macro-based test generator.

We consider three metrics. First, we measure how many pages of the application a test generator visits depending on the number of triggered events. This metric is useful because reaching a page is a pre-requisite to explore its behavior. Second, we measure how many usage scenarios of a program a test generator covers in a fixed testing budget. To this end, we specify a set of typical usage scenarios, as they may be specified for manual testing. Third, we measure the branch coverage of a program's JavaScript code that a test generator achieves while exercising the program.

5.2.1 Visited Pages

Figure 2 shows for each program the number of visited pages depending on the number of triggered events. For each data point, the graphs show the average over ten rep-

[3]Measured with https://github.com/AlDanial/cloc.

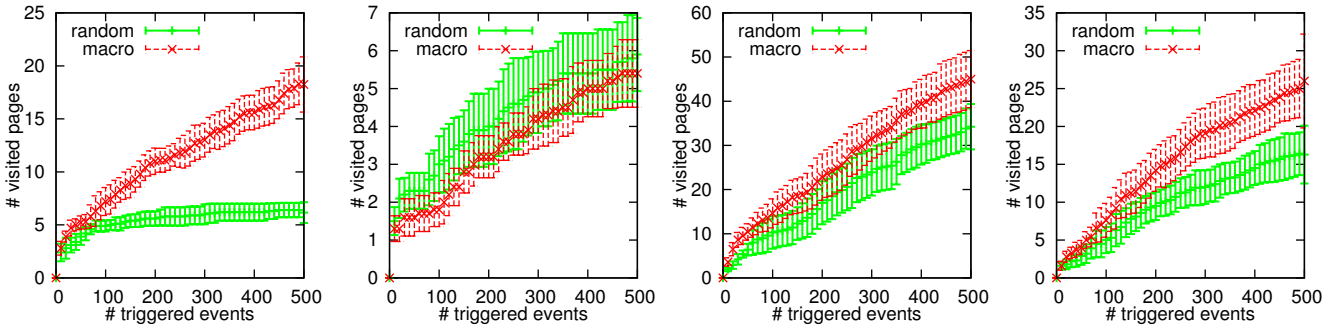


Figure 2: Number of visited pages depending on the number of triggered events. The bars show the 95%-confidence interval and the average value.

etitions of the experiment and the 95%-confidence interval. The figure shows that macro-based testing reaches significantly more pages than random testing for three of the four programs. For example, for Drupal the approach visits only about 18 pages after 500 triggered events, whereas random testing visits about 6 pages. For MODX, macro-based testing visits about 0.5 pages less than random testing, on average. However, the difference is not statistically significant, as indicated by the mostly overlapping confidence intervals.

5.2.2 *Covered Usage Scenarios*

To better understand whether reaching more pages translates into testing more usage scenarios, as they are typically used for manual testing, we specify a set of such scenarios. We limit this experiment to Drupal because specifying scenarios and checking whether they are triggered requires some effort. We specify the following usage scenarios:

- *Add content*: The user fills and submits a form to create a new article.
- *Preview content*: The user previews a new article before submitting it.
- *Comment article*: The user writes and submits a comment on an existing article.
- *Edit content*: The user edits an existing article and submits the changes.
- *Delete content*: The user deletes an article.
- *Simple search*: The user uses the search form to search for an article.
- *Advanced search*: The user uses the advanced search, providing additional options to filter the search by including or excluding particular words.
- *User search*: The user searches for an existing user.

To detect whether a usage scenario is covered, we dynamically analyze the program during test generation. The analysis monitors the URL of the current page. If the URL matches a state where one of the usage scenarios may be covered, we inspect the state of the program and capture the events triggered by the test generator to check if the usage scenario is indeed covered. For example, to check if the "simple search" scenario is covered, we check if the input field of the search form is filled before submitting the search form.

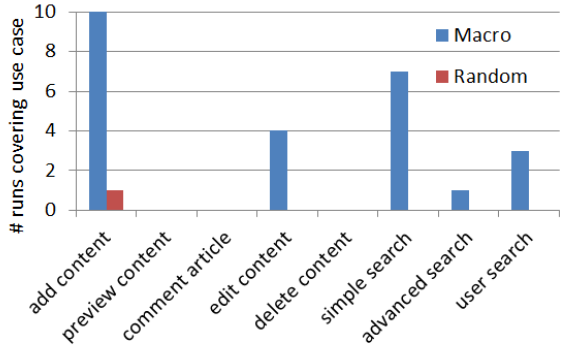


Figure 3: Number of runs that cover a specific usage scenario at least once.

Figure 3 shows how many of the ten runs with the macro-based and the random approach cover a specific usage scenario. With one exception, random testing does not cover any of the scenarios, whereas macro-based testing covers five of the eight scenarios in some or even all runs.

In addition to showing that the macro-based approach outperforms random testing, the results also illustrate some limitations of our approach. The usage scenarios missed by both approaches are due to three reasons. First, to be able to cover a scenario, the test generator must reach a particular state, which it fails to do for some scenarios within the testing budget of 500 events. Second, even when the program is in the specific state, the test generator may not trigger the "right" macro event that covers the scenario. For example, there may be other applicable macro events that the test generator triggers, or it may also select an event randomly. Finally, we observe that some of Drupal's pages are too big to be completely displayed. Since our current implementation does not interleave applying macro with scrolling the page, it cannot trigger events on hidden parts of the page.

5.2.3 *Branch Coverage*

For measuring coverage, we use JSCover[4], which instruments the JavaScript source code once it is loaded. Because finding all code that a program may load is difficult for JavaScript, we report the absolute number of covered branches, instead of a percentage.

[4]http://tntim96.github.io/JSCover/

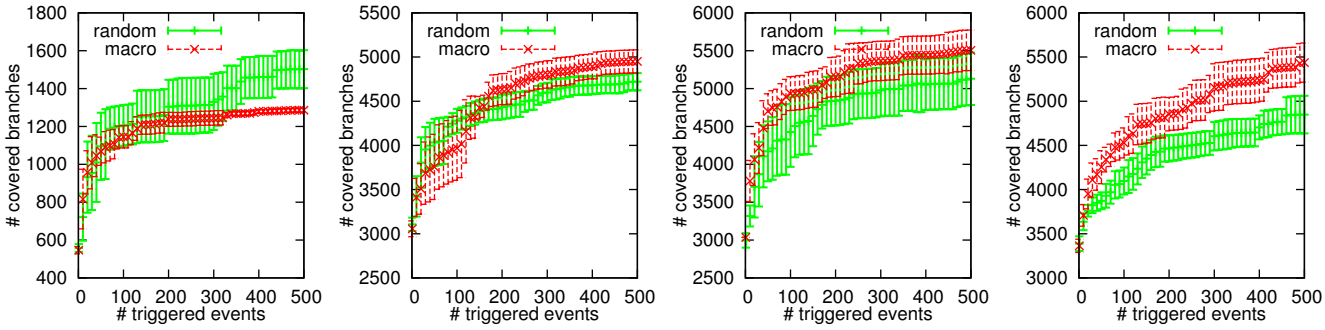


Figure 4: Branch coverage depending on the number of triggered events. The bars show the 95%-confidence interval and the average value.

Figure 4 summarizes the results. For three of the four programs, macro-based testing achieves higher coverage after 500 triggered events than random testing. For example, for Zurmo, the macro-based approach covers almost 5,500 branches, whereas random testing covers only about 4,800 branches. The exception is Drupal, where random testing outperforms macro-based testing after about 300 triggered events. One explanation for why the macro-based approach may cover fewer branches is that applying macros uses the program mostly in the expected way. In contrast, random testing also explores sequences that most users do not trigger and that executes code for handling errors. For example, while macro-based testing is likely to fill a form completely before submitting it, random testing often submits the form without filling in all required fields, which triggers code that warns about the missing fields.

5.3 Understanding the Steps of the Approach

To better understand the intermediate steps of our approach, Table 2 shows detailed information about the event traces, the macro inference, and the test generation.

Event traces. Columns 2-4 show how many trace events we record for each program, how many of them the approach identifies as frequent, and how many sequences we obtain after splitting sequences into per site subsequences. These sequences, ranging from 45 to 82 per program, are the input to the inference step.

Inference of macro events. Identifying frequent subsequences significantly increases the number of considered sequences, up to 158,969 for Zurmo. As shown by Column 6, considering only closed subsequences removes many of them, which results in up to 94,292 sequences for a program. The high number of closed frequent subsequences highlights the importance of combining subsequence mining with FSM inference. The FSM inference step summarizes the large number of sequences into a manageable number of macro events, ranging from 17 to 23 per program. Columns 8 and 9 show the average number of states and transitions per macro.

Test generation. The last two columns of Table 2 illustrate the effectiveness of the macro-based test generator. The last column shows how many of the inferred macros the generator applies at least once until reaching a final state. Overall, out of 76 macros, 66 are applied completely (87%), showing that the approach successfully uses most of the inferred macros. The second-to-last column shows how many attempts to apply a macro lead to a final state of the macro. Many attempts fail, e.g., because the macro is not fully applicable on the current page. This result confirms our design decision to combine macro-based testing with random testing, which is always applicable.

5.4 Performance

We measure how much time the different steps of the approach take. The time required for processing event traces and inferring macro events ranges between 13 seconds (Drupal) and 85 minutes (Zurmo). Most of the time is spent on mining frequent subsequences (over 99% for Zurmo). A simple way to control the computational effort of this step is to reduce the maximum length of frequent subsequences. Reducing the default value of 10 to 8 reduces the total preprocessing and inference time to values between 2.8 seconds (Drupal) and about one minute (Zurmo). However, even without this reduction, we consider the inference time to be acceptable in practice because it is a one-time effort that does not need to be repeated when generating new tests.

Macro-based test generation takes between 5.8 minutes (Drupal) and 10.8 minutes (Moodle) to create a sequence of 500 events. Most of the time is spent for determining which elements of a page have triggerable events and for executing the program under test. For comparison, these times are about 8% higher than the running times of the random test generator. We conclude that, when excluding the one-time effort of inferring macros, the macro-based approach is beneficial compared to random testing not only given a fixed budget of events to trigger but also given a fixed time budget.

5.5 Threats to Validity

There are several threats to the external validity of our findings. First, we use execution traces from users that were unfamiliar with the benchmark programs before participating in our experiment. Traces from other users and traces obtained over a longer period of time may yield other macro events and therefore also other results. Second, the analysis of covered usage scenarios is based on a relatively small set of scenarios for a single program. Third, our benchmark programs may not be representative for other web applications. To mitigate this threat, we select four popular open-source applications that cover different application domains. Finally, we compare our approach only to random test generation, and more sophisticated test generation approaches are

Table 2: Details of the data handled by the approach. For the test generation, we give the average success rate (%) for completing macro events and the number of macros that are completed at least once.

Program	Traces			Inference					Test generation	
	Events		Split	Subsequences		Macros			Successful	Completed
	Total	Freq.	seqs.	Freq.	Closed	Nb.	States (avg.)	Trans. (avg.)	attempts (%)	macros (avg.)
Drupal	4,574	4,360	82	8,331	4,393	18	14	34	21	17
MODX	7,093	6,106	45	9,300	3,995	17	9	28	27	14
Moodle	7,146	4,740	54	22,415	7,527	23	14	39	28	22
Zurmo	4,532	3,965	61	158,969	94,293	18	11	35	9	13

likely to be more effective than random testing. We believe that using macros to augment test generation is orthogonal to most other improvements over random testing. A possible threat to conclusion validity is that the random-based nature of the evaluated approaches may bias our results. To control for this threat, we provide the 95%-confidence intervals of the number of visited pages and the branch coverage.

6. RELATED WORK

Testilizer [13] is a UI-level test generator that uses an existing test suite to infer a finite state model of the program and that explores the program based on this model, while generating assertions about the DOM state. Brooks and Memon [7] infer a probabilistic finite state model of a GUI program from usage traces of the program; they use the model for guiding a test generator towards pairs of events that have a high probability to occur together. Our work differs by inferring multiple finite state models that each describe a specific interaction pattern with a program, instead of summarizing the entire program into a single model. Another difference is that macro events are applicable beyond the context from which they were inferred, whereas Testilizer aims at modeling a program without such an abstraction step. Extending our macro inference with probabilities for each event [7] is a promising direction for future work. User session-based testing [12] semi-automatically builds a model from usage traces of server-side web applications and uses them for testing. Instead, our approach infers macros fully automatically and focuses on client-side applications.

Existing capture and replay systems, such as Selenium[5], share the idea of recording user interactions to automate UI-level testing. In contrast to such systems, our approach does not replay exactly the same sequences of events as provided by a human, but applies abstractions of captured sequences of events in combination with random test generation. Besides UI-level capture and replay systems, other approaches record and replay executions at the level of individual operations [29, 39, 41].

Various approaches automatically generate UI-level tests by inferring a finite state model of the program [26, 24, 27, 11, 25, 8, 42], by using feedback from the program's execution [4], by symbolically reasoning about event handlers [40, 3, 17], or by steering toward responsiveness problems [35]. Choudhary et al. [9] experimentally compare several approaches for Android. Model-based approaches share the idea of representing the program as a finite state model. Our work differs because, instead of modeling the entire program, macro events model a specific part of a program in a generic way. In contrast to all the above approaches, our work exploits the knowledge of users to reach parts of the program that are difficult to reach otherwise.

Recent approaches for dynamically analyzing JavaScript-based web applications, such as TypeDevil [36], DLint [16], and JITProf [15], detect correctness, performance, and security problems. Combining dynamic analyses with macro-based UI-level test generation is likely to increase the effectiveness of such approaches.

Data mining techniques are widely used to extract and summarize data gathered through program analysis, e.g., for specification mining [2, 21, 14, 32, 28, 46, 32] and for detecting anomalies that are likely bugs [19, 18, 1, 37, 46, 43, 44, 33, 34]. Variants of the k-tails algorithm are used for extracting [20, 22, 23] and validating temporal specifications [31], and summarizing system logs [5] and event-based processes [10, 38]. Frequent subsequence mining helps, e.g., to identify API usage patterns [46] and to find copy-and-paste bugs [18]. Our work is the first to adapt frequent subsequence mining and the k-tails algorithm for summarizing UI events.

7. CONCLUSION

Testing complex programs with UI-level tests is an important yet non-trivial task. Automated UI-level testing techniques facilitate this task by generating sequences of events. Unfortunately, fully automated approaches often cannot reach particular states of an application and therefore do not explore all possible behavior, e.g., because it requires a particular sequences of events. This paper explores how to exploit recorded sequences of user interactions to automatically generate tests that go beyond the recorded sequences. To this end, we introduce macro events, which summarize recurring sequences of low-level UI events that correspond to a single logical step. Our approach combines macro events with random testing to create new tests that are more effective than purely random testing. In particular, the generated tests reach more pages of a web site, cover more usage scenarios, and often achieve higher coverage than random testing. Macro-based testing complements existing UI-level test generation techniques, such as random testing and model-based testing, and can be combined with them.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This research is supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within "CRISP", and by the German Research Foundation within the Emmy Noether project "ConcSys".

[5]http://www.seleniumhq.org/

9. 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 *ESEC/FSE*, pages 25–34, 2007.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[3] S. Arlt, P. Borromeo, M. Schäf, and A. Podelski. Parameterized GUI tests. In *ICTSS*, pages 247–262, 2012.

[4] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[5] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *ICSE*, pages 252–261, 2013.

[6] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. *IEEE T Comput*, 21:592–597, 1972.

[7] P. A. Brooks and A. M. Memon. Automated GUI testing guided by usage profiles. In *ASE*, pages 333–342, 2007.

[8] W. Choi, G. Necula, and K. Sen. Guided GUI testing of Android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[9] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (E). In *ASE*, pages 429–440, 2015.

[10] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM T Softw Eng Meth*, 7(3):215–249, 1998.

[11] C. Duda, G. Frey, D. Kossmann, R. Matter, and C. Zhou. Ajax Crawl: Making Ajax applications searchable. In *ICDE*, pages 78–89, 2009.

[12] S. G. Elbaum, G. Rothermel, S. Karre, and M. F. II. Leveraging user-session data to support web application testing. *IEEE Trans Softw Eng*, pages 187–202, 2005.

[13] A. M. Fard, M. MirzaAghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *ASE*, pages 67–78, 2014.

[14] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008.

[15] L. Gong, M. Pradel, and K. Sen. JITProf: Pinpointing JIT-unfriendly JavaScript code. In *ESEC/FSE*, pages 357–368, 2015.

[16] L. Gong, M. Pradel, M. Sridharan, and K. Sen. DLint: Dynamically checking bad coding practices in JavaScript. In *ISSTA*, pages 94–105, 2015.

[17] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *ISSTA*, pages 67–77, 2013.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans Softw Eng*, 32(3):176–192, 2006.

[19] Z. Li and Y. Zhou. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE*, pages 306–315, 2005.

[20] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *WCRE*, pages 51–60, 2006.

[21] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE*, pages 265–275, 2006.

[22] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *ESEC/FSE*, pages 345–354, 2009.

[23] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[24] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *ICST*, pages 121–130, 2008.

[25] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *ICST*, pages 81–90, 2012.

[26] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw Test Verif Reliab*, pages 137–157, 2007.

[27] A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *ICSE*, pages 210–220, 2009.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *ESEC/FSE*, pages 383–392, 2009.

[29] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[30] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Trans Knowl Data Eng*, 16(11):1424–1440, 2004.

[31] M. Pradel, P. Bichsel, and T. R. Gross. A framework for the evaluation of specification miners based on finite state machines. In *ICSM*, pages 1–10, 2010.

[32] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *ASE*, pages 371–382, 2009.

[33] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*, pages 288–298, 2012.

[34] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE*, pages 925–935, 2012.

[35] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. In *OOPSLA*, pages 33–47, 2014.

[36] M. Pradel, P. Schuh, and K. Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *ICSE*, pages 314–324, 2015.

[37] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE*, pages 240–250, 2007.

[38] S. P. Reiss and M. Renieris. Encoding program executions. In *ICSE*, pages 221–230, 2001.

[39] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, pages 677–694, 2011.

[40] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528, 2010.

[41] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In *ESEC/FSE*, pages 488–498, 2013.

[42] S. Thummalapenta, K. V. Lakshmi, S. Sinha, N. Sinha, and S. Chandra. Guided test generation for web applications. In *ICSE*, pages 162–171, 2013.

[43] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *ICSE*, pages 496–506, 2009.

[44] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *ASE*, pages 295–306, 2009.

[45] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *SDM*, pages 166–177, 2003.

[46] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP*, pages 318–343, 2009.