EventBreak: Analyzing the Responsiveness of User Interfaces through Performance-Guided Test Generation

Michael Pradel Parker Schuh George Necula Koushik Sen
EECS Department
University of California, Berkeley

Abstract

Event-driven user interface applications typically have a single thread of execution that processes event handlers in response to input events triggered by the user, the network, or other applications. Programmers must ensure that event handlers terminate after a short amount of time because otherwise, the application may become unresponsive. This paper presents EventBreak, a performance-guided test generation technique to identify and analyze event handlers whose execution time may gradually increase while using the application. The key idea is to systematically search for pairs of events where triggering one event increases the execution time of the other event. For example, this situation may happen because one event accumulates data that is processed by the other event. We implement the approach for JavaScript-based web applications and apply it to three real-world applications. EventBreak discovers events with an execution time that gradually increases in an unbounded way, which makes the application unresponsive, and events that, if triggered repeatedly, reveal a severe scalability problem, which makes the application unusable. The approach reveals two known bugs and four previously unknown responsiveness problems. Furthermore, we show that EventBreak helps in testing that event handlers avoid such problems by bounding a handler's execution time.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.8 [*Software Engineering*]: Metrics

Keywords Web applications; Testing; Test generation; Responsiveness; Performance

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660233

1. Introduction

Any event-driven user interface application, such as a web application or an application running on mobile devices, should react quickly to events triggered by the user. Such applications typically have a single thread of execution that processes event handlers. For example, the JavaScript execution model implemented in popular web browsers has an event dispatcher that takes event handlers from a queue and dispatches them to a single thread of execution. While an event handler is executing, the browser does not react on user input. Therefore, developers of web applications must ensure that each event handler terminates quickly, for example, by ensuring that the execution time cannot exceed particular bounds.

If an event handler runs too long, the application is perceived as unresponsive. To deal with this problem, platforms for event-driven applications monitor event handlers and ask the user to interrupt long-running handlers. For example, Firefox and the Android platform show a pop-up if an event handler runs longer than a maximum execution time and ask the user to force-stop the script and application, respectively. Developers are highly motivated to avoid this situation because users may perceive it as a crash-like termination of the program. Unfortunately, performance problems are common in event-driven applications and many of them lead to unresponsive applications [22].

As a motivating example, consider a bug in the Joomla content management system, a complex web application used by various popular web sites.[1] The administrative interface of a Joomla-based web site allows for adding menu items to the site, and it provides a way to set properties of entire menus. The event handler triggered when changing the properties of a menu validates the names of all menu items every time it is called. This validation is computationally expensive and therefore, the execution time of the handler increases quickly when a user adds more menu items. For web sites with a large number of menu items, this behavior makes Joomla's administrative interface unresponsive.

[1] `http://www.joomla.org`

Users of Joomla have reported this problem, and it has been addressed by the developers.[2]

Traditional manual testing is very unlikely to discover this problem because the application becomes unresponsive only after repeatedly executing a particular sequence of input events. This sequence involves creating a menu item and saving the properties of the corresponding menu, as well as other events needed to navigate between pages that allow for triggering these events. Even if a test triggers this sequence, the responsiveness problem may easily be missed because the execution time of the buggy event handler increases gradually. Furthermore, testing is often optimized for increasing code coverage, whereas here we need tests that persist in one area of the code long enough to spot performance trends. A brute-force search for event sequences that trigger responsiveness problems is prohibitive, since real-world web applications typically involve hundreds of pages, each having hundreds of possible events to trigger.

This paper addresses the problem of analyzing the responsiveness of event-driven applications through automated testing. We present EventBreak, a performance-guided test generation technique that exercises a web application by creating sequences of user input events, such as clicking on DOM elements, scrolling, and filling forms. The key idea is to leverage measured performance of event handlers to steer the test generation towards potential responsiveness problems. EventBreak identifies pairs of events that, if triggered alternately, may eventually make the application unresponsive because triggering one event increases the execution time of the other event. We call such a pair of events a *slowdown pair*. The test generator systematically analyzes potential slowdown pairs by repeatedly navigating between the two events with the help of an inferred, approximate finite-state model of the application. The output of EventBreak is a set of *cost plots* that show how the execution time of one event changes as a result of triggering another event. Developers can use these plots to check whether event handlers scale as expected and to identify handlers with an increasing and potentially unbounded cost.

We envision two usage scenarios of our approach. First, it can serve as a fully automated testing technique. In this scenario, EventBreak starts by randomly exploring the application under test for potential responsiveness problems, and then targets the exploration toward potential slowdown pairs discovered during the random exploration. Second, the approach can leverage existing tests or usage traces. In this scenario, EventBreak exploits the available information by inferring potential slowdown pairs and by analyzing them in more detail. That is, the approach amplifies existing testing efforts while focusing on potential responsiveness problems.

To the best of our knowledge, this paper is the first to address responsiveness problems in event-driven application through performance-guided test generation. The closest existing work falls into two categories. On the one hand, there are test generation approaches for event-driven applications [2, 3, 6, 16, 27, 33, 34]. They aim for high coverage of the application's source code, which is an appropriate goal for discovering correctness problems, but it does not necessarily expose responsiveness problems. On the other hand, there are static [17, 22] and dynamic [30, 35, 38, 40] analyses to detect performance problems. These analyses focus on particular root causes of problems and do not address event-driven applications. In contrast to existing work, EventBreak focuses on the cost of event handlers in event-driven applications and systematically analyzes an application's responsiveness.

We evaluate EventBreak with three real-world web applications and show that the approach detects slowdown pairs that correspond to responsiveness problems. EventBreak detects two known bugs and four previously unknown responsiveness problems. The slowdown pairs include a problem in Joomla that makes the application unresponsive, and a problem in Drupal that crashes the application and therefore makes it unusable. The targeted test generation effectively explores potential slowdown pairs despite using an approximate model of the application. On average, EventBreak successfully triggers 89% of all target events it tries to trigger when exploring slowdown pairs, while taking only 33 events to reach a state where a target event can be triggered.

In summary, this paper contributes the following:

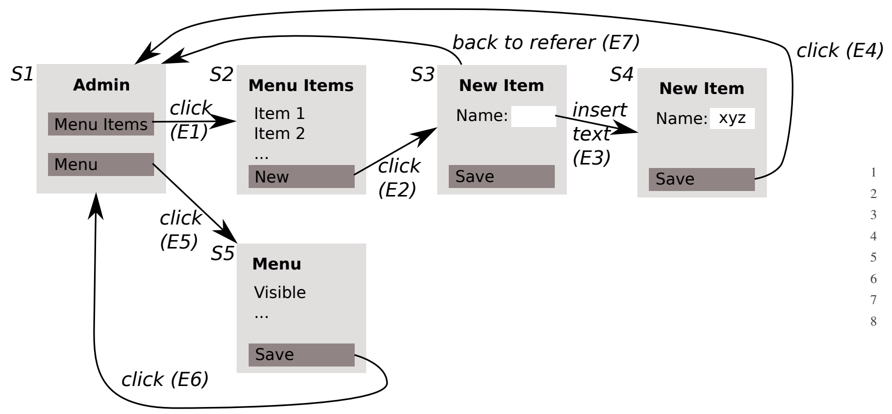
- The first automated testing technique for analyzing the responsiveness of event-driven user interface applications.
- Algorithms that leverage measured performance to generate test input sequences that steer an application toward events with gradually increasing cost.
- Empirical results that show that the approach scales to real-world web applications and that it provides insights about the performance and responsiveness of these applications.

2. Motivating Example and Overview

The following section motivates our work by elaborating on the motivating example mentioned in Section 1 and provides an overview of our approach. The example application is the administrative interface of the Joomla content management system. The administrative interface is used to manage web sites build upon Joomla. For illustration purposes, we simplify the application and the example.

Figure 1a illustrates different states of the web application and how states can be reached from other states by triggering events. Lightgray boxes represent states and arrows show how triggering an event leads to another state. In the example, state $S1$ is the initial state of the application. By clicking on the "Menu Items" button (event $E1$), the user gets to state $S2$. Here, the application lists all menu items and provides

[2] Issue 30274



(a) Application states and events.

```
// validate.js: Validate form fields
var elements = form.getElements('fieldset')
    .concat(Array.from(form.elements));
for (var i=0;i < elements.length; i++) {
  if (this.validate(elements[i]) == false) {
    valid = false;
  }
}
```

(b) Code from Joomla that is executed when saving the menu (event $E6$).

Figure 1. Motivating example (simplified version of Joomla Issue 30274).

a button for creating a new menu item (event $E2$). Clicking this button leads to state $S3$, where the user can choose a name for the new menu item by inserting text into a text field (event $E3$). Once the text field is filled, the application is at state $S4$, where the user can save the menu item and get back to state $S1$ (event $E4$). Instead of adding new menu items, the user can also modify the properties of the entire menu by clicking “Menu” in $S1$ (event $E5$), which leads to $S5$. At $S5$, the user can save the menu and return to $S1$ (event $E6$). The figure omits many more states and events. In particular, the user can reach most states from any other state by directly accessing the corresponding URL, as illustrated by event $E7$.

The application has a responsiveness problem because the execution time of event $E6$ increases whenever event $E4$ is triggered. Figure 1b shows an excerpt of the JavaScript code that handles event $E6$. The code validates that all form fields related to the menu contain correct values. The array `elements` contains, among other entries, one entry for each menu item of the menu. Unfortunately, the number of menu items is unbounded and the code in Figure 1b does not bound the number of iterations. As a result, the execution time of the code in Figure 1b grows whenever the user adds a new menu item. This unbounded growth makes the application unresponsive when $E6$ is triggered while having a large number of menu items.

There are three typical ways to fix such a problem. First, the developers can bound the amount of computation done in a single handler. In the example, one might split the `elements` array into chunks and handle each in a separate handler, so the browser can react to other events between executing these handlers. Second, an application can bound the number of data items that can exist in the system. For example, the Joomla developers may limit the maximum number of menu items. Third, the developers can modify the application logic so that triggering one event does not directly influence the execution time of handling another event. The Joomla developers fix the example bug using the third approach by adding a cache that avoids re-validating menu items when the menu is saved.

EventBreak discovers this responsiveness problem in two phases. In the first phase, EventBreak observes an execution of the application and records the cost associated with each triggered event. We measure the cost of an event as the number of conditionals that are evaluated during its execution. For example, the approach may record several occurrences of $E4$ and $E6$ and their respective costs. EventBreak analyzes the recorded costs and infers that triggering $E4$ potentially increases the cost of executing $E6$.

In the second phase, EventBreak further explores the potential dependence between $E4$ and $E6$ by generating a sequence of input events that alternately triggers $E4$ and $E6$. To address the problem that triggering one event directly after the other may not be possible, the approach uses an inferred finite state model of the application [21], similar to the model shown in Figure 1a. Based on the model, EventBreak searches for short paths between those states that allow for triggering $E4$ and $E6$, respectively, and repeatedly follows these paths. For example, EventBreak may repeatedly execute the event sequence $E1-E2-E3-E4-E5-E6$. Our approach addresses the challenge that the inferred model approximates the application, and that the shortest path in the inferred model may not be feasible in the application. To address this challenge, the approach considers not only the shortest path but also longer paths, and it randomizes the search for paths. After exploring the dependence between $E4$ and $E6$, EventBreak finally reports a cost plot that shows that $E6$’s cost increases linearly when triggering $E4$.

3. Definitions and Problem Statement

In the following, we define the most important terms used throughout the paper, and we specify the problem that we are addressing.

3.1 Definitions

The basic unit of computation in event-driven programs is an event handler. This paper considers JavaScript-based web applications and identifies events as follows:

Definition 1 (Event)
An event is a tuple (d, t, s, s')*, where*

- d *is an identifier of the DOM element on which the event is dispatched*
- t *is the type of event*
- s *is the state of the application before triggering the event*
- s' *is the state of the application after the event has been triggered*

As the identifier d, we use the XPath of a DOM element, which uniquely identifies an element, such as a button, on a page. The type t of an event indicates what kind of user input the event refers to. For example, the type can be *click* or *mouseover*. The states s and s' refer to an abstraction of the web application's state. For our experiments, we use the document title and the URL of the current page to abstract the page's state. URLs give a reasonable state abstraction for applications that use "deep linking", that is, the URL reflects enough of the state to bring a user back to the same state. Finding a reasonable state abstraction for a particular web application is a problem orthogonal to this work [24, 34]. In addition to events dispatched on DOM elements, EventBreak supports several special events, such as scrolling the page and going back to a previously seen URL. To go back to previously seen URLs, EventBreak keeps track of the referrer URL whenever a link leads to a new page and creates a "back to referrer" event for each observed referrer.

Triggering an event causes the execution of its event handler. We measure the cost of handling an event as follows:

Definition 2 (Cost of event handling)
The cost $c(e)$ *of executing the handler of an event* e *is the number of conditionals evaluated as a result of triggering* e*.*

This definition considers a logical cost of handling an event. Instead, we could measure the actual cost on the client machine, for example, the handler's wallclock execution time. The rationale for preferring a logical cost is to avoid problems related to accurately measuring machine-level performance [8, 11, 29].

3.2 Problem Statement

The problem we address in this paper is analyzing the responsiveness of event-driven applications. To this end, we search for pairs of events where triggering one event increases the cost of executing another event.

Definition 3 (Slowdown pair)
A slowdown pair (e_{cause}, e_{effect}) *consist of an event* e_{cause} *and an event* e_{effect}*, where for any sequence* $e_{effect}, .., e_{cause}, .., e_{effect}$*, the cost of the second* e_{effect} *is larger than the cost of the first.*

As illustrated in Section 2, such pairs of events may correspond to responsiveness problems that developers should address. In addition to revealing responsiveness problems, searching for slowdown pairs is useful for testing that mechanisms to avoid long-running event handlers work as expected.

We focus on pairs of events for two reasons. First, in a preliminary study of real-world responsiveness bugs, we notice several bugs that show up if two particular events are repeated alternately. Second, analyzing pairs of events reduces the search space to a manageable size, even for complex web applications. If we instead analyze whether arbitrary sequences of events expose responsiveness problems, the number of possible sequences becomes too large to explore in practice.

4. Performance-Guided Test Generation

This section presents our approach to find and explore slowdown pairs in complex web applications through automated, targeted test generation. EventBreak consist of two main phases:

1. The first phase explores the application to record a trace of events and their respective cost (Section 4.1). By default, the first phase of EventBreak loads a web site and explores it by randomly triggering events. As an alternative, EventBreak can build upon existing testing efforts and use a trace obtained from executing a test suite or from usage traces recorded from real users. The approach leverages the trace to infer potential slowdown pairs (Section 4.2).
2. The second phase systematically explores potential slowdown pair by experimentally evaluating the hypothesis that a given pair of events is indeed a slowdown pair (Section 4.3). To explore a potential slowdown pair (e_{cause}, e_{effect}), EventBreak triggers the two events of the pair in an alternating way and analyzes how the cost of e_{effect} changes over time. We observe the problem that triggering a particular event may only be possible in a particular state. EventBreak addresses this problem by using an inferred finite-state model of the application and by using the model to reach a state where the desired event can be triggered. If EventBreak finds evidence that a potential slowdown pair is indeed a slowdown pair, it summarizes the pair into a cost plot. This plot shows the cost of e_{effect} as a function of the number of times that e_{cause} has been triggered (Section 4.4).

4.1 Gathering Performance Data

The first phase of EventBreak dynamically analyzes an interactive execution of the web application to record a history of its events and their associated cost.

Definition 4 (Event-cost history)
An event-cost history h is a sequence of pairs $(e_0, c_0), .., (e_k, c_k)$, where

- *e_i is an event;*
- *c_i is the cost $c(e_i)$ of handling the event;*
- *e_0 has been triggered in the initial state of the web application; and*
- *e_{i+1} has been triggered in the state reached by e_i.*

To obtain an event-cost history, we load the initial page of the application and trigger a sequence of events. To facilitate this process in the absence of manually created tests, EventBreak comes with a simple test generator that randomly picks the next event to trigger from the set of all currently enabled events. An event is enabled if the user can trigger it in the current state of the application, and not enabled otherwise. For example, a button that is hidden behind another DOM element is not enabled. After loading the initial page, the test generator repeatedly executes the following steps:

1. *Query the browser for all currently enabled events.*
2. *Randomly pick an event from all enabled events.* The test generator randomly chooses among all enabled events with a uniform distribution, with the exception of events that go back to the URL of a previously seen referrer. If any such "back to referrer" event is available, the test generator chooses to go back to a referrer with a user-defined probability β, and picks from all other events with probability $1-\beta$ (we set β to 0.1). If the test generator decides to go back to a referrer, it picks from all referrers seen in the history with a uniform distribution. The rationale for this approach is that the number of referrer URLs may grow in an unbounded way, for example, because the application generates new URLs. Without a fixed β, the probability to go back to a referrer would become larger over time, while the probability to trigger any other kind of event would decrease.
3. *Execute and measure cost.* The test generation executes the event e, measures its cost $c(e)$, and appends the pair $e, c(e)$ to the event-cost history.

As an alternative to randomly exploring the application, EventBreak can incorporate manual testing by recording the events triggered by a tester and their associated cost.

To measure the cost of events (Definition 2), EventBreak instruments all JavaScript code of the application. The goal of the instrumentation is to keep track of the number of conditionals evaluated in reaction to an event. To this end, EventBreak adds the following instrumentation:

- A global variable `condCtr` that counts the number of evaluated conditionals.
- For each branching statement (`if`, `while`, `do while`, `for`, `for in`), a statement `condCtr++` at the beginning of each branch.

Id	Event	Cost
1	E5	2
2	**E6**	14
3	E1	4
4	E2	3
5	E3	1
6	E4	8
7	E5	2
8	**E6**	18
9	E1	4
10	E2	3
11	E7	2
12	E5	2
13	**E6**	18
14	E1	4
15	E2	3
16	E3	1
17	E4	8
18	E5	2
19	**E6**	22

Potential slowdown pairs:

Pair	Support	Confidence
(E1, **E6**)	2	2/3=67%
(E2, **E6**)	2	2/3=67%
(E3, **E6**)	2	2/2=100%
(E4, **E6**)	2	2/2=100%
(E5, **E6**)	2	2/3=67%

Details on pair (E1, E6):

- Supporting evidence: Cost of E6 increases twice when E1 occurs in between (Ids 2 and 8, Ids 13 and 19).
- Refuting evidence: Cost of E6 does not increase even though E1 occurs in between (Ids 8 and 13).

Figure 2. Example of event-cost history and inference of potential slowdown pairs.

- For each function, a statement `condCtr++` at the beginning of the function. We increment the counter at each function entry because we consider function dispatch as a conditional, and to keep track of the cost of recursive calls.

Example An execution of the example application in Figure 1a may give the history shown in the left part of Figure 2. Each event has an associated cost. In practice, the cost values are higher because many event handlers execute hundreds or even thousands of conditionals. The following section describes how EventBreak uses these costs.

4.2 Inferring Potential Slowdown Pairs

Based on the event-cost history, EventBreak identifies potential slowdown pairs by considering each pair of unique events in the history as a candidate slowdown pair. For each such candidate slowdown pair c, the approach gathers evidence that supports the hypothesis that c is a slowdown pair and evidence that refutes this hypothesis. Based on this evidence, EventBreak computes a confidence score (intuitively, how likely it is that c is a slowdown pair) and a support value (intuitively, how many occurrences of the slowdown pair exist in the history) for each candidate slowdown pair.

Algorithm 1 details our approach to infer potential slowdown pairs from a given history. The algorithm computes the set $\mathcal{E}$ of unique events in the history, where uniqueness refers to Definition 1. For example, if the history contains two click events for a button with the same identifier while being in the same application state, and if clicking the button leads to the same state, then these two events are considered two instances of the same event. The algorithm computes the

Algorithm 1 Infer potential slowdown pairs.

```
Input: Event-cost history $h$
Output: Set $\mathcal{S}$ of potential slowdown pairs
 1: $\mathcal{E} \leftarrow$ unique events in $h$
    // candidate slowdown pairs:
 2: $\mathcal{S}_{cand} \leftarrow \{(e_{cause}, e_{effect}) \mid e_{cause} \in \mathcal{E}, e_{effect} \in \mathcal{E}\}$
 3: Initialize $AllEvidence$ to 0 for all $s \in \mathcal{S}_{cand}$
 4: Initialize $SuppEvidence$ to 0 for all $s \in \mathcal{S}_{cand}$
 5: for all $e_{effect} \in \mathcal{E}$ do
 6:   if $e_{effect}$ occurs at least $supp_{min} + 1$ times in $h$ then
 7:     $c_0 \leftarrow -1$ // cost of most recent $e_{effect}$
 8:     $\mathcal{E}_{between} \leftarrow \emptyset$
 9:     for all $(e, c)$ in $h$ do
10:       if $e = e_{effect}$ then
11:         if $c_0 > -1$ then // Second occurrence of $e_{effect}$
12:           for all $e_{cause} \in \mathcal{E}$ do
13:             if $c_0 < c$ or $e_{cause} \in \mathcal{E}_{between}$ then
14:               Increment $AllEvidence(e_{cause}, e_{effect})$
15:             if $c_0 < c$ and $e_{cause} \in \mathcal{E}_{between}$ then
16:               Increment $SuppEvidence(e_{cause}, e_{effect})$
17:         $c_0 \leftarrow c$
18:         $\mathcal{E}_{between} \leftarrow \emptyset$
19:       else
20:         $\mathcal{E}_{between} \leftarrow \mathcal{E}_{between} \cup \{e\}$
21: $\mathcal{S} \leftarrow \emptyset$
22: for all $s \in \mathcal{S}_{cand}$ do
23:   if $AllEvidence(s) > 0$ and $SuppEvidence(s) > supp_{min}$ and $\frac{SuppEvidence(s)}{AllEvidence(s)} > conf_{min}$ then
24:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
```

set $\mathcal{C}$ of candidate slowdown pairs as the set of all pairs of events in $\mathcal{E}$.

For the example execution in the left part of Figure 2, the algorithm initially considers each pair built from $\{E1, .., E7\}$ as a candidate slowdown pair.

To validate or invalidate the hypothesis that a candidate slowdown pair $s = (e_{cause}, e_{effect})$ is a slowdown pair, the algorithm searches for evidence that supports or refutes this hypothesis. As *evidence that supports the hypothesis*, we consider two occurrences of e_{effect} where the cost of the first occurrence is smaller than the cost of the second occurrence, and where e_{cause} occurs between the two occurrences of e_{effect}. As *evidence that refutes the hypothesis*, we consider two occurrences of e_{effect} where either the cost does not increase, or where e_{cause} does not occur between the two occurrences of e_{effect}. The first case (the cost does not increase) considers the situation where one event does not influence the cost of another. The second case (e_{cause} does not occur between the occurrences of e_{effect}) considers the situation where the cost of e_{effect} grows but not because of e_{cause}. The algorithm considers only the shortest possible sequences with two occurrences of e_{effect}.

For the example in Figure 2, consider the candidate pair $(E1, E6)$. As evidence that supports the hypothesis that this pair is a slowdown pair, the algorithm finds that the cost of $E6$ increases twice when $E1$ occurs in between. As evidence that refutes the hypothesis, the algorithm finds that the cost of $E6$ does not increase once even though $E1$ occurs in between. The algorithm ignores, for example, the sequence between Id 2 and Id 13 because it is not the shortest possible sequence that contains two occurrences of $E6$.

To gather evidence about each candidate slowdown pair s, the algorithm maintains two maps: $AllEvidence$ maps s to the total number of pieces of evidence related to s, and $SuppEvidence$ maps s to the number of pieces of evidence that support the hypothesis that s is a slowdown pair. Lines 3 and 4 initialize these maps to zero for all candidate slowdown pairs. The algorithm iterates through all events e_{effect} (line 5) and for each of them, iterates once through the history (line 9). For efficiency, events that occur so infrequently that their candidate slowdown pairs cannot exceed a specified minimum support are ignored (line 6). While iterating through the history, the algorithm maintains the cost c_0 of the current most recent occurrence of e_{effect} and the set $\mathcal{E}_{between}$ of events that occur between the two occurrences of e_{effect}. If the algorithm finds an occurrence of e_{effect} that has been preceded by another occurrence of e_{effect}, it updates the evidence maps $AllEvidence$ and $SuppEvidence$ for all candidate pairs that involve e_{effect} (lines 14 and 16).

Based on the evidence gathered for all candidate slowdown pairs, the final part of the algorithm (lines 21 to 24) computes the set $\mathcal{S}$ of potential slowdown pairs. For this purpose, the algorithm checks for each candidate s whether the supporting evidence reaches the minimum support and whether the confidence $\frac{SuppEvidence(s)}{AllEvidence(s)}$ reaches the minimum confidence. All slowdown pairs that fulfill these requirements are returned as potential slowdown pairs.

The time complexity of Algorithm 1 is $\mathcal{O}(|h| \cdot e^2)$, where $|h|$ is the number of event-cost pairs in the history h, and where e is the number of unique events in h. Since the number of unique events of an application is typically bounded,[3] the algorithm scales well to large initial executions.

For Figure 2, the algorithm finds five potential slowdown pairs with confidence and support as shown in the table on the upper right of the figure.

4.3 Targeted Exploration of Slowdown Pairs

The inferred potential slowdown pairs are likely to exist based on the information available in the event-cost history. The following presents a targeted test generation approach to confirm whether a potential slowdown pair (e_{cause}, e_{effect}) can indeed increase the cost of handling e_{effect}. The basic

[3] The number of unique events depends on how to represent the state of the application, which we discuss in Section 7.

idea is to analyze the slowdown pair by trying to trigger a sequence of events e_{effect}, .., e_{cause}, .., e_{effect}, .., e_{cause}, .., e_{effect} etc., and to check whether the cost $c(e_{effect})$ increases. A major challenge in realizing this idea is that triggering an event requires the application to be in a state where the event is available.

4.3.1 Leveraging an Application Model

To allow the test generator to trigger a particular event e, it may be necessary to trigger a sequence of other events that brings the application into a state where e is available. To effectively find such a sequence of events, EventBreak leverages a finite state model of the application. Such an *application model* is a non-deterministic finite state machine that has model states, which abstract the application states, and transitions, which represent events. EventBreak abstracts the web application's state into a model state based on the document title and URL of the current page (Section 3.1).

Manually specifying an application model for a complex web application is a tedious task, and there is no explicit model for most existing web applications. Instead of relying on a manually created model, EventBreak obtains an approximate application model by applying an algorithm for passive automata learning to the sequence of events in the event-cost history. Inferring a model is not the contribution of this paper and we use a state of the art approach for inferring a finite state machine from a given sequence of events [21].

Our targeted test generation approach must address the problem that the learning algorithm yields an application model that approximates the application. The model is approximate in two ways. First, the model may accept sequences of events that are not possible in the application because the learning algorithm heuristically merges model states that may not correspond to equivalent application states. That is, the model may partly overapproximate the application's behavior. Second, the model may not accept sequences of events that are possible in the application because the history does not contain the particular sequence of events. That is, the model may partly underapproximate the application's behavior.

Figure 3 shows the approximate application model that the learning algorithm infers from the history in Figure 2. The bold items represent states; arrows represent transitions. The model overapproximates the application because it specifies that triggering $E4$ (save the new menu item) in the "New Item" state leads back to the "Admin" state. However, this is only true after $E3$ (give the new item a name) has been triggered.

4.3.2 Systematic Exploration of Slowdown pairs

Based on an approximate application model, EventBreak systematically explores each potential slowdown pair. Algorithm 2 summarizes the test generation approach. The goal of the algorithm is to validate or invalidate the hypothesis that a potential slowdown pair is indeed a slowdown pair.

Algorithm 2 Targeted exploration of slowdown pairs.

Input: Approximate application model M, set $\mathcal{S}$ of potential slowdown pairs
Output: Map G from slowdown pairs to cost vectors

```
1:  $G \leftarrow$ empty map
2:  $s \leftarrow s_0$ of $M$ // initial state of application model
3:  for all $(e_{cause}, e_{effect}) \in \mathcal{S}$ do
4:    $costs \leftarrow$ empty list
5:    $success \leftarrow true$
6:    while $isWorth(costs)$ and $|costs| < maxCosts$ and $success$ do
7:      $(success, M, s, c) \leftarrow trigger(M, s, e_{effect})$
8:      if $success$ then
9:        $costs.append(c)$
10:     $(success, M, s, c) \leftarrow trigger(M, s, e_{cause})$
11:   if $|costs| = maxCosts$ then
12:     $G \leftarrow G \cup \{(e_{cause}, e_{effect}) \mapsto costs\}$
```

Algorithm 3 Function $trigger$, which tries to trigger a target event, possibly by triggering other events beforehand.

Input: Approximate application model M, current state s, target event e_{target}
Output: Success flag, updated model M, updated state s, cost c

```
1:  $eventsLeft \leftarrow maxEvents$
2:  while $eventsLeft > 0$ do
3:    $\mathcal{E}_{avail} \leftarrow availableEvents()$
4:    $seqs \leftarrow shortSeqsToTarget(M, s, e_{target}, \mathcal{E}_{avail})$
5:    if $|seqs| = 0$ then
6:      return $(false, M, s, -1)$
7:    $e \leftarrow pickFromFirstEvents(seqs)$
8:    $(s_{dest}, c) \leftarrow triggerInApp(e)$
9:    $(M, s) \leftarrow updateModelAndState(M, s, e, s_{dest})$
10:   if $e = e_{target}$ and $e_{target}.s' = s_{dest}$ then
11:     return $(true, M, s, c)$
12:   $eventsLeft \leftarrow eventsLeft - 1$
13: return $(false, M, s, -1)$
```

The algorithm takes a model M and a set $\mathcal{S}$ of potential slowdown pairs, and it returns a map G that assigns a list of cost values of e_{effect} to each pair (e_{cause}, e_{effect}) that is found to be a slowdown pair. Since the pair is found to be a slowdown pair, the cost values in the list are monotonically increasing (Definition 3).

The algorithm iterates through all potential slowdown pairs (line 3) and analyzes each of them. To analyze a particular slowdown pair (e_{cause}, e_{effect}), the algorithm tries to alternately trigger the two events and all events that may be necessary to enable triggering these two events. While doing so, the test generator builds a list $costs$ of cost values for executing e_{effect} (line 9). The exploration of a slowdown pair continues until one of the following conditions holds (line 6):

- The $costs$ of the slowdown pair are not strictly monotonically increasing, that is, the targeted exploration reveals that the potential slowdown pair is not a slowdown pair. The helper function $isWorth$ checks this condition. Alternative implementations of $isWorth$ are possible, for example, to check the overall trend of e_{effect}'s cost instead of requiring a strictly monotonic increase.
- The length of $costs$ has reached a user-specified length $maxCosts$. That is, the test generation has alternately triggered the two events of the slowdown pair $maxCosts$ times and the cost of e_{effect} has increased each time. In this case, the test generator confirms that the pair is indeed a slowdown pair.
- The test generator is unable to alternately trigger the two events of the slowdown pair, which is indicated by $success$ becoming $false$. For example, this situation may happen when the test generator cannot find a sequence of events that ends in the desired event. In this case, the test generator cannot confirm that the pair is a slowdown pair.

For our running example, the algorithm considers all potential slowdown pairs in Figure 2 and quickly discards most of them. For example, the pair $(E5, E6)$ is discarded because triggering $E5$ and $E6$ alternately does not increase the cost of $E6$. The algorithm confirms the pair (E_4, E_6) as a slowdown pair because repeatedly triggering E_4 indeed increases the cost of executing E_6.

4.3.3 Triggering a Target Event

When the test generator tries to trigger a particular event (either e_{cause} or e_{effect}), we call this event the *target event*. To trigger a particular target event, the test generator calls $trigger$, which is summarized in Algorithm 3. The algorithm takes the application model M, the current state s, and a target event e_{target}, and tries to trigger events that eventually allow for triggering the target event. The basic idea is to execute the following steps:

1. Search the application model for short sequences $seqs$ of events from the current state to a state where the target event is available (line 4).
2. From all events that appear at the beginning of a sequence in $seqs$, randomly pick an event e (line 7). According to the model, triggering e brings the application closer to a state where e_{target} is available.
3. Trigger e in the application (line 8), which yields the destination state s_{dest} and the cost $c(e)$. The algorithm updates the application model and the current state (line 9).
4. If the triggered event e is the target event e_{target}, return the updated application model and state, along with the cost c of the target event (line 11). Otherwise, continue these steps until we reach e_{target} or until the algorithm has triggered $maxEvents$ events. In the latter case, the algorithm returns $false$ to indicate that it cannot trigger e_{target}.

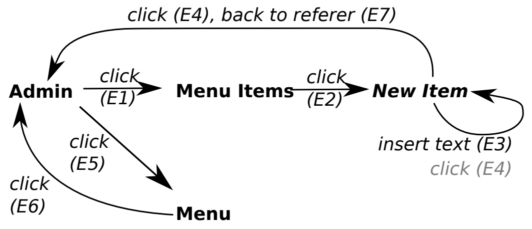


Figure 3. Approximate application model for the application in Figure 1a.

Function $shortSeqsToTarget$ queries the model to obtain a set of sequences from the current state to a state s_{target} where the target event is available. The function performs a breadth-first search starting at the current state and returns all shortest sequences that lead to s_{target}. In addition to all shortest sequences, the algorithm returns some longer sequences by probabilistically continuing the breadth-first search beyond a shortest sequence. Whenever the breadth-first search could follow another event that extends the current sequence beyond the length of the shortest sequence, it does so with a specified probability γ (we set γ to 0.33). This approach includes some sequences that are longer than the shortest sequence and probabilistically terminates the search eventually. The rationale for considering not only the shortest sequences is to find a sequence even if the model is imprecise (Section 4.3.4).

After triggering an event e in the application, Algorithm 3 calls $updateModelAndState$. This function updates the current state to the destination state of e. Furthermore, the function updates the model M to reflect what triggering e has revealed about the application. If the destination state of e matches the destination state predicted by the model, than the model remains unchanged. If, however, the model predicts an incorrect destination state for e, then the approach updates the model by rerunning the model learning algorithm on the current history, which ends with event e.

Example Consider the history in Figure 2 and the potential slowdown pair $(E4, E6)$. To explore this pair, Algorithm 2 tries to trigger $E6$, $E4$, $E6$, etc. Suppose, the algorithm has successfully triggered $E6$ and the application is again at the "Admin" state. Now, Algorithm 2 invokes Algorithm 3 to trigger event $E4$ and possibly other events that may be necessary to reach $E4$. Algorithm 3 uses the model in Figure 3 to perform the following iterations of its main loop:

1. From its current state s, "Admin", the algorithm finds two short sequences leading to $E4$: $E1-E2-E4$ and $E1-E2-E3-E4$. It picks $E1$ as the next event (line 7), which leads to state "Menu Items".

2. At state “Menu Items”, the algorithm again finds two short sequences leading to $E4$: $E2 - E4$ and $E2 - E3 - E4$. It picks $E2$ as the next event, which leads to state “New Item”.
3. At state “New Item”, the algorithm again finds two sequences $E4$ and $E3 - E4$, that is, the algorithm must decide between $E3$ and $E4$. Suppose the algorithm picks $E4$. After triggering this event, the application remains at state “New Item”, which does not match the current model. Therefore, the algorithm re-learns the model, which adds the transition printed in gray to Figure 3.
4. Still at state “New Item”, the algorithm now finds three short sequences that lead to $E4$: $E4$, $E3 - E4$, and $E4 - E4$. That is, the algorithm randomly decides between $E4$ and $E3$. Suppose, it picks $E3$, which fills in a name for the new menu item. After triggering this event, the application remains at state “New Item”, which matches the current model.
5. Still at state “New Item”, the algorithm again considers three sequences: $E4$, $E3 - E4$, and $E4 - E4$. Suppose, it now picks event $E4$, which leads to state “Admin”. The triggered event matches the target event and the actual destination state of the application matches the destination state of the target event. That is, the algorithm has reached its goal and returns (line 11).

4.3.4 Dealing with Imprecise Models

The above example illustrates how Algorithm 3 gets closer to the target event even though the model is imprecise. Due to overapproximation, the shortest event sequence to the target event in the model may not be feasible in the application. For example, the shortest sequence from $S1$ that ends with $E4$ does not include $E3$. However, the application forces users to enter a name for a menu item ($E3$) before saving the menu item ($E4$). To allow the algorithm to find a feasible event sequence, which may or may not be the shortest according to the model, it gathers all short event sequences to the target and randomly picks a first event from these sequences. Another problem related to imprecise models is that, due to underapproximation, the model may not contain any sequence that leads from the current state to the target event. To avoid getting stuck, the algorithm stops searching for the target event after a specified number $maxEvents$ of events and returns with the success flag set to $false$ (line 6).

4.4 Summary of Responsiveness Problems

After systematically exploring each potential slowdown pair, the final step of EventBreak is to summarize the information about slowdown pairs and to present it to the user. Algorithm 2 returns a map G from slowdown pairs to cost vectors. The final step of EventBreak creates a report for each slowdown pair that contains the following information:

- The two events e_{cause} and e_{effect} of the slowdown pair.
- A cost plot that shows the cost of e_{effect} as a function of the number of times that e_{cause} and e_{effect} have been alternated.

The plot allows developers to quickly assess the importance of a slowdown pair based on the absolute cost values and based on the trend of these values. EventBreak can be combined with existing work on measuring empirical complexity that fits the observed cost values to a model that predicts its further development [12].

For our running example, EventBreak creates a plot that shows that the cost of executing event E_6 linearly increased with the number of executions of event E_4.

5. Implementation

The test generator is implemented as a Firefox add-on. The add-on obtains the list of all currently enabled JavaScript events through the browser’s event listener service. To handle non-JavaScript events, such as scrolling, filling out forms, and clicking on links, the add-on adds dummy handlers to the web page before querying the event listener service, essentially turning a non-JavaScript event into a JavaScript event. For example, the add-on attaches to each form field a dummy handler that modifies the form field when being triggered. For events that require input data from the user, such as as filling text fields of a form, we use random input data.

To instrument JavaScript code (Section 4.1), we parse it with Esprima[4], modify the resulting AST, and generate instrumented code with Escodegen.[5] We modify Spidermonkey (snapshot 151486), the JavaScript engine of Firefox, to intercept JavaScript code before it is executed. The modified browser intercepts all JavaScript code of web pages, instruments it, and executes only instrumented code. By implementing the instrumentation in the browser, we ensure to instrument all code, including code that is dynamically generated and loaded via `eval()`. In total, modifying Spidermonkey requires to change 261 lines of C++ code.

For inferring potential slowdown candidates (Algorithm 1), we use $conf_{min} = 0.8$ and $supp_{min} = 3$. For targeted exploration (Algorithms 2 and 3), we use $maxEvents = 200$ and $maxCosts = 20$.

6. Evaluation

The evaluation is driven by the following questions:

- *Does EventBreak detect slowdown pairs in complex web applications?* In total, the approach detects and confirms six unique slowdown pairs in three programs. Two pairs correspond to known bugs and the other four pairs are previously unknown responsiveness problems.

[4] `http://esprima.org/`

[5] `https://github.com/Constellation/escodegen`

- *Does searching slowdown pairs provide insights about the responsiveness of an application?* The detected slowdown pairs include a confirmed bug that makes the application unresponsive, and a scalability problem that crashes the application and makes it unusable. The cost plots produced by EventBreak help understand the performance of the analyzed event handlers.
- *How effective is the targeted exploration in alternately triggering the two events of a potential slowdown pair?* EventBreak successfully triggers 3,166 out of 3,563 target events that it tries to trigger (89%). On average, it takes 33 other events to reach a state where the target event can be triggered.
- *How much does the initial execution (first phase) influence the ability to detect slowdown pairs?* The overall effectiveness of EventBreak depends on the potential slowdown pairs that are exposed in the initial execution.

6.1 Experimental Setup

We use three real-world web applications for our experiments: (i) Joomla 3.2.1.0, which loads up to 1,678 JavaScript files/scripts comprising 9,364kB; (ii) Drupal 8.0.alpha7-0, which loads up to 296 JavaScript files/scripts comprising 3,104kB; and (iii) Tizen Todo List, which loads up to 21 JavaScript files/scripts comprising 672kB. Joomla and Drupal are popular content management systems that are representative of large web applications with a complex client-side component. Tizen Todo List is a task management application developed by others as an example of building applications for the Tizen platform. It is representative for mid-size, JavaScript-focused web applications.

We apply EventBreak to each application in the two usage scenarios described in Section 1. These scenarios differ in how to obtain the event-cost history during the first phase of EventBreak. On the one hand, we use random test generation to explore applications in a fully automatic way. On the other hand, we manually exercise each application by triggering events that might have interesting performance properties. For example, we trigger events that accumulate data and events that are likely to process these data. Each manual testing scenario takes less than five minutes of manual effort. We manually tested the applications without knowing their implementations. Table 1 lists the usage scenarios of the evaluation, along with the number of events they include:

- Scenario 1: We manually test Joomla to reproduce the bug described in Section 2. Therefore, we create several menu items and save the menu several times.
- Scenario 2: We manually test Joomla by adding categories (to be used for grouping articles), by listing all categories, and by sorting the categories according to different criteria, such as their title and their ID.
- Scenario 3: We manually test Drupal by creating several articles and by listing all articles.
- Scenario 4: We manually test Todo List by adding several tasks to the todo list of the current day.
- Scenarios 5 to 8: EventBreak automatically explores different components of Joomla, Drupal, and Todo List. For Joomla and Drupal, we limit the search space to subsets of the entire application.

6.2 First Phase

The right-most of the columns for the first phase in Table 1 shows how many potential slowdown pairs the approach infers. EventBreak explores all these pairs in the second phase of the approach.

6.3 Confirmed Slowdown Pairs

In total, the second phase of EventBreak detects and confirms six unique slowdown pairs, some of which are detected by more than one usage scenario (last column of Table 1). In the following, we describe several slowdown pairs and what they reveal about the responsiveness of the analyzed web application.

6.3.1 Database Connection Error in Drupal

From usage scenario 3, EventBreak infers the following potential slowdown pair:

- e_{cause} is to click the “Save and publish” button that finalizes the process of adding a new article.
- e_{effect} is to click the “Content” link in the menu of the application, which leads to a page that lists all articles.

During the targeted exploration phase, EventBreak repeatedly triggers these two events because the cost of e_{effect} is gradually increasing. After about 20 articles have been added, the application crashes and shows an error message in the browser. The reason is that a query to the MySQL database results in a reply that exceeds the maximum packet size of the database. We use both Drupal and MySQL in the default configuration shipped as part of the popular Bitnami LAMP stack. Unfortunately, this default configuration makes large parts of Drupal unusable because it crashes whenever the user tries to list all articles. The problem was reported by others and has been recognized by the developers.[6]

Figure 4 shows the cost plot produced by EventBreak. The cost of handling e_{effect}, which loads the “Content” page, increases gradually and suddenly drops when Drupal crashes.

6.3.2 Bounded Cost in Drupal

We fix the problem described in Section 6.3.1 by increasing the maximum packet size of the MySQL database, and we re-run the targeted exploration phase of EventBreak. The approach again explores the potential slowdown pair from Section 6.3.1 and adds more and more articles because the

[6] Drupal Issue 121390

ID	Program	First phase					Second phase			
		Driver	Events	Model		Potential	Events	Target events		Confirmed
			(unique)	States	Trans.	slowd. pairs		All	Reached	slowd. pairs
1	Joomla (menus, modules)	Manual	69 (21)	9	21	24	7,710	245	96%	2
2	Joomla (categories)	Manual	19 (8)	3	8	3	174	15	100%	1
3	Drupal (articles)	Manual	26 (7)	5	7	4	925	82	96%	1
4	Todo List	Manual	30 (5)	2	6	4	1,251	318	100%	2
5	Joomla (menus, modules)	Random	5,000 (2,350)	141	2,350	358	44,120	1,169	88%	1
6	Joomla (categories)	Random	5,000 (822)	20	823	405	44,181	1,099	86%	0
7	Drupal (articles)	Random	5,000 (511)	20	511	141	16,715	364	78%	0
8	Todo List	Random	1,000 (69)	6	69	56	5,010	271	99%	1
Total							120,086	3,563	89%	

Table 1. Usage scenarios for the evaluation. For the first phase, we give the number of (unique) events triggered, the size of the approximate model in terms of states and transitions, and the number of inferred potential slowdown pairs. For the second phase, we give the number of events triggered, how many target events EventBreak tries to reach, how many of them (%) it reaches, and the number of confirmed slowdown pairs.

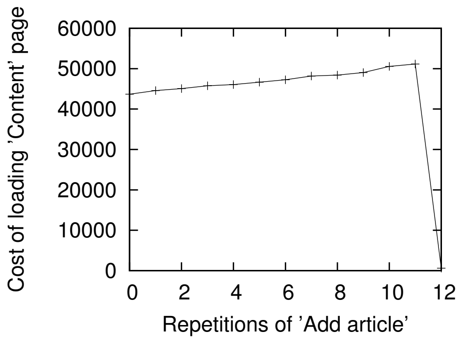


Figure 4. Cost plot for Drupal, which drops when Drupal crashes because it exceeds the maximum packet size of MySQL.

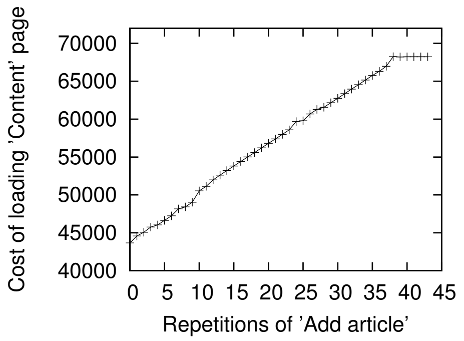


Figure 5. Cost plot for Drupal. The cost stabilizes when the total number of articles reaches 50 because the developers successfully bound the maximum cost of the event handler.

cost of listing all articles increases gradually: Each new article leads to the evaluation of 3,533 additional conditionals when listing all articles. After 50 articles have been added, the cost of listing all articles remains constant despite adding more articles. The cost plot in Figure 5 illustrates this behavior. The reason why the cost stabilizes is that the Drupal developers effectively bound the maximum cost of e_{effect}: Drupal list at most 50 articles per page and the user must visit another page to see more articles. The example shows that EventBreak is useful for confirming that upper bounds for the cost of event handlers work as expected. For example, this may be useful if a developer has fixed a responsiveness bug and wants to validate that the fix works.

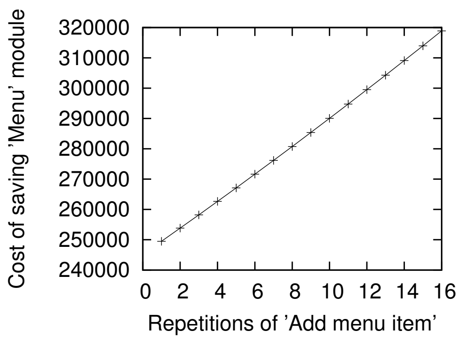


Figure 6. Cost plot for Joomla Issue 30274.

6.3.3 Responsiveness Bug in Joomla (Issue 30274)

For both usage scenarios 1 and 5, EventBreak infers the following potential slowdown pair, which corresponds to the bug of the motivating example (Section 2):

- e_{cause} is to click the "Save & Close" button that finalizes the process of adding a new menu item.
- e_{effect} is to click the "Save & Close" button that stores the properties of the menu.

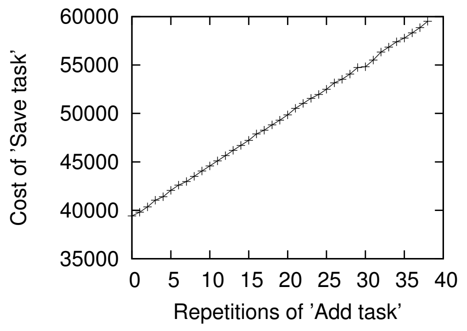


Figure 7. Cost plot for Todo List. It reveals that the cost of saving a new task grows in an unbounded way.

EventBreak can reproduce this known bug both based on manual testing and in a fully automatic manner. Figure 6 shows the cost plot, which illustrates that the cost of e_{effect} grows linearly with respect to the number of repetitions of e_{cause}.

6.3.4 Unbounded Cost of Sorting Categories in Joomla

For scenario 2, EventBreak infers the following potential slowdown pair:

- e_{cause} is to click the "Save as Copy" button that duplicates an existing category.
- e_{effect} is to click the column head "Title" of the table that lists all categories. This event causes Joomla to sort all categories by their title.

When exploring this potential slowdown pair, EventBreak finds that the cost of e_{effect} increases as a result of triggering e_{cause}. The reason is that Joomla implements the sorting of categories in a single event handler that, independently of the number of categories, sorts all existing categories. As a result, sorting categories may lead to a responsiveness problem for users that have many categories. This problem has been previously unknown.

6.3.5 Unbounded Growth in Todo List

From scenarios 4 and 8, EventBreak infers the following potential slowdown pair:

- e_{cause} is to click the "Add task" button, which adds a new task to a todo list.
- e_{effect} is to click the "Save task" button, which stores a new task to local storage.

When exploring this pair, EventBreak confirms that it is indeed a slowdown pair and produces the cost plot in Figure 7.

The slowdown pair reveals several previously unknown responsiveness problems in the design of Todo List, which affect the handler executed when adding a new task. First, the handler iterates through all tasks for the same day to group tasks by when they are due. Second, the handler creates DOM elements for each task of the currently displayed day whenever a new task has been added, including for tasks that become visible only when scrolling. Third, the handler iterates through all tasks in the entire database of tasks to check if the new task has been moved from another day. When adding a new task, this check is not only unnecessarily executed, but it couples the cost of adding a new task to the total number of stored tasks.

All three issues cause the cost of saving a new task to be unbounded because the cost grows whenever a new task is added for the currently displayed day. For users of Todo List who have only a small number of tasks per day, the first two issues may not be relevant in practice. However, the third issue severely limits the application's scalability to a large number of tasks because it causes the cost of saving a task to grow depending on the total number of stored tasks. To fix the problem, the code should avoid comparing a newly added task to all existing tasks.

6.3.6 Other Confirmed Slowdown Pairs

In addition to the slowdown pairs described above, EventBreak finds two more pairs that lead to an unbounded, linearly increasing execution time of event handlers. First, the approach finds another slowdown pair in Joomla: Adding a menu items increases the cost of loading a page that lists all menu items. Second, the approach finds another slowdown pair in Todo List: Saving a new task increases the cost of starting to add a new task. The second pair is the inverse of the pair in Section 6.3.5. The problem is that Todo List queries the current DOM via JQuery when constructing the UI for editing a new task. Since the DOM size depends on the number of existing tasks, this query gradually becomes more expensive when adding new tasks. Both problems may make the respective application unresponsive and have not been previously reported.

6.4 Unconfirmed Slowdown Pairs

EventBreak rejects most potential slowdown pairs because they are not confirmed by the targeted exploration. We inspect a sample of such unconfirmed slowdown pairs and find two main reasons why pairs are not confirmed as slowdown pairs. First, an event e_1 may coincidentally occur between two occurrences of another event e_2, which has an increasing cost. For example, when exploring the potential slowdown pairs inferred from scenarios 1 and 5, EventBreak considers several pairs similar to the pair in Section 6.3.3. Saving a new menu item requires several other events to happen before, such as setting a name for the menu item. Because these other events always occur before saving the menu item, Algorithm 1 finds them as potential cause for the increased cost of saving the menu. However, Algorithm 2 rejects these potential slowdown pairs because they are not the root cause of the problem. Second, some events have a varying execution time, for example, depending on the input written into a text field of a form. Since the root cause for these variations is not another event, the targeted exploration rejects any pairs related to such events.

The inspection of unconfirmed slowdown pairs shows that the targeted exploration phase of EventBreak is a vital part of the approach. It effectively focuses the user's attention on events that indeed have a gradually increasing cost and finds the event that causes this increase.

6.5 Effectiveness of Targeted Exploration

To assess the effectiveness of Algorithms 2 and 3 in exploring potential slowdown pairs, we measure how many events EventBreak triggers during the second phase of the approach, how many target events it succeeds to trigger, and for how many target events it gives up because Algorithm 3 exceeds the maximum number of triggered events. Table 1 shows the results for each usage scenario. In total, EventBreak triggers 120,086 events and thereby reaches 3,166 target events. On average, the approach requires 33 events to reach a state at which it can trigger the target event. Out of 3,563 target events that EventBreak tries to trigger, it fails to reach 397 (11%). For example, this may happen because reaching the target event requires that a particular sequence of events happens beforehand, which Algorithm 3 fails to find within the given maximum number of triggered events. We conclude that EventBreak is effective in triggering particular target events.

6.6 Influence of Usage Scenarios for Initial Execution

EventBreak amplifies problems exposed by an initial execution and therefore can detect responsiveness problems only if the involved events occur multiple times in the initial execution. The results from different usage scenarios (Table 1) illustrate the influence of the initial execution. Four of the six detected problems are detected based on an initial manual execution (scenarios 1 to 4) but not based on an initial random exploration (scenarios 4 to 8). By inspecting the event-cost histories of different scenarios, we find two reasons. For three of the four problems missed with initial random exploration, the event-cost history exposes the slowdown pair but does not expose it clearly enough for Algorithm 1 to infer the problematic pair. For example, the slowdown pair that exposes the bug in Drupal (Section 6.3.1) is inferred but discarded because the confidence of 0.67 is below our threshold of $conf_{min} = 0.8$ and because the support of 2 is below our threshold of $supp_{min} = 3$. For one of the four problems missed with initial random exploration (Section 6.3.4), the event-cost history contains one of the involved events exactly once, which is not enough to identify a potential slowdown pair.

7. Discussion and Future Work

Driver for First Phase Our approach is orthogonal to the usage scenario that drives the first phase of the approach. The two scenarios used in our evaluation, short manual tests done without knowing the applications' implementations and a simple form of random testing, provide a lower bound for EventBreak's effectiveness. As an alternative to the scenarios we evaluate here, EventBreak can build on other drivers for the first phase. First, EventBreak can build upon more sophisticated automated test generation approaches [6, 34], which will explore an application more effectively than our random exploration. Second, EventBreak can leverage an existing suite of correctness tests and turn them into responsiveness tests by amplifying the initial execution. Third, EventBreak can be combined with a mechanism to record execution traces from real users. In this scenario, the approach can reveal responsiveness problems that may not yet have become apparent because the users did not repeat the involved events often enough, and help avoiding them in the future. Based on our initial experience with EventBreak, we expect that its overall effectiveness will improve when using more sophisticated drivers for the first phase.

State Abstraction Our definition of events (Section 3.1) and the inferred application model (Section 4.3.1) rely on a state abstraction function that transforms a concrete application state into a more abstract representation. The effectiveness of EventBreak depends on the fact that an event that takes longer when being repeatedly triggered is represented as the same abstract event each time it is triggered. If, instead, the state abstraction function created a new state each time the event is triggered, then our approach could not find the responsiveness problem because there would not be an event with increasing execution time. Currently, EventBreak uses a simple state abstraction function that represents the web application's state based on the document title and the URL of the current page. This abstraction is sufficient for the applications used in the evaluation but may be insufficient for other applications. Future work will explore techniques to automatically identify a state abstraction function for a particular application, so that abstract states are precise enough to allow for navigating the application, yet generic enough to yield a bounded number of states.

8. Related Work

Various testing techniques infer a model of a user interface application to create tests that increase coverage [6, 9, 24–27, 34]. Our work is orthogonal to existing model learning techniques and contributes by leveraging a model to guide testing efforts towards potential responsiveness problems. The problems found by EventBreak are likely to be missed by approaches focused on coverage because triggering these problems requires the repeated execution of particular event sequences. Brooks and Memon use a finite-state model to generate tests that represent realistic usage scenarios [4]. Jensen et al. [16] leverage a model, along with symbolic summaries of event handlers, to create event sequences that reach a particular target statement. Our work differs by considering another kind of target (handlers that are likely to be part of a responsiveness problem) and by using an inferred and therefore approximate model. Other approaches to test user interface applications focus on infrequently selected widgets [23], or perform concolic execution, where input data of events are handled as symbolic variables [2, 10, 33] Azim and Neamtiu leverage a static taint-style analysis to generate tests for event-based applications [3]. All these approaches do not focus on performance problems.

Static [14] and dynamic [12, 42] analyses for finding the computational complexity of a function is related to the cost plots produced by EventBreak. These approaches analyze the computational complexity of individual functions, whereas we consider event sequences. Xiao et al. [36] propose a multi-execution profiling approach to identify workload-dependent performance bottlenecks, for example, in methods that block the UI thread. In contrast to EventBreak, their approach requires users to provide workloads, and it focuses on problems caused by passing large amounts of input data to an application. Wise [5] uses symbolic test generation to find inputs that trigger the worst-case complexity of a program. Algorithmic denial of service [7] is a class of attacks that attempts to reduce the performance of a server by repeatedly sending data that exposes bad performance. SpeedGun [31] is a technique for automated performance regression testing of thread-safe classes. Our work shares the idea of generating input to trigger performance problems. EventBreak differs from [5], [7], and [31] by generating sequences of events instead of input data. Grechanik et al. [13] describe a strategy for selecting test cases that may expose performance problems, assuming to have a large set of existing test inputs.

Jin et al. [17] and Liu et al. [22] describe static code checkers that search for occurrences of common performance-related antipatterns. Several profiling approaches find excessive memory usage and unnecessary computation, for example, repeated executions of similar behavior in loops [30], underutilized or overutilized containers [37], unnecessarily copied data [38], and objects where the cost to create them exceeds the benefit from using them [39]. Yan et al. use reference propagation profiling to detect common patterns of excessive memory usage [40]. These analyses focus on particular root causes of suboptimal performance. Instead, our approach analyzes event handlers as a black box and finds handlers with noteworthy responsiveness properties independent of a specific root cause.

StackMine [15] mines a stream of stack trace snapshots of a multi-threaded application to detect performance problems. Wert et al. [35] propose a framework to systematically search for occurrences of performance antipatterns in distributed systems. One of their strategies, called "Direct Growth", identifies growing response times over the measurement time of a system under test. In contrast to both [35] and [15], our approach systematically generates input to assess whether a potential performance problem exists. Furthermore, our approach identifies the event that causes the execution time of another event to increase.

Adamoli et al. [1] evaluate approaches for automating performance tests of GUI applications by capturing and replaying interactive executions. Our approach is complementary to capture and replay tools because it creates input sequences. Khoo et al. propose a programming model to ease the task of writing responsive event-driven applications by splitting long-running computations into asynchronous callbacks [19].

Killian et al. [20] propose to find performance problems in event-driven, distributed systems by searching anomalies in how long an event takes to be processed. Our work also searches for abnormal performance characteristics but focuses on events that have a gradually increasing, potentially unbounded execution time. AppInsight [32] instruments mobile application binaries to identify performance-critical paths after deploying the application. In contrast, EventBreak can analyze applications before deploying them to users. Jovic et al. [18] propose a profiler that identifies perceptible performance problems in GUI applications. Their approach reports a sorted list of methods that may cause latency, whereas our approach confirms each potential problem through systematic event generation. Mi et al. [28] propose a runtime analysis for distributed systems that explains why two similar executions expose different performance characteristics. EventBreak differs from [28] by systematically creating tests for this purpose instead of relying on existing executions.

A recent study [22] shows that event-driven applications often suffer from performance bugs. It reports that many real-world bugs manifest only when particular user interactions occur and that many bugs cause the application to become unresponsive. Our work addresses these problems. Other studies confirm that performance bugs are a common problem [17] and that developers face problems in reproducing performance problems reported by users [41].

9. Conclusions

Developers of event-driven program currently rely on manual testing and profiling to find performance and scalability problems. This paper presents EventBreak, an approach for analyzing the responsiveness of web applications through performance-guided test generation. The approach identifies and explores pairs of input events that, if triggered alternately, take longer and longer to process. Since processing an event blocks the single thread that executes JavaScript code in the browser, events should always terminate quickly to avoid making the application unresponsive. EventBreak allows developers to find events that have a potentially unbounded increase of their execution time or to verify that the execution time of an event is bounded. We apply the approach to complex web applications and show that it finds previously known and previously unknown bugs, and that it provides insights about the performance of particular events.

Our implementation and all data required to reproduce our results are available at:
`https://github.com/michaelpradel/WebAppWalker`

10. Acknowledgments

The work presented in this paper is supported by NSF grants CCF-1017810, CCF-0747390, CCF-1018729, CCF-1423645, and CCF-1018730, and gifts from Samsung and Mozilla.

References

[1] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth. Automated GUI performance testing. *Softw Qual J*, pages 801–839, 2011.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE*, page 59, 2012.

[3] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660, 2013.

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

[5] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.

[6] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA*, 2013.

[7] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *SSYM*, 2003.

[8] A. B. de Oliveira, S. Fischmeister, A. Diwan, M. Hauswirth, and P. F. Sweeney. Why you should care about quantile regression. In *ASPLOS*, pages 207–218, 2013.

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

[10] S. R. Ganov, C. Killmar, S. Khurshid, and D. E. Perry. Test generation for graphical user interfaces based on symbolic execution. In *AST*, pages 33–40, 2008.

[11] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, 2007.

[12] S. Goldsmith, A. Aiken, and D. S. Wilkerson. Measuring empirical computational complexity. In *ESEC/FSE*, pages 395–404, 2007.

[13] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE*, pages 156–166, 2012.

[14] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[15] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *ICSE*, pages 145–155, 2012.

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

[17] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *PLDI*, pages 77–88, 2012.

[18] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *OOPSLA*, pages 155–170, 2011.

[19] Y. P. Khoo, M. Hicks, J. S. Foster, and V. Sazawal. Directing javascript with arrows. In *DLS*, pages 49–58, 2009.

[20] C. E. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *FSE*, pages 17–26, 2010.

[21] B. Lambeau, C. Damas, and P. Dupont. State-merging dfa induction algorithms with mandatory merge constraints. In *ICGI*, pages 139–153, 2008.

[22] Y. Liu, C. Xu, and S. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, 2014.

[23] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for android apps. In *ESEC/FSE*, pages 224–234, 2013.

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

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

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

[27] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE*, pages 122–134, 2008.

[28] N. Mi, L. Cherkasova, K. M. Ozonat, J. Symons, and E. Smirni. Analysis of application performance and its change via representative application signatures. In *NOMS*, pages 216–223, 2008.

[29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS*, pages 265–276, 2009.

[30] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *ICSE*, pages 562–571, 2013.

[31] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *ISSTA*, pages 13–25, 2014.

[32] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: mobile app performance monitoring in the wild. In *OSDI*, pages 107–120, 2012.

[33] 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.

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

[35] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *ICSE*, pages 552–561, 2013.

[36] X. Xiao, S. Han, D. Zhang, and T. Xie. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *ISSTA*, pages 90–100, 2013.

[37] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

[38] G. H. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

[39] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[40] D. Yan, G. H. Xu, and A. Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *ICSE*, pages 134–144, 2012.

[41] S. Zaman, B. Adams, and A. E. Hassan. A qualitative study on performance bugs. In *MSR*, pages 199–208, 2012.

[42] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *PLDI*, pages 67–76, 2012.