Front. Comput. Sci. China, 2008, 2(3)
DOI 10.1007/s11704-009-0000-0

RESEARCH ARTICLE

Model-based Automated Testing of JavaScript Web Applications via Longer Test Sequences

Pengfei GAO[1], **Yongjie XU**[1], **Fu SONG** (✉)[1], **Taolue CHEN**[2]

1 School of Information Science and Technology, ShanghaiTech University, Shanghai 201210, China
2 Department of Computer Science, University of Surrey, Guildford GU2 7XH, UK

Abstract JavaScript has become one of the most widely used languages for Web development. Its dynamic and event-driven features make it challenging to ensure the correctness of Web applications written in JavaScript. A variety of dynamic analysis techniques have been proposed which are, however, limited in either coverage or scalability. In this paper, we propose a simple, yet effective, model-based automated testing approach to achieve a high code-coverage within the time budget via testing with longer event sequences. We implement our approach as an open-source tool LJS, and perform extensive experiments on 21 publicly available benchmarks. On average, LJS is able to achieve 86.5% line coverage in 10 minutes. Compared with JSDep, a state-of-the-art breadth-first search based automated testing tool enriched with partial order reduction, the coverage of LJS is 11-19% higher than that of JSDep on real-world large web applications. Our empirical findings support that proper longer test sequences can achieve a higher code coverage in JavaScript Web Application testing.

Keywords Model-based Testing, Automated Testing, JavaScript Web Applications

1 Introduction

JavaScript is a highly dynamic programming language with first-class functions and "no crash" philosophy, which allows developers to write code without type annotations, and to generate and load code at runtime. Partially because of these programming flexibilities, Web applications based on JavaScript are gaining increasing popularity. These features are, however, double-edged sword, making Web applications prone to errors and intractable to static analysis.

Dynamic analysis has proven to be an effective way to test JavaScript Web applications [1–10]. Since it requires testcases to explore the state space of the application, various approaches for automated testcase generation have been developed in literature, which can generate event sequences and/or associated input data of events. The event sequences concern the order in which event handlers are executed (e.g., the order of clicking buttons), while the input data concerns the choice of values (e.g., strings, numbers and objects). Generation of both event sequences and input data is important to achieve a high code coverage, and has been extensively studied.

In general, event sequences are generated by randomly selecting event handlers with heuristic search strategies [1–3, 7]. These approaches are able to analyze large real-world applications, but are usually left with a low code coverage. One possible reason, as mentioned in [6, 11], is that the event sequences are insufficiently long to explore parts of the code which may trigger the error. For example, in an experiment of [6], the uncovered code of the benchmark *tetris* is mainly due to the function *gameOver* which will only be invoked after a long event sequence. For traditional white-box testing and GUI testing, it has been shown that increasing the length of test sequences could improve coverage and fail-

Received month dd.yyyy; accepted month dd.yyyy

E-mail: songfu@shanghaitech.edu.cn

ure detection [12–15]. However, this has not been fully exploited in testing JavaScript Web applications. One of the reasons is that existing approaches usually generate event sequences from scratch by iteratively appending events to the constructed sequences up to a maximum bound, and the number of event sequences may blow up exponentially in terms of this bound. Therefore, for efficiency consideration, the maximum bound often has to be small (for instance, less than 6 [6]). To mitigate these issues, pruning techniques (e.g. mutation testing [5, 8] and partial order reduction [11]) were proposed to remove redundant event sequences, which allow to increase the length of sequences in a reasonable time. On the other hand, the input data is generated by either randomly choosing values with lightweight heuristic strategies [2, 3], or using heavyweight techniques (e.g., symbolic/concolic execution) [1, 4, 6, 9]. These works either consider unit testing or usually simply reuse the aforementioned methods to generate event sequences.

In this work, we focus on the issue of event sequence generation. In particular, we propose a novel model-based automated testing approach to achieve a high code coverage in a reasonable time by generating and executing long event sequences. Our approach mainly consists of two key components: the model constructor and the event sequence generator. The model constructor iteratively queries an execution engine to generate a finite-state machine (FSM) model. It explores the state space using long event sequences in a way to avoid prefix event subsequences re-executing and backtracking. To improve the scalability, we propose a state abstraction approach, as well as a weighted event selection strategy, to construct small-sized FSM models. The event sequence generator creates long event sequences by randomly traversing the FSM model from the initial state. We implement our approach in a tool **L**onger **J**ava**S**cript (LJS). To compare with other methods, we also implemented an event sequence generator from JSDep [11] based on the FSM model. One of the distinguished features of our approach is its simplicity; it turns out that one can achieve a higher testing coverage for JavaScript Web applications by adopting a simple test strategy which is easy to implement.

We demonstrate the efficiency and effectiveness of our tool on 21 publicly available benchmarks taken from JSDep [11], which includes 17 real-world Web applications, ranging from hundreds to thousands of lines of code. On average, our approach is able to achieve 86.5% line coverage in 10 minutes. On large applications, the coverage of LJS is 11-19% higher than that of JSDep. We find that proper long event sequences can indeed improve the coverage with respect to the application under test, and we provide concrete, empirically validated approaches to generate long event sequences.

In summary, the main contributions of this paper are

- We propose the first method to construct finite-state machine models to represent the behaviors of JavaScript Web applications, taking both the previously executed events and DOM event dependency into account;
- We present a new automated testing approach for generating longer event sequences of client-side JavaScript Web applications by leveraging the proposed finite-state machine models;
- We implement these new methods in an open-sourced tool LJS (on GitHub) and evaluate them on a large set of JavaScript Web applications to demonstrate the efficiency and effectiveness.

Structure. The remainder of the paper is organized as follows. In Section 2, we introduce basic notations and a running example to motivate our work. In Section 3, we first present an overview of our approach and then elaborate the details of the FSM model construction and testcase generation. We report the experimental results and compare the performance of our tool LJS with that of JSDep in Section 4. We review the related work in Section 5. We finally conclude in Section 6.

2 Preliminaries and Running Example

In this section, we first briefly recap JavaScript Web applications, DOM event dependency and finite state machines. Then, we present a running example and discuss limitations of existing approaches.

2.1 JavaScript Web Application

Client-side Web applications consist of script files executed by Web browsers. When a browser loads a Web page, it parses the script files, represents them as a Document Object Model (DOM) tree, and then executes the top-level script code. Each node in the DOM tree represents an object on the Web page and may also be associated with a set of events. Each event may have some event handlers (e.g., callback functions such as *onload* and *onclick*) which are either statically registered inside the HTML file or dynamically registered by executing functions. When an event occurs (e.g., a button is clicked), the corresponding event handlers are executed sequentially. Although the browser ensures that each callback function is executed atomically, the execution of the entire Web application exhibits nondeterminism due to the interleaving of the executions of multiple callback functions.

2.2 DOM Event Dependency

Given a JavaScript Web application, let R_c and R_d respectively denote the *control* and *data* dependency relation over the functions of the application. For each pair of the functions (usually event handlers) m_1 and m_2, $(m_1, m_2) \in R_c$ (resp. $(m_1, m_2) \in R_d$) if there are two statements st_1 and st_2 in m_1 and m_2 respectively such that the execution of st_1 affects the control (resp. data) flow of st_2. Given two DOM events e_1 and e_2, e_2 is *dependent on* e_1, denoted by $e_1 \to e_2$, if one of the following conditions holds:

1. there are event handlers m_1 and m_2 of e_1 and e_2 respectively such that $(m_1, m_2) \in (R_c \cup R_d)^*$,
2. the execution of m_1 registers, removes, or modifies m_2.

Given two event sequences ρ_1 and ρ_2, ρ_1 and ρ_2 are *equivalent* if ρ_1 can be transformed from ρ_2 by repeatedly swapping adjacent and independent events of ρ_2. More details on control, data and DOM event dependencies can be found in Sung et al. [11].

2.3 Finite State Machine

A (nondeterministic) Finite State Machine (FSM) is a tuple

$$M = (S, I, \delta, s_0),$$

where S is a finite set of states with $s_0 \in S$ as the initial state, I is a finite input alphabet, $\delta \subseteq S \times I \times S$ is the transition relation. A transition $(s_1, e, s_2) \in \delta$ denotes that, after reading the input symbol e at the state s_1, the FSM M can move from the state s_1 to the state s_2. We denote by $\mathsf{supp}(s)$ the set $\{(e, s') \in I \times S \mid (s, e, s') \in \delta\}$. Given a word $e_1 \cdots e_n \in I^*$, a *run* of M on $e_1 \cdots e_n$ is a sequence of states $s_0 s_1 \cdots s_n$ such that for each $1 \leq i \leq n$, $(s_{i-1}, e_i, s_i) \in \delta$. We denote by ϵ the empty word. Note that, in this work, we will use an FSM to represent the behaviors of a JavaScript Web application, rather than to recognize regular languages. Because of this, final states are not included in the definition of FSMs.

2.4 Running Example

Consider the running example in Figure 1, where the HTML code defines DOM elements of three checkboxes and one button. The JavaScript code defines a global variable `count` and five functions manipulating the global variable `count` and DOM elements. Initially, the three checkboxes, named `A`, `B` and `C`, are unchecked; the button, named `Submit`, does not have any *onclick* event handler. The *onclick* event handlers of the three checkboxes are the functions `FA`, `FB` and `FC` respectively. For each `X`∈{`A`,`B`,`C`}, when the checkbox `X` is clicked (i.e., the corresponding event occurs), its state (checked/unchecked) is switched, and then the *onclick* event handler (i.e., the function `FX`) is executed. The function `FX` first determines whether the state of `X` is checked or not. If it is checked, the global variable `count` is increased by one; otherwise the global variable `count` is decreased by one. Finally, the function `FX` invokes the function `CheckedEnough` to verify whether the value of the global variable `count` is no less than 3, i.e., whether there are at least three checkboxes in the checked state. If `count`>=3, then the function `FSubmit` is registered to

```
<!DOCTYPE html>
<html>
<head>
    <p> Example with 3 checkboxes and 1 button </P>
</head>
<body>
<div id="checkboxes">
  <input id="A" type="checkbox" onclick="FA(this)"> A
  <input id="B" type="checkbox" onclick="FB(this)"> B
  <input id="C" type="checkbox" onclick="FC(this)"> C
</div>
<button id="Submit" type="button" > Submit </button>
<script>
  var count = 0;
  function FA(node) {
    if(node.checked == false) count = count - 1;
    else count = count + 1;
    CheckedEnough();
  }
  function FB(node) {
    if(node.checked == false) count = count - 1;
    else  count = count + 1;
    CheckedEnough();
  }
  function FC(node){
    if(node.checked == false) count = count - 1;
    else count = count + 1;
    CheckedEnough();
  }
  function CheckedEnough() {
    var  b = document.getElementById("Submit");
    if(count >= 3) b.onclick = FSubmit;
    else b.onclick = null;
  }
  function FSubmit() {
    alert("Submit successfully");
  }
</script>
</body>
</html>
```

Fig. 1 Example HTML page and associated JavaScript code

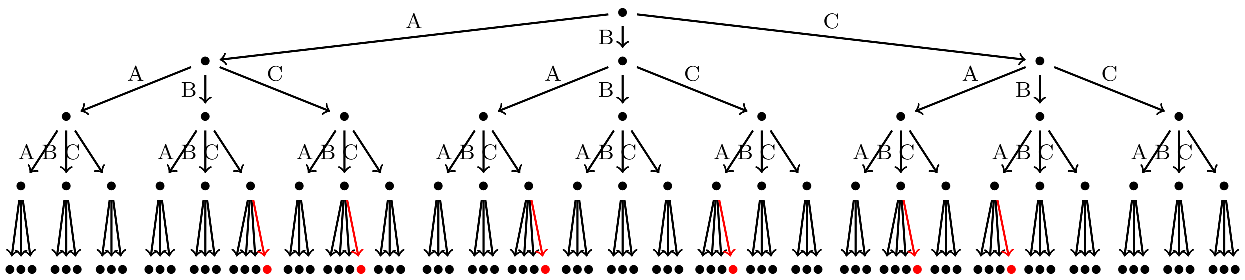

Fig. 2 The search tree of our running example, where the missed labels of edges in black color are respectively `A,B,C`, and the missed labels of edges in red color are `Submit`

the *onclick* event handler of the button `Submit`; otherwise (i.e., `count`<3), the *onclick* event handler of the button `Submit` is removed. If the button `Submit` is clicked when the function `FSubmit` serves the *onclick* event handler of `Submit`, the function `FSubmit` is executed and prints a message to the console.

In this example, for `X,Y`∈{`A,B,C`}, the *onclick* event of the checkbox `X` is dependent upon the *onclick* event of the checkbox `Y`, and the *onclick* event of the button `Submit` is dependent upon the *onclick* event of `X`.

2.5 Limitations of Existing Approaches

We now demonstrate why long event sequences are important for automated testing of JavaScript Web applications. Tools like Artemis [2] and JSDep [11] generate event sequences by systematically triggering various DOM events up to a fixed depth. After loading the Web page, these tools start by exploring all the available events at the initial state. If a new state is reached by executing a sequence of events, all the available events at the new state are appended to the end of the event sequence. The procedure is repeated until timeout or a fixed depth is reached. The search tree of our running example up to the depth four is depicted in Figure 2, where the unshown labels of the edges in black are respectively `A,B,C`, and those in red color are `Submit`. Each edge labeled by `X`∈{`A,B,C,Submit`} denotes the execution of the *onclick* event handler of `X`, and each node denotes a state. For each event sequence ρ of length three, if ρ is a permutation of `A;B;C`, then there are four available events `A,B,C,Submit` after ρ; otherwise there are three available events `A,B,C`.

A naïve algorithm (such as the default algorithm in Artemis) would inefficiently explore the event space, i.e., the full tree in Figure 2, and may generate $6 + \sum_{i=1}^{4} 3^i = 126$ event sequences. However, many of them are redundant. For instance, the sequences `A;B;C;Submit` and `B;A;C;Submit` actually address the same part of the code, hence one of them is unnecessary for testing purposes. To remedy this issue, JSDep implemented an approach based on partial-order reduction in Artemis which prunes redundant event sequences by leveraging DOM event dependencies.

To cover all code of the running example, each event handler of all the three checkboxes has to be executed at least two times (examining checked/unchecked states). Therefore, a sequence of length seven (e.g., `A;B;C;Submit;A;B;C`) is sufficient to fully cover the code. However, if one sets the depth bound of the test sequence to be seven for the full code coverage, the default search algorithm in Artemis and JSDep may explore at least $\sum_{i=1}^{7} 3^i = 3,279$ event sequences. Notice that both Artemis and JSDep may re-execute previously executed test sequences in order to explore the event space further, which is time-consuming. Partially because of this, within a time limit these tools often generate and execute short event sequences only. Similar to the classic program analysis, it is not hard to envisage that short event sequences would hamper testing coverage. Indeed, in our running example, covering the function `Submit` requires test sequences with length at least 4. Unfortunately, existing approaches suffer from the "test sequence explosion" problem when increasing the depth bound of the testing sequences. In this work, we propose a model-based, automated testing approach for JavaScript Web applications, aiming to generate long event sequences to improve the code coverage, but do so in a clever way to alleviate the issue of exponential blowup of event sequences.

3 Methodology

In this section, we first present an overview of our approach and then elaborate the details of our FSM model

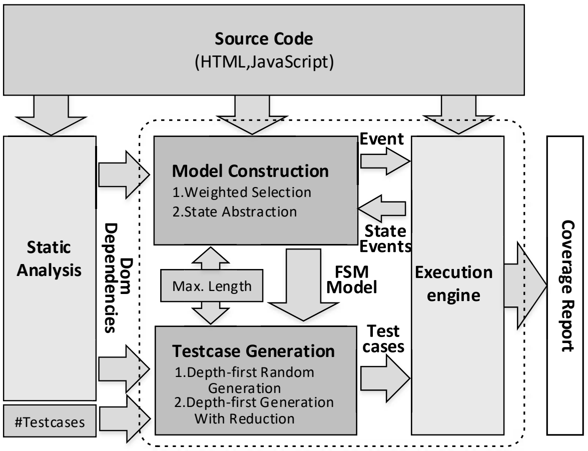

Fig. 3 Framework overview of LJS

construction and testcase generation procedures.

3.1 Overview of Our Approach

Figure 3 presents an overview of our approach, which consists of four components: static analysis, execution engine, model construction and testcase generation. Given the HTML/JavaScript source file(s) of a (client-side JavaScript Web) application, a length bound of event sequences (Max. Length), and a bound of the number of event sequences to be generated (#Testcases) as inputs, LJS outputs a (line) coverage report. Internally, LJS goes through the following steps. (1) LJS first computes the DOM event dependencies via static analysis. (2) Then, LJS constructs an FSM model of the application with a variant of depth-first search by leveraging an execution engine and the DOM event dependencies. The FSM model is used to generate long event sequences. (3) Finally, all the generated event sequences are iteratively executed on the execution engine and output the coverage report. In the sequel we elaborate these steps in more detail.

Static Analysis. Given the HTML/JavaScript source file(s) of a Web application, this component computes the DOM event dependencies. In our implementation, we leverage JSDep to compute the DOM dependency. It first constructs a control flow graph (CFG) of the JavaScript code, and then traverses the CFG and encodes the control and data flows (i.e., the dependency relations) in Datalog. The DOM event dependencies are finally computed via a Datalog inference engine. More details can be found in Sung et al. [11]. The analysis can handle dynamic registration, triggering and removal of event handlers, but not other dynamic features such as dynamic code injection and obfuscation. Therefore, our approach inherits both the advantage and disadvantage of JSDep.

Execution Engine. The execution engine is used for the FSM model construction and event sequence execution. It loads and parses the source file(s), and then executes the top-level JavaScript code. For the FSM model construction, the execution engine interacts with the model constructor by iteratively receiving input events and outputting an (abstract) successor state and a set of available events at the successor after executing the input event. For event sequence execution, the execution engine receives a set of event sequences, executes them one by one and finally outputs a coverage report. We implemented an execution engine by leveraging Artemis [2] with its features such as the event-driven execution model, interaction with DOM of web pages, and dynamic event handler detection.

Model Construction. The model constructor interacts with the execution engine by iteratively making queries to generate an FSM model up to the given length bound (i.e., Max. Length). The FSM model is intended to represent the behaviors of the application. A state of the FSM model denotes an (abstract) state of the application, and a transition (s, e, s') denotes that after executing the event handler e at the state s, the application enters the state s'. The model constructor starts with the FSM containing only one initial state, explores new state s' by selecting one event e available at the current state s, adds (s, e, s') to the FSM model, and continues exploring the state s'. To take previously executed events and DOM event dependencies into account during the FSM model construction, we propose a weighted event selection strategy where the selection of events is guided by their weights. When the length bound is reached, LJS allows to restart the exploration from the initial state and adds new states and transitions to the FSM model, in order to make the FSM model more complete.

Testcase Generation. The testcase generator traverses the FSM model to generate event sequences. LJS supports two event sequence generation algorithms: (1) partial-order reduction (POR) based event sequence generation, and (2) random event sequence generation. The first algorithm traverses the FSM model from the initial state and covers all the paths up to a (usually small) bound, while the redundant event sequences are pruned based on the POR from JSDep [11]. It usually generates many short event sequences and is regarded as the baseline algorithm. The second algorithm repeatedly and randomly traverses the FSM model from the

Algorithm 1 Model Construction

```
Input: An application $P$ and a Max. bound $d$
Output: An FSM model $M = (S, I, \delta, s_0)$
 1: $i := 0$;
 2: $I := \emptyset$;
 3: $\delta := \emptyset$;
 4: $cur :=$ GetInitPage($P$);
 5: $s_0 :=$ GetState($cur$);
 6: $S := \{s_0\}$;
 7: while $i < d$ do
 8:     $e :=$ GetEvent($cur$);
 9:     $suc :=$ GetNextPage($e$);
10:     $s' :=$ GetState($suc$);
11:     $\delta := \delta \cup \{(s_0, e, s')\}$;
12:     $I := I \cup \{e\}$;
13:     $S := S \cup \{s'\}$;
14:     $s_0 := s'$;
15:     $cur := suc$;
16:     $i := i + 1$;
17: end while
18: return $(S, I, \delta, s_0)$;
```

initial state to generate a small number of longer (up to a usually large bound) event sequences, and, as such, does not cover all possible paths, hence alleviates the event sequence explosion problem.

To give a first impression of the performance of LJS, we ran Artemis, JSDep and LJS on the running example introduced in Section 2.4. LJS reached 100% (line) coverage in 0.2s using one event sequence with length 7, while Artemis executed 3,209 event sequences in 10min and reached 86% coverage, and JSDep executed 64 event sequences in 0.7s and reached 100% coverage. We also ran JSDep and LJS on a variant of the running example which contains 10 checkboxes and where the condition `count` >= 3 is replaced by the condition `count` >= 6. LJS reached 100% coverage in 0.4s using one event sequence with length $(10 \times 2 + 1) = 21$, while JSDep executed 462 event sequences in 27.4s and reached 100% coverage. This suggests that a small number of longer event sequences could outperform a large number of shorter event sequences for applications with intensive DOM event dependency.

In the rest of this section, we will detail our FSM model and testcase generation procedures.

3.2 Model Construction

Algorithm 1 presents the model construction procedure, which takes an application P and a maximum bound d as the input, and outputs an FSM model $M = (S, I, \delta, s_0)$. After initializing the counter i, the input alphabet I and the set of transition relation δ (Lines 1–3), it first calls the procedure `GetInitPage` which loads and parses P, executes the top-level JavaScript code, and finally returns the initial Web page cur and the set of available events on this page which are dynamically detected (Line 4). The initial state of the FSM model s_0 is obtained by calling the procedure `GetState`(cur) (Line 5). Intuitively, the procedure `GetState` computes a state from the source code of the Web page cur (see below). After the initialization, Algorithm 1 iteratively selects and executes events to explore the state space up to d rounds. In each iteration (Lines 7–16), it selects one available event e on the current Web page cur by invoking the procedure `GetEvent` (Line 8), based on some event selection strategy (see below). Then, it calls the procedure `GetNextPage` to get the next Web page suc by executing e (Line 9). The state s' of the new Web page suc is also obtained by calling the procedure `GetState` (Line 10). Finally, the transition (s, e, s'), the event e and the state s' are added to the FSM M (Lines 11–13) respectively, and the variables cur and i are updated accordingly (Lines 15 and 16). As mentioned before, LJS allows to restart the exploration of the state space from the initial state, so this process may be repeated many times.

Overall, our model construction explores the state space in a depth-first fashion with a large length bound (Max. Length in Figure 3); we can avoid re-executing previously executed event sequences, and do not track state changes during execution.

3.2.1 State Abstraction

As mentioned before, the states of the FSM model are used to represent those of the application and are computed from the Web pages by invoking the procedure `GetState`. We note that state abstraction is crucial to describe the application states. On the one hand, the states of the FSM model should contain enough data to distinguish different application states, as unexplored application states maybe be wrongly skipped if their abstracted states were explored before, which may happen if a coarse-grained state abstraction is adopted. On the other hand, overly fine-grained state abstraction may generate states which are indistinguishable with respect to some coverage criteria, resulting in an explosive or even infinite state-space [16]. Therefore, the implementation of the procedure `GetState` requires a balance between the precision and the scalability.

In Artemis, the state abstraction is implemented by computing the hash value of the Web page, which in-

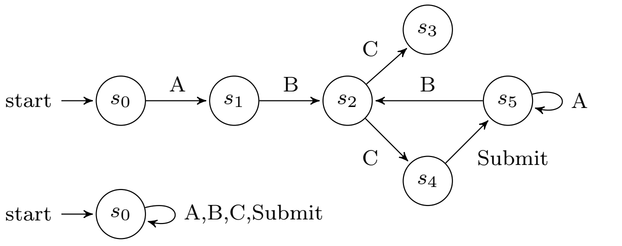

Fig. 4 The FSM models of the running example: the top FSM is constructed using the state abstraction from Artemis and the bottom FSM is constructed using our new state abstraction

cludes the Web page layout and the dynamically updated DOM trees, but not the concrete values of CSS properties or application variables, nor the server-side states (unless an initial server state is provided by the user). It was demonstrated that this state abstraction is effective and efficient [2].

However, in our experiments, we find it is still too fine-grained. Artemis typically assigns random values to the attributes of the DOM nodes which are taken into account in the state abstraction of Artemis. In our running example, Artemis assigns a random Boolean value to the *implicit* attribute `Value` of each checkbox, though the value of this attribute does not affect the code coverage. (Here, “implicit" means that the attribute is not explicitly given in the source code, but the object has such an attribute.) The FSM model constructed using the state abstraction from Artemis is depicted in Figure 4(top-part). To avoid a prohibitively large (sometimes even infinite) state-space and improve efficiency, we use a state abstraction based on the state abstraction of Artemis, but discard the random values of the implicit attributes in the DOM tree. This allows to focus on the state changes made by executing events rather than random value assignments. The FSM model constructed by our state abstraction is depicted in Figure 4(bottom-part), which is much smaller in size.

Experimental results have confirmed that our state abstraction approach significantly reduces the size of the FSM model with a comparable (or even better) line coverage (cf. Section 4.2).

3.2.2 Weighted Event Selection

It is common that several events are available on a Web page. Evidently, the selection of events will affect the quality of the FSM model. To take both the previously executed events and the DOM event dependency into account, we propose a weighted event selection strategy so that the selection of events is guided by their weights, where the weights capture the impacts of both the previously executed events and the DOM event dependencies. Technically, we focus on:

- **Frequency of event execution.** In principle, all the events should be given opportunities to execute. As a result, an event which has been executed would have a lower priority to be selected in the subsequent exploration.
- **DOM event dependency.** Some corner-case code may be explored only by some specific event sequences due to their dependency. To expose the corner code using long event sequences, the events that *depend upon the previous selected events* warrant a higher chance to be selected.

In the weighted event selection strategy, each event e is associated with a weight, which is adjusted dynamically at runtime. The weight of the event e is defined as follows:

$$\texttt{weight}(e) = \frac{\alpha_e \times x + \beta_e \times (1-x)}{N_e + 1}$$

where α_e and β_e are weight parameters, x is a Boolean flag determined by the DOM event dependency ($x = 1$ if e depends upon the previous selected event; $x = 0$ otherwise), and N_e is the number of the times that e has been executed.

With the weighted event selection strategy, the procedure `GetEvent`(cur) randomly returns one of the events which has the highest weight among all available events on the Web page cur. In our experiments, α_e and β_e are set to be 0.7 and 0.3 respectively, which are the best configuration after tuning.

Recall the running example in Figure 1. At the initial state, all the available *onclick* events of the checkboxes `A`, `B` and `C` have the same weights 0.3 (note that the *onclick* event of the button `Submit` is not available therein), i.e., they have the same chance to be selected. Suppose the *onclick* event of the checkbox `A` is selected, the weights of the *onclick* events of the checkboxes `A`, `B` and `C` are updated to $\frac{0.7}{2}$, $\frac{0.7}{1}$, $\frac{0.7}{1}$ respectively. LJS then randomly chooses one of the *onclick* events of the checkboxes `B` and `C`. Suppose the *onclick* event of the checkbox `B` is selected on this occasion. The weights of all the available *onclick* events of `A,B,C` become $\frac{0.7}{2}$, $\frac{0.7}{2}$, $\frac{0.7}{1}$, which imply that the *onclick* event of the checkbox `C` will be selected at the next step. After that, the weights of the *onclick* events `A,B,C,Submit` are updated to $\frac{0.7}{2}$, $\frac{0.7}{2}$, $\frac{0.7}{2}$, $\frac{0.7}{1}$ (note that the *onclick* event of the button `Submit` now becomes available).

Algorithm 2 Baseline Event Sequence Generation with POR

```
         An FSM $M = (S, I, \delta, s_0)$
Input:   A bound $d$ of the length of test sequences
         The DOM event dependency relation $\to$
Output: A set of test sequences $T$
 1: $T := \emptyset$;
 2: $ss :=$ NewStack();
 3: $ss$.Push($s_0$);
 4: EXPLORE($ss$);
 5: return $T$;
 6: procedure EXPLORE(Stack : $ss$)
 7:   $s := ss$.Top();
 8:   $s$.SelectedEvent := null;
 9:   if $ss$.Length() $\leq d$ then
10:     $s$.done := $\emptyset$;
11:     $s$.sleep := $\emptyset$;
12:     $E := \{e \in I \mid \exists s.(e, s') \in$ supp($s$)$\}$;
13:     while $\exists e \in E \setminus$ ($s$.done $\cup$ $s$.sleep) do
14:       $s$.done := $s$.done $\cup \{e\}$;
15:       $s$.SelectedEvent := $e$;
16:       for all $s' \in \{s' \in S \mid (e, s') \in$ supp($s$)$\}$ do
17:         $s'$.sleep := $\{e' \in s$.sleep $\mid e \not\to e' \wedge e' \not\to e\}$;
18:         $ss$.push($s'$);
19:         EXPLORE($ss$);
20:         $s$.sleep := $s$.sleep $\cup \{e\}$;
21:       end for
22:     end while
23:   end if
24:   if $s$.SelectedEvent = null then
25:     $\rho := \epsilon$;
26:     for all $s' \in ss$ from bottom to top do
27:       $\rho := \rho \cdot s'$.SelectedEvent;
28:     end for
29:     $T := T \cup \{\rho\}$;
30:   end if
31:   $ss$.Pop();
32: end procedure
```

Algorithm 3 Long Event Sequence Generation

```
         An FSM $M = (S, I, \delta, s_0)$
Input:   A bound $d$ of the length of test sequences
         A bound $m$ of the number of test sequences
Output: A set of test sequences $T$
 1: $T := \emptyset$;
 2: while $|T| < m$ do
 3:   $\rho := \epsilon$;
 4:   $s := s_0$;
 5:   while $|\rho| < d$ do
 6:     $(e, s) :=$ RandomlySelectOnePair(supp($s$));
 7:     $\rho := \rho \cdot e$;
 8:   end while
 9:   $T := T \cup \{\rho\}$;
10: end while
11: return $T$;
```

3.3 Testcase Generation

In this work, as mentioned in the introduction, we focus on the event sequence generation while the input data is chosen randomly. We first present the baseline algorithm for generating event sequences with partial-order reduction (POR) which is inspired by [11, 17], and then discuss how to generate long test sequences.

3.3.1 Baseline Event Sequence Generation

Assume the FSM model $M = (S, I, \delta, s_0)$, a length bound d, and the DOM event dependency relation $\to$. Algorithm 2 (excluding Lines 17 and 20) generates all possible event sequences with length up to d stored in T.

After initialing the set T for storing test sequences (Line 1) and the working stack ss for storing the event sequence with the initial state s_0 as the bottom element (Lines 2–3), it invokes the procedure EXPLORE to generate test sequences.

The procedure EXPLORE traverses the FSM model M in a depth-first manner, where E denotes the set of available events at the state s, s.SelectedEvent denotes the selected event at the state s, and s.done denotes the set of all previously selected events at the state s. The procedure EXPLORE first obtains the top state s on the stack ss which is achieved via replaying the recorded event sequence reaching s. Then, it checks whether the bound d is reached (Line 14). If ss.Length() $> d$, then the while-loop is skipped. After that, if s.SelectedEvent = null (indicating that the sequence in ss has reached the maximum length d), the event sequence stored in the stack ss is added to the set T. Otherwise, the while-loop will explore a previously unexplored event and invoke the procedure EXPLORE recursively. The while-loop terminates when all the available events at the state s have been explored. Note that in this case, s.SelectedEvent $\neq$ null, hence the sequence in ss will not be added to T.

With Lines 17 and 20, Algorithm 2 implements the partial-order reduction based on the notion of sleep-set [18]. In principle, it first partitions the event sequences into equivalence classes, and then explores the representative from each equivalence class. In detail, each event e explored at the state s is inserted into its sleep set s.sleep (Line 20). When another event e' is explored at s, the sleep set of the state s is copied to the sleep set of the next state s' (Line 15) if the DOM events e and e' are independent of each other. Later, each event at the state s will be skipped if it is in the sleep set of state s (Line 13), because executing this event is guaranteed to reach a previously explored state.

3.3.2 Long Event Sequence Generation

Algorithm 3 shows the pseudocode of our random event sequence generation. Given the FSM model $M = (S, I, \delta, s_0)$, a maximum length bound d and a maximum bound m of the number of event sequences, Algorithm 3 randomly generates m number of event sequences with length d. Each iteration of the outer while-loop computes one event sequence with length d until the number of event sequences reaches m. In the inner while-loop, it starts from the initial state s_0 and an empty sequence $\rho = \epsilon$. At each state s, the inner while-loop iteratively and randomly selects a pair (e, s') of event and state denoting that executing the event e at the state s moves to the state s', then appends e to the end of previously computed sequence ρ. Algorithm 3 repeats this procedure until the length of ρ reaches the maximum bound d. At this moment, one event sequence is generated and stored into the set T. The outer while-loop enters its next iteration.

Algorithm 3 may not generate all the possible event sequences, and may generate redundant event sequences, but less often, due to the large maximum bound. We remark that the POR technique cannot be integrated to Algorithm 3.

4 Experiments

We have implemented our method as a software tool LJS. It exploits JSDEP for computing DOM event dependency and a modified automated testing framework ARTEMIS as the execution engine. For comparison purposes, we implemented LJS in such a way that individual techniques are modularized and can be enabled on demand. This enables us to compare the performance of various approaches with different configurations, in particular, (1) our state abstraction versus the state abstraction from [2], and (2) baseline event sequence generation with POR (i.e., Algorithm 2) versus long event sequence generation (i.e., Algorithm 3). To demonstrate the efficiency and effectiveness of LJS, we compared LJS with JSDEP on same benchmarks. (We note that in the literature [11] JSDEP was shown to be superior to ARTEMIS, so a direct comparison between LJS and ARTEMIS is excluded.)

The experiments are designed to answer the following research questions:

RQ1. How efficient and effective is LJS compared with JSDEP?

RQ2. How effective is our coarse-grained state abstraction compared with the state abstraction from [2]?

RQ3. How effective is the long event sequence generation compared with the baseline algorithm?

To make the comparison on a fair basis, we evaluated LJS on publicly available benchmarks of JSDEP, which consist of 21 client-side JavaScript Web applications with 18,559 lines of code in total. Columns 1-2 of Table 1 show the name of the application and the number of lines of code. We ran all experiments on a server with 64-bit Ubuntu 12.04 OS, Intel Xeon(R) E5-2603v4 CPU (1.70 GHz, 6 Cores), and 32GB RAM. To answer **RQ1-3**, we conducted three case studies. The time used to compute the DOM event dependency is usually marginal and can be safely ignored, so is not counted in line with [11]. The coverage measure is the aggregation of that from model construction and testcase execution respectively which are separated in the last experiment. For statistics, we ran LJS on each application 21 times and took the average as the result in order to alleviate randomness. As our approach is non-deterministic and involves random choices, we conduct statistical analysis to show the statistical significance of the differences. Note that for statistical analysis, approximation using the normal distribution is fairly good when sample sizes are greater than 20 (see Mann-Whitney U-test on Wikipedia). Thus we ran LJS on each application 21 times.

4.1 **RQ1:** LJS vs. JSDEP

We study **RQ1** by performing two experiments assessing effectiveness and efficiency. For effectiveness, we compare the (line) coverage of LJS with JSDEP in 600 seconds. For efficiency, we compare the time used by LJS and JSDEP to achieve the same coverage. The experiments of LJS were performed with the following configurations: Max. Length=99, our new state abstraction, running Algorithm 1 two times, and long event sequence generation. Experiments of JSDEP were performed with the setting given in Sung et al. [11].

Following the suggestion [19], we perform the following statistical analysis. The first one is a two-tailed non-parametric One-Sample Wilcoxon test with the null hypothesis H_0: JSDEP and LJS achieve the same coverage. The second one is the one-tailed Wilcoxon signed-rank test with the null hypothesis is that JSDEP can achieve

Table 1 Coverage of LJS and JSDep in 600 seconds

Name	♯Loc	LJS				JSDep			p-value	Effect size
		Coverage	sd	♯Tests	Max. length	Coverage	♯Tests	Max. length		
case1	59	100.0%	0.000	4803	99	100.0%	1409	705	NA	NA
case2	72	100.0%	0.000	4470	99	100.0%	3058	549	NA	NA
case3	165	100.0%	0.000	2789	99	100.0%	7811	575	NA	NA
case4	196	**87.0%**	0.000	2885	99	**77.9%**	8594	**500**	$< 1 \times 10^{-5}$	1
frog	567	**96.6%**	0.006	**10**	99	**84.6%**	**86**	**16**	5×10^{-5}	1
cosmos	363	87.9%	0.059	268	99	79.5%	973	243	9.6×10^{-4}	0.71
hanoi	246	88.3%	0.013	1470	99	82.5%	902	225	1×10^{-5}	0.95
flipflop	525	97.1%	0.004	60	99	96.3%	284	71	1×10^{-5}	1
sokoban	**3056**	**88.2%**	0.027	55	99	**77.6%**	**203**	**51**	5×10^{-5}	1
wormy	570	44.2%	0.052	34	99	41.0%	323	18	0.70512	0.67
chinabox	338	83.7%	0.003	7	99	82.3%	92	9	2×10^{-5}	1
3dmodel	**5414**	**85.0%**	0.001	**9**	99	**71.5%**	**66**	**10**	1×10^{-5}	1
cubuild	**1014**	**87.5%**	0.035	**8**	99	**72.8%**	**153**	**17**	5×10^{-5}	1
pearlski	960	55.1%	0.000	82	99	54.9%	214	52	$< 1 \times 10^{-5}$	1
speedyeater	784	90.0%	0.008	550	99	82.1%	1497	374	6×10^{-5}	1
gallony	300	94.5%	0.001	2395	99	94.5%	1611	95	2.4×10^{-4}	0.05
fullhouse	528	92.1%	0.012	1168	99	86.3%	889	222	3×10^{-5}	1
ball_pool	**1745**	**93.2%**	0.005	**2**	99	**74.2%**	**18**	**3**	6×10^{-5}	1
harehound	468	95.0%	0.004	498	99	94.5%	1224	116	1.3×10^{-4}	0.81
match	369	72.5%	0.002	1142	99	73.2%	4050	845	1×10^{-5}	0
lady	820	79.1%	0.002	0	99	75.7%	35	8	5×10^{-5}	1
Average	883.8	**86.5%**	0.011	1081	99	**81.0%**	1595	224	NA	NA

Note: Name shows the name of the benchmark, ♯Loc shows number of lines of code, Coverage shows the line coverage, sd shows the standard deviation (sd) of coverage obtained in 21 runs, ♯Tests shows the number of test cases, Max length shows the maximum length of test sequences, and p-value is the two-tailed non-parametric One-Sample Wilcoxon test.

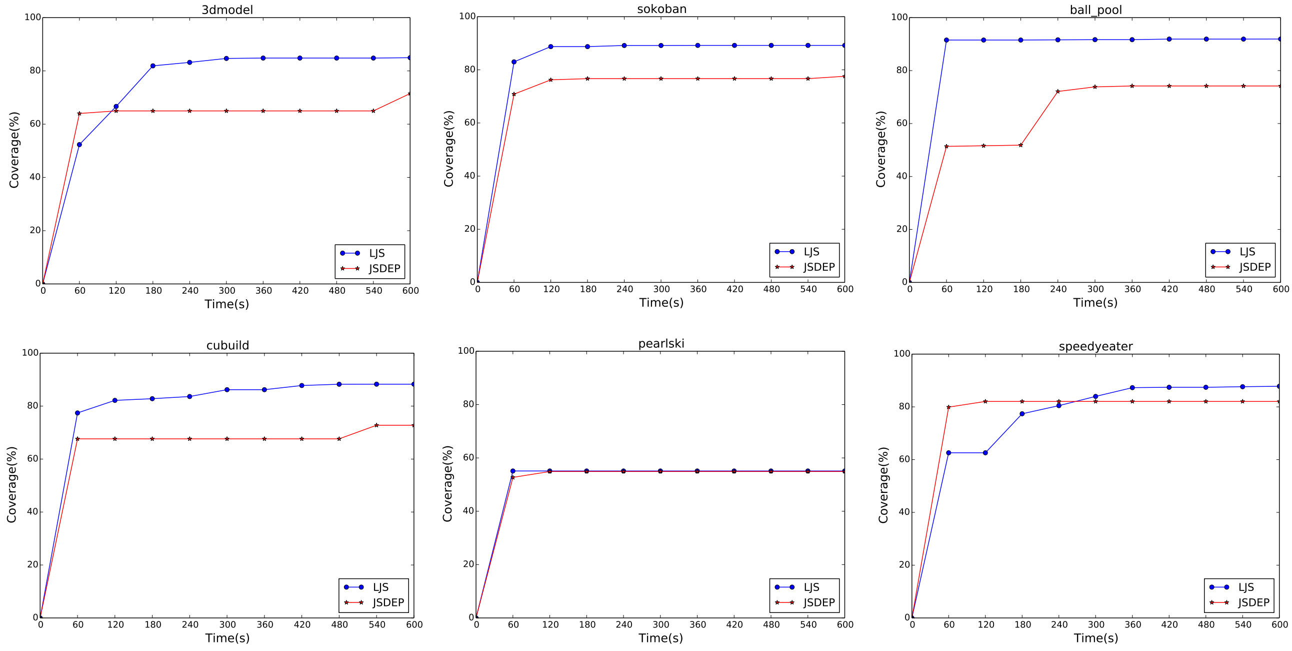

Fig. 5 Coverage of LJS and JSDep, as a function of the execution time

higher coverage than that of LJS on the benchmarks. These tests provide an overall comparison of the two tools. We also calculate the effect size using Vargha and Delaney's $\widehat{A}_{12}$ statistics, which measures the probability that LJS yields higher coverage than that of JSDEP. For instance, $\widehat{A}_{12} = 0.8$ entails one would obtain better results 80% of the time with LJS.

Table 1 shows the results of LJS and JSDEP in 600 seconds, where Columns 3-6 (resp. Columns 7-9) show line coverage, standard deviation (sd) of coverage obtained in 21 runs, number and length of event sequences after running LJS (resp. JSDEP) on average. Column 10 shows the relevant p-value of the two-tailed non-parametric One-Sample Wilcoxon test. Column 11 shows effect size. Note that significant improvements are highlighted in bold font.

Overall, we can observe an increase in the average coverage from 81% by JSDEP to 86.5% by LJS. (We remark that the average coverage was increased from 67% by ARTEMIS to 80% by JSDEP in 600s [11]. In our experiment, JSDEP performed slightly better.) Perhaps more importantly, on large applications such as *sokoban*, *3dmodel*, *cubuild* and *ball_pool*, the coverage of LJS is 11-19% higher than that of JSDEP. We can also observe that a small number of long event sequences could outperform a large number of short event sequences. For instance, LJS achieves 93.2% coverage on *ball_poll* using only 2 test cases with length 99, while JSDEP only achieves 74.2% using 18 test cases with maximal length 3. Similar results can also be observed on *frog*, *sokoban*, *3dmodel*, *cubuild*, etc. However, it should be emphasized that long event sequences, but of low quality, may not improve the coverage, as observed from the results of *case4*, *speedyeater* and *fullhouse*. This demonstrates that our model-based testing improves the coverage.

Furthermore, we conduct case studies to understand the experimental results. First, we investigate why LJS outperforms JSDEP on *ball_pool*, where LJS achieves 19% higher than JSDEP. We found that there is a function, called `getBodyAtMouse()`, which is invoked only when the global variable `isMouseDown` is true. While the global variable `isMouseDown` becomes true only when three DOM events `mousedown@body`, `mousedown@document` and `click@document` occur consecutively. A test sequence generated by our tool LJS contains this subsequence, hence covers the function `getBodyAtMouse()`. However, the 18 test sequences generated by JSDEP do not contain such subsequence. Consequently, JSDEP fails to cover the function `getBodyAtMouse()`. Second, we investigate why LJS does not improve the coverage on *pearlski* and *gallony*. For *pearlski*, both tools achieve only 55% code coverage. We found that *pearlski* contains a lot of unreachable code, e.g., functions that are defined but never invoked. Indeed, both tools covered almost all the reachable code. For *gallony*, we found that one branching is not covered by both tools, as the branching condition is too difficult to be satisfied using random input data. Third, we investigate why JSDEP achieves a higher coverage than LJS on *match*. After a depth-analysis, we found that JSDEP covers one more line of code than LJS and this line is guarded by a complicated branching condition. On these benchmarks, the lengths of evert sequences generated by JSDEP are long enough.

According to the findings of our case studies, to further improve the coverage, some path conditions should be satisfied. One solution is to integrate symbolic execution with our approach. We leave this as future work.

From the results of the two-tailed non-parametric One-Sample Wilcoxon test, in all nontrivial benchmarks the null hypothesis H_0 is rejected with the p-values reported in Column 10. In particular, 17 out of 18 benchmarks have p-values less than 0.001, indicating the statistical significance of the differences. For the one-tailed Wilcoxon signed-rank test, H_0 is rejected with the p-value is 2.3×10^{-4}, which also indicates that LJS obtains a higher coverage than JSDEP. From Column 11, we can observe that 16 out of 18 benchmarks have effect size greater than 0.67, indicating that LJS has higher probability to achieve a better coverage. Moreover, 12 out of 16 benchmarks have effect size 1 indicating that LJS outperforms JSDEP almost certainly.

Figure 5 shows the coverage that LJS and JSDEP achieved by running on the top 6 largest applications in time ranging from 60 seconds to 600 seconds with step size 60 seconds. (Note that the application *lady* is excluded in this experiments because its model construction takes more than 600 seconds; cf. Table 1.) The x-axis and y-axis represent the execution time budget and the achieved coverage respectively. Overall, as the execution time budget increases, the coverage of LJS is higher than that of JSDEP and the coverage becomes stable finally. Moreover, the rate of LJS to achieve a higher coverage is, in most cases, slightly higher than that of JSDEP. For instance, on *sokoban*, *ball_pool* and *cubuild*, LJS always achieves higher coverage than JSDEP with the increase of execution time. On *3dmodel* and *speedyeater*, though LJS achieves lower coverage than JSDEP at the beginning time, it finally achieves higher coverage than JSDEP. This is because at the beginning time, our approach spends time on the FSM model construction.

Table 2 Comparison of state abstraction techniques

Name	State abstraction from [2]				Our state abstraction				p-value of $\lvert S\rvert$	p-value of $\lvert\delta\rvert$
	Coverage	$\lvert S\rvert$	$\lvert\delta\rvert$	Time(s)	Coverage	$\lvert S\rvert$	$\lvert\delta\rvert$	Time(s)		
case1	98.6%	1.0	2.0	0.3	98.6%	1.0	2.0	0.2	NA	NA
case2	98.3%	1.0	4.0	0.3	99.4%	1.0	4.0	0.3	NA	NA
case3	99.3%	1.0	6.0	0.4	98.2%	1.0	6.0	0.4	NA	NA
case4	85.6%	1.0	8.0	0.4	85.9%	1.0	8.0	0.4	NA	NA
frog	95.9%	**199.0**	**198.0**	96.3	96.0%	**27.6**	**102.0**	120.7	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
cosmos	79.9%	**127.1**	**196.5**	4.8	78.7%	**68.0**	**143.7**	4.7	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
hanoi	88.6%	105.8	186.7	0.8	88.5%	105.9	185.7	0.7	0.96987	0.50301
flipflop	96.6%	30.6	113.9	17.5	97.1%	28.6	108.3	17.6	0.36409	0.33882
sokoban	86.5%	**69.6**	**189.6**	18.9	87.7%	**31.2**	**126.5**	21.0	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
wormy	42.0%	188.3	197.9	32.8	42.0%	131.0	186.1	31.8	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
chinabox	83.6%	63.1	154.3	132.1	83.7%	68.1	158.5	141.4	0.00052	0.08411
3dmodel	83.7%	**3.0**	**25.0**	96.7	83.3%	**1.0**	**6.0**	96.5	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
cubuild	85.0%	86.6	165.5	125.3	84.6%	83.4	163.1	121.6	0.14685	0.24113
pearlski	55.1%	**138.7**	**196.0**	13.5	55.1%	**72.4**	**176.4**	13.9	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
speedyeater	85.3%	**168.0**	**198.0**	1.6	84.4%	**4.4**	**60.2**	2.1	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
gallony	94.5%	64.8	171.6	0.5	94.5%	62.4	173.4	0.5	0.17342	0.39768
fullhouse	92.7%	**167.4**	**197.8**	0.8	92.2%	**28.4**	**119.0**	1.0	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
ball_pool	93.1%	42.4	143.0	283.5	93.2%	41.9	146.5	237.5	0.57851	0.2788
harehound	81.2%	**188.6**	**198.0**	0.9	87.2%	**11.2**	**93.4**	1.7	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
match	67.9%	**15.2**	**176.0**	1.0	70.3%	**4.1**	**55.3**	1.0	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
lady	79.1%	**164.2**	**197.6**	896.2	79.2%	**86.5**	**174.2**	856.1	$< 1 \times 10^{-5}$	$< 1 \times 10^{-5}$
Average	84.4%	**87.0**	**139.3**	82.1	84.8%	**41.0**	**104.7**	79.6	NA	NA

$|S|$ and δ show numbers of states and transitions of the FSM model, and p-values are the two-tailed non-parametric Mann-Whitney U-test on the numbers of states and transitions.

4.2 **RQ2:** Comparison of State Abstraction Techniques

We study **RQ2** by performing one experiment and comparing the obtained FSM model and coverage results using our coarse-grained state abstraction and the state abstraction from [2] respectively. In this experiment, LJS constructs the FSM model by running Algorithm 1 two times using Max. Length=99 and generates two event sequences from the FSM model. Taking the time into account, the experiment only generates two event sequences. We perform the two-tailed non-parametric Mann-Whitney U-test on each benchmark with the null hypothesis H_0: the two state abstractions obtain the same numbers of states and transitions, as well as the one-tailed Wilcoxon signed-rank test on the whole benchmarks with the null hypothesis H_0: the coarse-grained state abstraction of LJS produces more states and transitions.

Table 2 shows the results in average, where Columns 2-5 (resp. Columns 6-9) show the coverage, numbers of states and transitions of the FSM model (note that we take the average of these numbers), and execution time after running LJS with the state abstraction from [2] (resp. our new state abstraction). For most benchmarks, H_0 is rejected, and Columns 10-11 respectively show the p-values of the two-tailed non-parametric Mann-Whitney U-test on the numbers of states and transitions. For the one-tailed Wilcoxon signed-rank test, H_0 is rejected with the p-values 5.9×10^{-4} and 9.6×10^{-4} on the number of states and transitions respectively.

Overall, the numbers of states and transitions using our state abstraction are much smaller, with a dramatic decrease in some large applications such as *sokoban*, *3dmodel*, *pearlski*, *speedyeater* and *harehound*, while the performance of the two state abstractions in terms of average coverage remains comparable or even better. This indicates that our coarse-grained state abstraction is able to eliminate redundant states and transitions with high statistical confidence.

4.3 **RQ3:** Comparison of Event Sequence Generations

We study **RQ3** by performing two experiments and comparing quality of event sequences generated by the long event sequence generation (Algorithm 3) and baseline event sequence generation with POR (Algorithm 2). In the first experiment, LJS constructs the FSM model by running Algorithm 1 two times using our coarse-grained state abstraction and generates two event sequences from the FSM model, while the maximum bounds (i.e., Max. Length) are 9, 39, 69, 99, 129 and 159. In the second experiment, we first generate an FSM model using the same configuration with a fixed maximum bound 99. For each application, from the same FSM model, LJS with Algorithm 3 enabled iteratively generates and executes event sequences with maximum bound 99 within 600 seconds time bound. Meanwhile, LJS with Algorithm 2 enabled generates and executes all event sequences with redundant sequences pruned up to some maximum bound so that the execution time exceeds 600 seconds. Moreover, the execution terminates when the execution time reaches 1200s. We perform the two-tailed non-parametric One-Sample Wilcoxon test on each benchmark (with the null hypothesis H_0: the two algorithms obtain the same coverage) and the one-tailed Wilcoxon signed-rank test on the whole benchmarks (with the null hypothesis H_0: Algorithm 1 produces a lower coverage than Algorithm 2.

Figure 6 shows the results of the first experiment, where the x-axis and y-axis represent the maximum bound and the achieved coverage with that bound respectively; the thick line (in boldface) shows the trend of average coverage. Overall, the average coverage increases quickly when the maximum bound increases from 9 to 39, but only slightly when it increases from 39 to 159.

Table 3 shows the results of the second experiment, where Columns 2-5 (resp. Columns 6-9) show the coverage (of testcase execution *only*), maximum sequence length, execution time and number of event sequences after running LJS with Algorithm 2 (resp. Algorithm 3) enabled. For the two-tailed non-parametric One-Sample Wilcoxon test, in most benchmarks, H_0 is rejected where Columns 10 shows the p-values. For the one-tailed Wilcoxon signed-rank test, H_0 is rejected with p-value 0.002.

Overall, the average coverage of Algorithm 3 (i.e., long event sequence generation) is 6.2% higher than that of Algorithm 2 (i.e., baseline event sequence generation with POR) with less execution time. In particular, the coverage improvement of Algorithm 3 is more prominent for the applications *case2*-*case4*, *chinabox*, *3dmodel* and *cubuild*. These results also confirm that executing fewer long event sequences could achieve a higher coverage than executing more short event sequences. The p-values of two statistical tests indicate that our long event sequence generation has better performance than Baseline with POR with high statistical confidence.

4.4 Discussion

LJS currently supports off-line testing (i.e., the source code is available) when the DOM event dependency is enabled. Our approach is also applicable in on-line testing if the DOM event dependency is computed dynamically, as it does not need to record and replay.

In the sequel, we note some limitations of the experiments. The experiments are based on the publicly available benchmarks that include only 4 large-scale Web applications (with more than 1k LOC, maximum 5k LOC). For the future work, we plan to experiment on more large-scale benchmarks. The main challenges include the shortage of benchmarks and that JSDEP [11] supports limited JavaScript constructs amenable to DOM dependency computing. Moreover, the current evaluation is based on coverage, but it would also be interesting to evaluate the fault detection, for which the test oracle should be investigated, as JavaScript employs the “no crash” philosophy. Furthermore, we note that there are several ways to obtain long sequences based on FSM, for instance, by following the dependency chains or giving the priority to novel states. These strategies deserve further exploration on top of our current findings and framework.

5 Related Work

We discuss the related work in the areas of model-based testing and automated JavaScript Web application testing.

5.1 Model-based Automated Testing

Model-based testing (MBT) has been widely used in software testing (cf. [20–22] for surveys). Mainstream MBT techniques differ mainly in three aspects: models of the software under test, model construction, and testcase generation. Several models, such as state-based (e.g., pre-/post-condition) and transition-based (e.g., UML and I/O automata), have been proposed. The FSM model used in this work is one of the transition-based models. Model construction is one of the most important tasks in MBT. It is usually time-consuming and error-prone to manually construct models for GUI-based applications [23, 24]. Therefore, most works use static or dynamic analysis to construct models, for example, [15, 25] for mobile/GUI applications. However, it is rather difficult to statically construct models for JavaScript Web applications due to their dynamic characteristics [3].

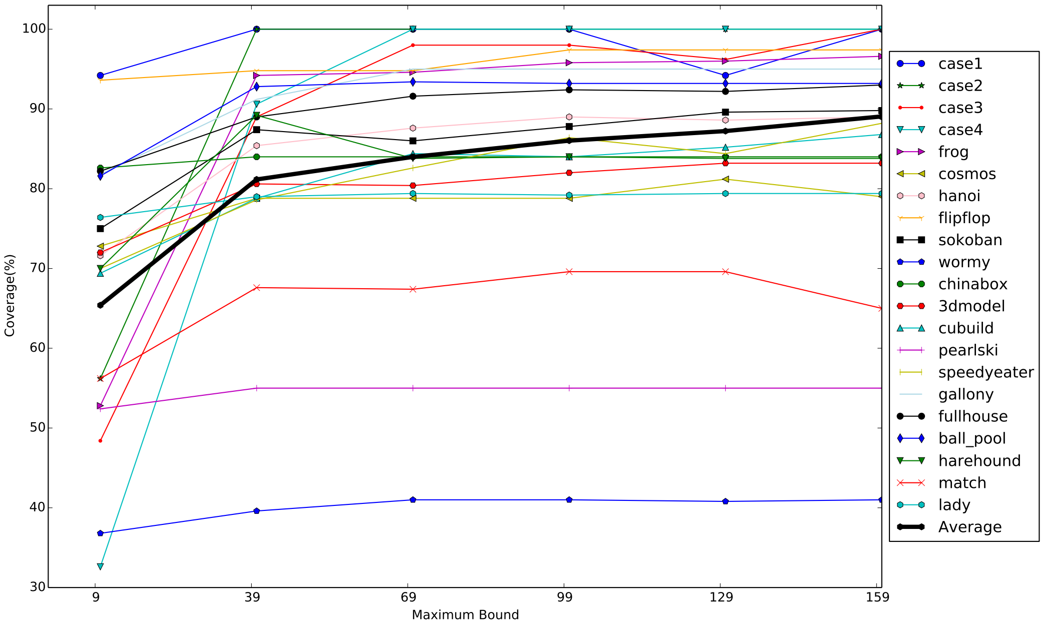

Fig. 6 Coverage of LJS, as a function of the length bound

Table 3 Comparison of event sequence generation algorithms

Name	Baseline with POR				Long Sequence Generation				p-value
	Coverage	Max. Length	Time(s)	♯Tests	Coverage	Max. Length	Time(s)	♯Tests	
case1	100.0%	16	829	65504	100.0%	99	600	4796	NA
case2	**87.8%**	9	1125	**87040**	**100.0%**	99	600	4482	$< 1 \times 10^{-5}$
case3	**68.6%**	7	931	**54432**	**100.0%**	99	600	2800	$< 1 \times 10^{-5}$
case4	**61.8%**	6	674	**32768**	**87.0%**	99	600	2887	$< 1 \times 10^{-5}$
frog	88.6%	4	664	592	95.1%	99	600	6	5×10^{-5}
cosmos	78.0%	5	1072	4415	88.2%	99	600	826	5×10^{-5}
hanoi	86.8%	6	889	4276	88.6%	99	600	1417	$< 1 \times 10^{-5}$
flipflop	97.0%	6	1200	419	97.0%	99	600	23	$< 1 \times 10^{-5}$
sokoban	88.6%	5	1200	1697	84.0%	99	600	66	1×10^{-5}
wormy	41.2%	13	657	317	49.7%	99	600	37	3.6×10^{-4}
chinabox	**78.0%**	8	**729**	**64**	**83.7%**	99	600	**10**	3×10^{-5}
3dmodel	**72.0%**	4	**1200**	**93**	**85.0%**	99	600	**11**	$< 1 \times 10^{-5}$
cubuild	**75.2%**	6	**1019**	**238**	**88.3%**	99	600	**10**	5×10^{-5}
pearlski	52.1%	7	960	653	51.1%	99	600	78	2.4×10^{-4}
speedyeater	82.3%	5	**1200**	17106	88.0%	99	600	548	$< 1 \times 10^{-5}$
gallony	94.5%	8	1200	9821	92.6%	99	600	2039	1×10^{-5}
fullhouse	79.2%	8	1200	12097	75.2%	99	600	757	1×10^{-5}
ball_pool	92.1%	6	1200	22	93.4%	99	600	3	5×10^{-5}
harehound	92.2%	4	1200	7949	95.5%	99	600	487	4×10^{-5}
match	73.2%	5	738	13341	72.1%	99	600	1172	$< 1 \times 10^{-5}$
lady	73.2%	5	696	12	79.2%	99	600	2	5×10^{-5}
Average	**79.2%**	7	980	**14898**	**85.4%**	99	600	**1069**	NA

Coverage *only* includes that of event sequence execution, ♯Tests shows the number of event sequences and p-value is the two-tailed non-parametric One-Sample Wilcoxon test.

Regarding work on model construction for JavaScript Web applications, [26] have to construct a model manually; [27] extracts the model via static analysis, but lacks of considering dynamic nature of JavaScript; [3] construct FSM models via dynamic analysis to crawl Web applications. The main difference between our work and theirs [3] is the way in which the model is constructed. Our model construction pursues a larger depth without backtracking, but does not cover all possible event sequences, whereas [3] cover all the possible event sequences up to a length bound with backtracking. [28] also reported FSM model constructions, but did not give detail of their algorithm, nor included JavaScript coverage. Existing testcase generation algorithms mainly focus on the systematic generation of event sequences with a rather limited length bound due to the "test sequence explosion" problem. Our approach generates long event sequences, but strategically avoid covering all possible event sequences (up to a length bound) to mitigate the exponential blowup problem.

5.2 JavaScript Web Application Automated Testing

Web application testing has been widely studied in the past decade, differing mainly in targeted Web programming languages (e.g., PHP [29] and JavaScript [2, 3]), and testing techniques (e.g., model-based testing [20–22], mutation testing [5,8], search-based testing [29] and symbolic/concolic testing [1,4,6]; cf. [10,16] for surveys). We mainly compare with the work on automated testing of JavaScript Web Application.

Test sequences of JavaScript Web Applications consist of event sequences and input data for each event. Existing work creates event sequences via exploring the state space by randomly selecting events with heuristic search strategies. For instance, Kudzu [1] and Crawljax [3] randomly select available events. Moreover, Crawljax relies on a heuristic approach for detecting event handlers, hence may not be able to detect all of them. Artemis [2] uses the heuristic strategy based on the observed read and write operations by each event handler in an attempt to exclude sequences of non-interacting event handler executions. EventBreak [7] uses the heuristic strategy based on performance costs in terms of the number of conditions in event handlers in an attempt to analyze responsiveness of the application. These approaches usually cover all the sequences up to a given, usually small, length bound. In order to explore long event sequences in limited time, delta-debugging based method [30] and partial-order reduction [11] were proposed for pruning redundant event sequences. Our approach does not cover all possible event sequences, hence can create long event sequences within the time budget. Experimental results show that our approach can achieve a high line coverage than Artemis even with partial-order reductions [11].

Another research line in automated testing of JavaScript Web applications is to generate high quality input data of events using symbolic/concolic testing, e.g., [1, 4, 6, 9]. These approaches are able to achieve a high coverage, but rely heavily on the underlying constraint solver. They generally do not scale well for large, realistic programs, because the number of the feasible execution paths of a program often increases exponentially in the length of the path. Our work focuses on the generation of long event sequences, but choose the input data randomly, which is orthogonal, and could be complementary, to the more advanced input data generation methods. A transfer technique has been proposed based on the automation engine framework Selenium to transfer tests from one JavaScript Web application to another [31]. This is orthogonal to our work.

6 Conclusion

We have proposed a model-based automated testing approach for JavaScript Web applications. Our approach distinguishes from others in making use of long event sequences in both the FSM model construction and the testcase generation from the FSM model. We have implemented our approach in a tool LJS and evaluated it on a selection of benchmarks. The experimental results showed that our approach is more efficient and effective than Artemis and JSdep. Furthermore, we empirically found that proper longer test sequences can achieve a high line coverage.

Acknowledgements The authors would like to thank Dr. Ting Su for his valuable comments and suggestions. P. Gao, Y. Jie and F. Song were partially supported by the National Natural Science Foundation of China (NSFC) grants (No. 62072309, No. 61532019 and No. 61761136011). T. Chen is partially supported by the National Natural Science Foundation of China (No. 61872340), Guangdong Science and Technology Department Grant (No. 2018B010107004) and Natural Science Foundation of Guangdong Province, China (No. 2019A1515011689).

References

1. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S,

Song D. A symbolic execution framework for JavaScript. In: Proceedings of IEEE Symposium on Security and Privacy. 2010, 513–528

2. Artzi S, Dolby J, Jensen S H, Møller A, Tip F. A framework for automated testing of JavaScript web applications. In: Proceedings of International Conference on Software Engineering. 2011, 571–580
3. Mesbah A, Van Deursen A, Lenselink S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. ACM Transactions on the Web, 2012, 6(1):1–30
4. Sen K, Kalasapur S, Brutch T G, Gibbs S. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. 2013, 488–498
5. Mirshokraie S, Mesbah A, Pattabiraman K. Efficient JavaScript mutation testing. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation. 2013, 74–83
6. Li G, Andreasen E, Ghosh I. SymJS: automatic symbolic testing of JavaScript web applications. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014, 449–459
7. Pradel M, Schuh P, Necula G, Sen K. EventBreak: analyzing the responsiveness of user interfaces through performance-guided test generation. In: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages & Applications. 2014, 33–47
8. Mirshokraie S, Mesbah A, Pattabiraman K. Guided mutation testing for JavaScript web applications. IEEE Transactions on Software Engineering, 2015, 41(5):429–444
9. Sen K, Necula G C, Gong L, Choi W. MultiSE: multi-path symbolic execution using value summaries. In: Proceedings of Joint Meeting on Foundations of Software Engineering. 2015, 842–853
10. Andreasen E, Gong L, Møller A, Pradel M, Selakovic M, Sen K, Staicu C. A survey of dynamic analysis and test generation for JavaScript. ACM Computing Surveys, 2017, 50(5):66:1–66:36
11. Sung C, Kusano M, Sinha N, Wang C. Static DOM event dependency analysis for testing web applications. In: Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016, 447–459
12. Arcuri A. Longer is better: On the role of test sequence length in software testing. In: Proofceedings of International Conference on Software Testing, Verification and Validation. 2010, 469–478
13. Andrews J H, Groce A, Weston M, Xu R G. Random test run length and effectiveness. In: Proofceedings of IEEE/ACM International Conference on Automated Software Engineering. 2008, 19–28
14. Fraser G, Gargantini A. Experiments on the test case length in specification based test case generation. In: Proceedings of International Workshop on Automation of Software Test. 2009, 18–26
15. Carino S, Andrews J H. Evaluating the effect of test case length on GUI test suite performance. In: Proceedings of IEEE/ACM International Workshop on Automation of Software Test. 2015, 13–17
16. Li Y F, Das P K, Dowe D L. Two decades of web application testing - A survey of recent advances. Information Systems, 2014, 43:20–54
17. Cheng L, Yang Z, Wang C. Systematic reduction of GUI test sequences. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2017, 849–860
18. Godefroid P, van Leeuwen J, Hartmanis J, Goos G, Wolper P. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Heidelberg: Springer, 1996
19. Arcuri A, Briand L C. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability, 2014, 24(3):219–250
20. Dias-Neto A C, Travassos G H. A picture from the model-based testing area: Concepts, techniques, and challenges. Advances in Computers, 2010, 80:45–120
21. Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing approaches. Software Testing, Verification & Reliability, 2012, 22(5):297–312
22. Li W, Le Gall F, Spaseski N. A survey on model-based testing tools for test case generation. In: Proceedings of International Conference on Tools and Methods of Program Analysis. 2018, 77–89
23. Jensen C S, Prasad M R, Møller A. Automated testing with targeted event sequence generation. In: Proceedings of International Symposium on Software Testing and Analysis. 2013, 67–77
24. Takala T, Katara M, Harty J. Experiences of system-level model-based GUI testing of an android application. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation. 2011, 377–386
25. Rau A, Hotzkow J, Zeller A. Efficient GUI test generation by learning from tests of other apps. In: Proceedings of International Conference on Software Engineering: Companion Proceedings. 2018, 370–371
26. Andrews A A, Offutt J, Alexander R T. Testing web applications by modeling with fsms. Software and System Modeling, 2005, 4(3):326–345
27. Ricca F, Tonella P. Analysis and testing of web applications. In: Proceedings of International Conference on Software Engineering. 2001, 25–34
28. Dallmeier V, Burger M, Orth T, Zeller A. Webmate: a tool for testing web 2.0 applications. In: Proceedings of

Workshop on JavaScript Tools. 2012, 11–15

29. Alshahwan N, Harman M. Automated web application testing using search based software engineering. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering. 2011, 3–12
30. Nguyen C, Yoshida H, Prasad M R, Ghosh I, Sen K. Generating succinct test cases using don't care analysis. In: Proceedings of IEEE International Conference on Software Testing, Verification and Validation. 2015, 1–10
31. Rau A, Hotzkow J, Zeller A. Transferring tests across web applications. In: Proceedings of International Conference on Web Engineering. 2018, 50–64

Pengfei Gao received the B.S. degree in Communication Engineering from China University of Mining and Technology, Jiangsu, China, in 2017.

He is currently a Ph.D. student in ShanghaiTech University, Shanghai, China, supervised by Prof. Fu Song. His research interests include program analysis and software security.

Yongjie Xu received the B.S. degree in Computer Science and Technology from Shanghaitech University, Shanghai, China, in 2019. He is currently a M.S. student in ShanghaiTech University, supervised by Prof. Fu Song. His research interests include SAT solving and software security.

Fu Song received the B.S. degree from Ningbo University, Ningbo, China, in 2006, the M.S. degree from East China Normal University, Shanghai, China, in 2009, and the Ph.D. degree in computer science from University Paris-Diderot, Paris, France, in 2013.

From 2013 to 2016, he was a Lecturer and Associate Research Professor at East China Normal University. Since August 2016, he is an Assistant Professor with ShanghaiTech University, Shanghai, China. His research interests include formal methods and computer security, especially about automata, logic, model checking, and program analysis.

Dr. Song was a recipient of EASST best paper award at ETAPS 2012.

Taolue Chen received the B.S. and M.S. degrees from the Nanjing University, China, both in Computer Science. He was a junior researcher at the Centrum Wiskunde & Informatica (CWI) and acquired the Ph.D. degree from the Vrije Universiteit Amsterdam, The Netherlands. He is currently a lecturer at the Department of Computer Science and Information Systems, Birkbeck, University of London. His research interests include formal verification and synthesis, program analysis, logic in computer science, and software engineering.