Context-Based Event Trace Reduction in Client-Side JavaScript Applications

Jie Wang, Wensheng Dou*, Chushu Gao, Yu Gao, Jun Wei
University of Chinese Academy of Sciences, China
State Key Lab of Computer Science, Institute of Software, Chinese Academy of Sciences, China
{wangjie12, wsdou, gaochushu, gaoyu15, wj}@otcaix.iscas.ac.cn

***Abstract*—Record-replay techniques are developed to facilitate debugging client-side JavaScript application failures. They faithfully record all events that reveal a failure, but record many events irrelevant to the failure. Delta debugging adopts the divide-and-conquer algorithm to generate a minimal event subtrace that still reveals the same failure. However, delta debugging is slow because it may generate lots of syntactically infeasible candidate event subtraces in which some events can trigger syntactical errors (e.g., ReferenceError and TypeError), and thus cannot be replayed as expected. Based on this observation, we propose EvMin, an effective and efficient approach to remove failure-irrelevant events from an event trace. We use the variable usage information (e.g., DOM variable usage) in an event to model the event's context. We require that, each event in an event subtrace has the compatible context with its corresponding one in the original event trace. In this way, we avoid generating syntactically infeasible event subtraces, and dramatically speed up delta debugging. We have implemented EvMin and evaluated it on 10 real-world JavaScript application failures. Our evaluation shows that EvMin generates 72% fewer event subtraces, and takes 84% less time than delta debugging.**

Keywords-JavaScript; event trace reduction; delta debugging

I. INTRODUCTION

JavaScript has become the most popular language for client-side JavaScript applications. Due to its dynamic and event-driven features, it is challenging to debug client-side JavaScript applications. Thus, various record-replay techniques [1][2] are developed to faithfully reproduce the failures in client-side JavaScript applications.

However, client-side JavaScript applications usually keep running for a long time, e.g., writing a document in Google Doc may take a few hours. Thus, existing record-replay techniques [1][2][3][4] will generate a very long event trace. For example, Mugshot [1] can generate 75-795KB uncompressed event trace (nearly 3,000 events) per minute. It is time-consuming and exhausting to debug with such a long event trace. According to a recent study [5], a shortened event trace can significantly increase programmers' efficiency in failure diagnosis, fault localization and fault correction.

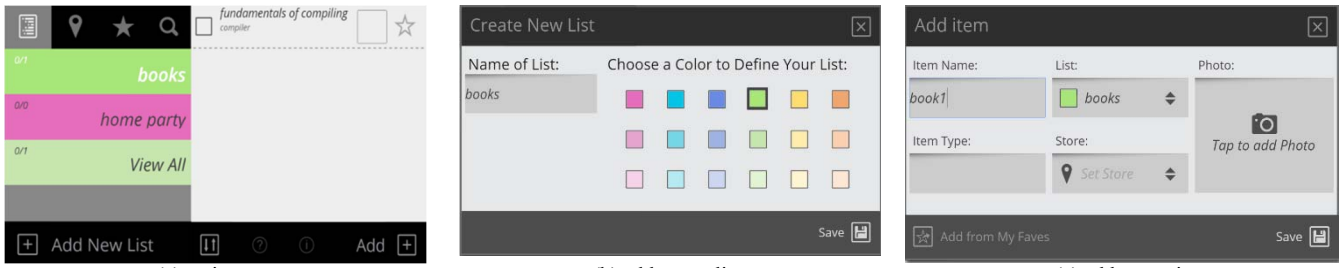
Delta debugging [6][7][8] is a general technique applicable to minimize failure-inducing inputs. Hammoudi et al. [5] adapted the delta debugging algorithm proposed by Zeller et al. [8] to minimize an event trace that leads to a JavaScript application failure. The algorithm partitions the event trace to a number of candidate subtraces and validates if each subtrace *s* as well as its complement subtrace (i.e., the events that are in the original trace but not in *s*) can reproduce the original failure. If such a subtrace *s* is found, then the algorithm repeats the same procedure on *s* until no shorter subtrace can be found.

There usually exist complicated dependences among the events in a real event trace, which is collected from the execution of a client-side JavaScript application. Consider the following example. Event e_i creates some DOM elements that are used by a following event e_j. If event e_i is ignored, then event e_j will trigger a TypeError. However, delta debugging ignores such a relationship among the events in the recorded event trace, and may generate many infeasible candidate subtraces. Thus, much time is wasted on generating and validating such infeasible candidate subtraces. For an example in our experiment, delta debugging takes 27 minutes for an event trace with only 617 events. Hierarchical Delta Debugging (HDD) [9] takes advantage of the tree structure of inputs (e.g., XML) to speed up delta debugging. However, HDD does not work in our situation due to the lack of similar tree-like structure in event traces in client-side JavaScript applications.

In this paper, we propose a novel approach, *EvMin*, to *effectively* and *efficiently* select minimal failure-inducing events from an event trace that reveals a failure. We observe that, for a candidate event subtrace that may reproduce the original failure, the shared variable (i.e., the variable can be accessed across events) usage information of each event, such as the existence of its accessed DOM structure and the types of its accessed variables, should stay the same as the corresponding information in the original trace. Otherwise, new syntactical failures (e.g., ReferenceError and TypeError) other than the original failure may occur. We consider this shared variable usage information about an event as its *context*. By requiring that each event in the candidate subtrace has the compatible context with its corresponding one in the original trace, EvMin can avoid exploring syntactically infeasible subtraces that may produce unexpected failures. Thus, EvMin can speed up delta debugging greatly.

We have implemented our approach as a prototype tool in pure JavaScript. Our tool enables easy integration into front-end source code (by switching a proxy in the browser), and avoids modifications to a JavaScript engine. Our evaluation on 10 real-world JavaScript application failures shows that EvMin can effectively find the minimal failure-inducing events. Compared with delta debugging, EvMin explores 72% fewer candidate subtraces and takes 84% less time on average.

* Corresponding author



(a) main page (b) add a new list (c) add a new item

Fig. 1. Shopping list application.

In summary, the contributions of this paper are as follows:

- We propose a context-based approach to effectively and efficiently perform event trace reduction in client-side JavaScript applications. Our approach can avoid exploring many syntactically infeasible candidate event subtraces that delta debugging does not.
- The evaluation on 10 real-world JavaScript application failures shows that our approach can effectively and efficiently remove failure-irrelevant events.

The remainder of this paper is organized as follows. Section II presents our motivation and challenges. Section III introduces our approach and Section IV describes the implementation details. Section V evaluates EvMin experimentally. Section VI and VII discuss the limitations of EvMin and introduce related work, respectively. Section VIII concludes this paper.

II. MOTIVATING EXAMPLE

In this section, we illustrate our motivation and approach overview using a real-world JavaScript application failure.

A. Example

Fig. 1 shows a client-side JavaScript application that is used to manage shopping lists. In Fig. 1, a new list can be added by clicking the *Add New List* button (in the bottom left of Fig. 1a) and a dialog will show up for adding a new list (Fig. 1b). A new item can be added to a given list by clicking the *Add* button (in the bottom right of Fig. 1a) and a dialog will show up for adding a new item (Fig. 1c). Fig. 2 shows the simplified source code for this example. The event handler *onload* (Lines 3-6) will be called when the page is loaded, *onAddNewList* (Lines 16-19) will be invoked when the *Add New List* (in Fig. 1a) button is clicked, *onSaveList* (Lines 21-33) will be invoked when the *Save* button in the dialog in Fig. 1b is clicked, and *onSaveItem* (Lines 35-49) will be invoked when the *Save* button in the dialog in Fig. 1c is clicked.

Items with duplicated names are not allowed in the same shopping list for this application. Otherwise, an exception will be thrown (Line 43 in Fig. 2). Fig. 3 shows a real event trace that triggers this failure. This event trace contains 31 events. However, only the following 12 events (in bold font in Fig. 3) are necessary to trigger this failure: event 0 (action0) that initializes a shopping list when the initial page is loaded, events 1-3 (action1) that create a list with name "books", events 14-17 (action3) that add a new book named "book1" to the list "books", and events 27-30 (action6) that add a book named "book1" again. In this event trace, action3 and action6 add items with the same name "book1" to the shopping list "books". All the other events are failure-irrelevant, such as events 18-21 that add an item with a different name (e.g., "book2") to list "books". The irrelevant events randomly interleave with the failure inducing events. We aim to find the minimal subtrace that reproduces the failure.

B. Existing Solutions

Given an event trace τ that raises a failure f, delta debugging [5][8] attempts to partition the event trace τ into a number of candidate subtraces and validates each subtrace as well as its complement subtrace. If a subtrace can raise the failure f, the algorithm treats the failure-triggering subtrace as trace τ and repeats the same procedure until there is no smaller subtrace that triggers the failure f. Delta debugging can achieve nearly binary search complexity in the best case. For the 31 events in Fig. 3, delta debugging generates 69 candidate subtraces for isolating the minimal failure-inducing subtrace.

We find that delta debugging generates and tests many infeasible candidate subtraces that cannot reproduce the failure. For each event in the given trace τ, it usually depends on some preconditions, such as the existence of a DOM element. If these preconditions are not satisfied, the generated subtraces may be infeasible, and trigger failures different from the original failure f. For our example in Fig. 3, event 4 depends on event 0 since its accessing DOM elements are loaded by event 0. It also depends on event 3, because event 3 registers the event listener *onclick* (Line 28) that event 4 triggers. Delta debugging does not care about such preconditions and may generate an infeasible trace {event 4-30} that introduces a new TypeError since the accessed DOM element is not loaded. Even if the DOM elements exist, event 4 cannot be triggered because event 3 that registers the event listener for event 4 is not selected.

Dynamic slicing approaches [10][11][12][13] adopt the def-use information to trace the data dependences and compute failure-relevant events by keeping all events that the failing event depends on. However, these approaches cannot remove all failure-irrelevant events. For example, the irrelevant event 21 cannot be removed because event 21 is depended by the failing event 30 (i.e., event 30 reads the value *curList.totalCount* written by event 21 at Line 46 in Fig. 2). Our approach differs from dynamic slicing approaches in that we can remove the failure-irrelevant events even if they are depended by the failing event, such as event 21.

```
1.  //shoppingLists is a shared variable used to store all lists.
2.  var shoppingLists;
3.  window.onload = function(){
4.     shoppingLists = new ShoppingLists();
5.     render();
6.  }
7.  /* Show the adding new list dialog*/
8.  function showEditListDialog(){
9.     document.getElementById('newlist-form').style.display ='block';
10. }
11. /* Hide the adding new list dialog*/
12. function hideEditListDialog(){
13.    document.getElementById('newlist-form').style.display = 'none';
14. }
15. /* Event handler for adding a new list */
16. function onAddNewList (){
17.    resetForm(); //Reset the form fields in the dialog
18.    showEditListDialog(); //Show the dialog
19. }
20. /* Event handler for saving a new list*/
21. function onSaveList (){
22.    var listName = getById('listName').value;
23.    var newList = new ShoppingList(listName);
24.    shoppingLists.pushList(newList); // Add newList
25.    hideEditListDialog();
26.    //Render the new created list to list pane
27.    newListView = createListView(newList);
28.    newListView.onclick = function(){
29.       var items = shoppingLists.getList(newList.name).getItems();
30.       renderItemsPane(items);
31.    }
32.    appendToListsPane(newListView);
33. }
34. /* Event handler for saving new item*/
35. function onSaveItem (){
36.    var itemName = document.getElementById('name').value;
37.    var boughtCount =document.getElementById('count').value;
38.    var listName = document.getElementById('item_list').value;
39.    var item = new Item(itemName, boughtCount);
40.    // Save new created item and update the screen
41.    var curList = shoppingLists.getList(listName);
42.    if(curList [item.name]!=null){
43.       throw new Error("duplicated item!");
44.    }
45.    pushItem(item);
46.    curList.totalCount += boughtCount;
47.    hideEditItemDialog();
48.    render();
49. }
50. /* Define class ShoppingList */
51. function ShoppingList (){
52.    this.totalCount = 0;
53.    …
54. }
```

Fig. 2. Code snippet for the shopping list application.

C. Our Approach

To effectively and efficiently find the minimal failure-inducing events for an event trace that triggers a failure, our approach utilizes the variable usage context of each event to guide the generation of candidate event subtraces. If an event *e* is selected in a candidate event subtrace, we require that the context of each event in a candidate event subtrace should keep the same with the context of its corresponding event in the original event trace. In this way, we avoid generating syntactically-infeasible candidate event subtraces that delta debugging does not. On the other hand, we relax the strict program dependence in dynamic slicing and use the variable usage context to generate shorter subtraces.

Action	Event	Event handler
0. Page load	**0. Load the page**	onload (Line 3)
1. Add list named "books"	**1. Click *Add New List* button** **2. Fill in the list name** **3. Click *Save* button**	onAddNewList onSaveList
2. Other actions	4. Click list "books" 5. Click list "books" …//changing settings	onclick (Line 28) onclick (Line 28)
3. Add item named "book1" to list "books"	**14. Click *Add* button** **15. Fill in the *name* of new item** **16. Fill in the *count* of new item** **17. Click *Save* button**	onAddItem onSaveItem
4. Add item named "book2" to list "books"	(Repeat 14~17) 18. Click *Add* button 19. Fill in the *name* of new item 20. Fill in the *count* of new item 21. Click *Save* button	 onAddItem onSaveItem
5. Other actions	…	
6. Add item named "book1" to list "books"	(Repeat 14~17) **27. Click *Add* button** **28. Fill in the *name* of new item** **29. Fill in the *count* of new item** **30. Click *Save* button**	 onAddItem onSaveItem

Fig. 3. A real event trace from ShoppingList application. *It triggers a failure because two items with the same name are added into the same list.*

To reach our purpose, two technical problems should be addressed first.

Problem 1: What information can be used as an event's context? How can we generate syntactically-feasible candidate subtraces?

We observe that many subtraces generated by delta debugging contain some events whose required shared information is not available, e.g., DOM elements. Thus, we use the shared variable usage information to model an event's context. For example, if an event's handler accesses some shared variables in the original trace, then we require that these shared variables are also accessible for its corresponding event in the reduced event trace. Thus, the new generated trace will not raise a TypeError.

We use the following variable information to model an event's context: the existence of required shared variables, shared variables' declaring scope and types. Note that, we do not care about these variables' concrete values, since concrete values may not affect the occurrence of the failure. For example, the variable *curList.totalCount* is accessed by event 30, event 21 and event 17. We observe that if we delete event 21, then the variable *curList.totalCount* in event 30 will read from event 17. The concrete value of *curList.totalCount* does not affect the occurrence of the failure.

Although some concrete values indeed affect the occurrence of the failure, it is challenging to analyze what concrete values can trigger the failure. Thus, we adopt the idea

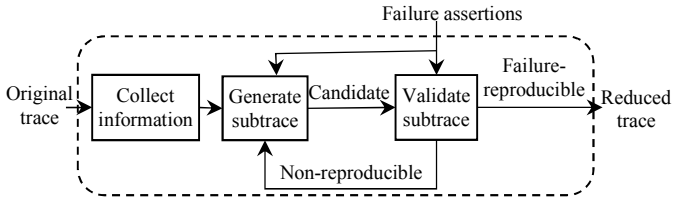


Fig. 4. Approach overview.

of replaying the subtrace like delta debugging, and check whether the expected failure occurs.

Problem 2: For client-side JavaScript applications, how to determine the context of each event in an event subtrace?

To handle problem 1, we need to determine the context of each event in a subtrace. Specially, for client-side JavaScript applications, it is challenging to handle the DOM variables (DOM is a tree-structure representation of a HTML page). DOM is a built-in object in the browser, and can be regarded as a variable with type *object*. However, there are many native APIs that operate the DOM elements or attributes, such as *appendChild*, *remove*, and *setAttribute*. An operation on a DOM element may indirectly affect another DOM element. Thus, it is challenging to decide the context of a DOM element (e.g., whether a certain DOM element exists). To handle this, we determine the compatible context of a DOM element by searching a DOM state and checking if it contains the necessary DOM elements.

III. APPROACH

Given an event trace that leads to a JavaScript application failure, our approach tries to identify a minimal event subtrace, which can still reproduce the failure. Fig. 4 shows the overview of our approach. The input of our approach includes: (1) The original event trace that triggers a failure. The event trace can be collected by existing event-based record-replay tools, such as Mugshot [1], which records low-level events according to the DOM specification [14]. (2) Failure assertions. We require developers to provide the assertions that describe the symptom of the original failure, just as similar measurements do [5][10][15], so that we can tell whether the original failure is reproduced.

The output of our approach is a ***failure-reproducible*** event subtrace. We consider an event subtrace as failure-reproducible (i.e., it can reproduce the failure) only if the failure assertions are satisfied during the replay of the event subtrace.

Our approach consists of three phases. First, we instrument the source code and replay the original event trace to collect the context information for each event (Section A). Second, we generate a candidate event subtrace that is likely to be failure-reproducible. We transform the candidate event subtrace generation into a constraint solving problem (Section B). Specifically, we require that, for a candidate event subtrace, all the events have compatible contexts with their corresponding ones in the original event trace. Third, we check whether the candidate event subtrace is failure-reproducible. We repeat steps 2 and 3 until a failure-reproducible event subtrace is found (i.e., the failure is reproduced; Section C).

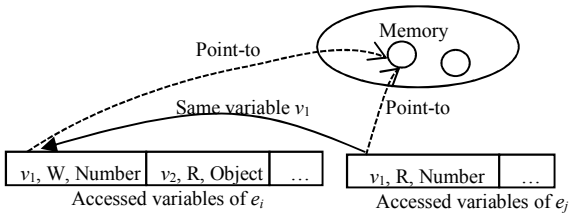


Fig. 5. Event memory access model. Events e_i and e_j access the same variable with the same name v_1 (e_i writes v_1, e_j reads v_1).

A. Collecting Context Information

As mentioned earlier in Section II.C, we use the event's variable usage information to model its context. Thus, we need to collect runtime memory access information. We use Fig. 5 to explain how memory is accessed for each event. An event handler may access many variables, each accessed variable can be denoted as a tuple <*variable name*, *accessing type*, *value type*>. The variable name denotes the name that is used to access the variable (e.g., v_1). The accessing type can be *declare* (D), *write* (W), and *read* (R). A variable can be declared, read and written multiple times. Since JavaScript is a dynamically typed programming language, we use the value type to denote the type of a variable during execution. The type information of a variable can be obtained by the *typeof* operator in JavaScript.

If two operations use the same variable name, they access the same variable. We name the variables that can be accessed by at least 2 events as *shared variables*. For example, in Fig. 5, variable v_1 is accessed by e_i and e_j, and thus v_1 is a shared variable. In Fig. 2, variable *shoppingLists* declared in Line 2 is a shared variable, and it is accessed by events {3, 14, 17, 18, 21, 27, 30}. Note that, the variable aliases can complicate our analysis, we will explain the solution in Section IV.B.

We collect the following context information during the execution of the original event trace. (1) For each event *e*, we collect its shared variable use set (i.e., what variables it uses). (2) For each shared variable *v*, we collect the event that declares *v*. (3) For each read of shared variable *v* or field access that is reachable from *v*, we collect its value type and a history of events that write to *v*. The value types can be: "undefined", "number", "string", "boolean", "function", and "object". Specially, for a DOM object, we mark it as "DOM" type instead of the general type "object"; for an object with value *null*, we mark it as "null" type. Note that, the "object" is a general type. Shared variables in "object" type can have complicated data structure, e.g., it may contain multiple fields, and the fields may be in "object" type again. Regarding this, we also collect the type information of fields that are reachable from the shared variable. To simplify presentation in this paper, we consider that a reachable field *f* from the shared variable *o* as a shared variable, and denote it as variable *o.f*.

B. Constraints for Candidate Subtraces

Given an event trace $\tau = \{e_1, e_2, \ldots, e_n\}$ that leads to a failure *f*, our approach aims to generate a minimum event subtrace τ' of τ which is failure-reproducible. The trace τ' should be as short as possible. For each event in the event trace τ, we use a bit variable s_i to denote whether its corresponding

event e_i is selected by τ'. If e_i is selected by τ', then $s_i = 1$, otherwise, $s_i = 0$. Then, the constraint to select an event e_i is

$$select(e_i) \equiv (s_i == 1)$$

The trace generating formula Φ is constructed by a conjunction of three sub-formulas: $\Phi(n) \equiv \varphi_c \wedge \varphi_l(n) \wedge \varphi_e$. Specifically, φ_c requires that each event in τ' has the compatible context with its corresponding event in the original trace τ. $\varphi_l(n)$ restricts the length of τ', and requires that only *n* events can be selected. Φ_e requires that the failure events must be selected. We introduce these constraints and their encodings in detail in the following subsections.

1) Compatible Context Constraint (φ_c)

Since some events in the original event trace may be deleted, the context of an event in a candidate subtrace may be different from its corresponding one in the original event trace. For an event e_i in the original event trace, if it is selected (marked as e_i') in the candidate event subtrace, we say that e_i and e_i' have *compatible context* if the following two conditions are satisfied: (1) All the variables used by e_i and e_i' are declared in the same scope with respect to their corresponding traces (φ_{c1}). (2) All the variables used by e_i and e_i' have the same type with respect to their corresponding event traces (φ_{c2}). We use *compatible context constraint* (φ_c) to make sure that all corresponding events in the original event trance and a candidate subtrace have compatible context. Formally, (φ_c) is

$$\varphi_c \equiv \varphi_{c1} \wedge \varphi_{c2}$$

The formula φ_{c1} is designed to guarantee the first condition. Specifically, it requires that if an event e is selected, then for each shared variable used by e, its declaring event should also be selected. In JavaScript, a variable can be explicitly declared using the keyword *var*. For example, every time the statement "*var v*" is executed, we regard it as a declaration to the variable *v*. Let $use(e)$ be the set of variables used by e, $dec(v)$ be the event that declares variable v. Formally, φ_{c1} is

$$\varphi_{c1} \equiv \wedge_{e \in \tau}(select(e) \Rightarrow \wedge_{v \in use(e)} select(dec(v)))$$

For example in Fig. 3. Event 4 reads variable *newList* (Line 29 in Fig. 2) that is declared by event 3 (Line 23 in Fig. 2), so we build a constraint $select(e_4) \Rightarrow select(e_3)$.

JavaScript has static scoping, so in some cases, the declaration of a variable during the execution of an event handler may be out of scope in all later event handlers. For example, in Fig. 2, the event handler (*onclick*, Line 28) of event 4 declares the variable *items* (Line 29) and then uses *items* later (Line 30). The event handler of event 5 performs the same operations. However, the two variables *items* used in events 4 and 5 are different. Thus, we should not build constraints among these two events.

The formula φ_{c2} is designed to guarantee the second condition, which ensures that each variable reads values in the same type as the one in the original trace, although their exact value may be different. Specifically, φ_{c2} requires that if an event e is selected, then for each variable *v* used by event e, at least one of the events that write to *v* in the same type is selected. For a variable *v* that is read by event *e*, we use $\boldsymbol{Events(e, v)}$ to denote the **type-compatible events** of *v*, which write to *v* in the same type. Formally, φ_{c2} is

$$\varphi_{c2} \equiv \wedge_{e \in \tau} select(e) \Rightarrow \wedge_{v \in use(e)}(\vee_{e_j \in Events(e,v)} select(e_j))$$

$Events(e, v)$ is calculated by comparing the collected type information. We give more details about how we obtain $Event(e, v)$ for JavaScript and DOM variables in Section IV.

Consider our example in Fig. 3. For general variable usage, we build a constraint $select(e_{30}) \Rightarrow (select(e_0))$ since event 30 read variable *shoppingLists*, which is modified by event 0. Similarly, we build a constraint $select(e_{30}) \Rightarrow (select(e_3) \vee select(e_{17}) \vee select(e_{21}))$. First, the field *curList.totalCount* that event 30 read at Line 46 is modified by event 3 (which modifies this field at Line 52 when a new shopping list is created at Line 23). Second, events 17 and 21 (which modifies this field at Line 46 when a new item is added to the current shopping list.). For DOM variable, we build a constraint $select(e_3) \Rightarrow (select(e_1) \wedge select(e_2))$, since event 3 read the DOM elements (i.e., *newlist-form* at Line 9 and *listName* at Line 22) that are modified by events 1 or 2. Similarly, for event 30, we build a constraint $select(e_{30}) \Rightarrow (select(e_{27}) \wedge select(e_{28}) \wedge select(e_{29}))$ since it read the DOM elements (i.e., *name* at Line 36, *count* at Line 37) that are modified by events 27, 28 and 29, respectively.

2) Length Constraint ($\varphi_l(n)$)

Since the length of the failure-related events is usually short [16][17], we generate candidate event subtraces from short to long. Length constraint is used to restrict the number of events that can be selected. Let n be the required specific length of candidate event subtraces. Formally, $\varphi_l(n)$ is

$$\varphi_l(n) \equiv (\sum_{ei \in \tau} s_i) == n$$

3) Failure Event Constraint (φ_e)

Failure event constraint (φ_e) is used to select the erroneous events that trigger the failure *f*. By selecting the erroneous events, we make sure only the events that affect the contexts of the erroneous event are selected. One of the following information is required to be given by the developers. (1) The event that the failure lies in, e.g., the event that throws an exception. (2) Assertions about the failure symptoms. Similarly, let $asserts$ be the set of events that contain assertion statements, and *fails* be the set of failure events. Formally, φ_e is

$$\varphi_e \equiv select(asserts) \vee select(fails)$$

C. Event Subtrace Generating and Validating

Our approach iteratively generates candidate event subtraces until a failure-reproducible event subtrace is found. The event subtrace generation algorithm is as follows. (1) We resolve $\Phi(n)$ using a SAT solver (e.g., z3 [18][19]). Initially, *n* is 1. (2) There may be multiple solutions for $\Phi(n)$ (i.e., there may be multiple candidate event subtraces with length *n*). Thus, we need to generate all possible solutions for length *n*. Unfortunately, a general SAT solver only provides a single solution for a given constraint. We use the technique similar as [20] to recursively find all the solutions. For each solution s that is provided by the SAT solver, we create a blocking clause c to constrain that the new solution s' is different from s. By resolving the constraint ($\Phi(n) \wedge c$), we get a new solution. This process is repeated until there is no solution. (3) If there is no solution, increase *n* to *n*+1 and then go to step 1.

Example. Consider our example in Fig. 3. After we generate all constraints, we feed the constraint $\Phi(n)$ (n is initially 1) to the constraint solver. There is no solution, since there is no 1-length-subtrace that can satisfy $\Phi(n)$. Until n is increased to 8, we can get the first solution: events 0-3 and 27-30 (n=8). However, this solution cannot reproduce the failure. We continue the above process and get another solution event 0-3, 14-17 and 27-30 (n=12), the process is terminated since this new solution can reproduce the failure.

Discussion. Our event subtrace generation process is similar to delta debugging [5][8]. The key difference is how to generate candidate event subtraces. Delta debugging iteratively and blindly (i.e., without knowing the context of an event) cuts the trace to subtraces with some granularity and tries to validate them. Our approach can accelerate this process due to the following two reasons: (1) Among the concerned events, we guide the subtrace generation by selecting events that have the compatible contexts with the ones in the original trace. So that we can generate shorter subtraces quickly. (2) Our approach does not search the whole trace. We only care about the events that affect the contexts of the erroneous events.

IV. IMPLEMENTATION

Our proposed approach is implemented as EvMin for client-side JavaScript applications. It is also applicable for other kinds of event-driven applications, such as Android or Java GUI. Note that, the compatible context constraints may not be the same for other kinds of event-driven applications. EvMin consists of four components: record/replay, information collector, trace generator and trace validator.

A. Record and Replay

We adopt a similar approach as Mugshot [1] to implement record and replay. Mugshot can record low-level events as an event trace when users browse a web site, and replay the execution based on a given event trace. It handles the non-determinism, such as random values, system times and dynamic data that are retrieved from the server, so that it can replay the original trace deterministically.

B. Information Collector

The information collector is a proxy server that instruments the source code. It uses the record and relay component to replay the recorded event trace, and collect context information to form the constraints (Section III.B). We explain how we collect the context information for each shared variable v as follows.

The event that declares v. We intercept all statements that declare variables using keyword "var". For example, every time the statement "*var v*" is executed, we regard it as a declaration to variable v. If v is declared by event e, then $dec(v)$ is mapped to e. Note that, although we simply treat a field f in an object variable o as a new variable, we only consider read and write to f, and do not generate $dec(f)$ for f.

The type of v. Obtaining type information is straightforward. We intercept the read and write operations to shared variables and record their value types. Specially, if the value is an instance of DOM object, we mark it as "DOM" type, and if the value of an object is *null*, we mark it as "null" type.

A history of events that modify v. For each variable v, we maintain a list of events that have written to v, so that we can find out the type compatible events for v.

The use set of event e. Every variable that is read by the handler of event e is added to $use(e)$.

Alias analysis. Obtaining the above information becomes complicated in the presence of aliases in JavaScript. When computing the events that define or modify v, we need to consider possible aliases that may refer to the same object in memory. The prevalent use of aliases and *dot notation* (e.g., accessing field *foo* of object o: $o.foo$) for accessing a variable in JavaScript often complicates the issue of code comprehension. Static analysis techniques often ignore this issue [21]. However, it is not a big issue for dynamic analysis. For variables accessed by names, we maintain runtime scopes (variables are stored in different scopes according to where they are defined). We denote each access of such variable v as (*object*, *name*) where the *object* is the scope of v and *name* is the accessed name of v. For a variable accessed by *dot notation* (e.g., $o.foo$), we use the same denotation (*object*, *name*) where *object* is the object reference (e.g., o) and *name* is the field (e.g., *foo*) that is accessed. When judging if two operations access the same variable, we check if their *objects* point to the same memory address and their *names* are equal at runtime.

C. Trace Generator

Constructing constraints. Trace generator automatically encodes context information into constraints according to Section III.B. It is straightforward to construct these constraints. However, it is challenging to obtain the type-compatible events (i.e., $Events(e, v)$) for DOM variables. In the following, we present how we obtain type-compatible events for general JavaScript variables and DOM variables, respectively. For any read of v in event e_i.

1) v is a general JavaScript variable

In this case, we directly compare the type information. Specifically, for any earlier event e_k which writes to v, if the type of v in event e_k is the same as the type of v in event e_i, then e_k is included in $Events(e, v)$.

2) v is a DOM variable

DOM variables are operated by native APIs. For example, using *appendChild* API to append a child to a DOM element, and *remove* API to remove a DOM element. The usage of these APIs are complicated, for example, the same DOM element can be operated via several ways, such as *getElementbyId*, *getElementbyClass*, *querySelector* or directly reading the children of its parent. Thus, analyzing the type-compatible modifications to a DOM element is not easy. Simply treating all events as type-compatible is impractical. Instead, our observation is that the DOM is changed to a new state after executing each event. If every accessed DOM element *ele* in event e_i also exists in the result DOM state updated by event e_k ($k < i$), then e_k is a type-compatible events with respect to *ele*.

We use Fig. 6 to explain how it works. Suppose that there is an event trace that contains 4 events: Event e_1 loads the

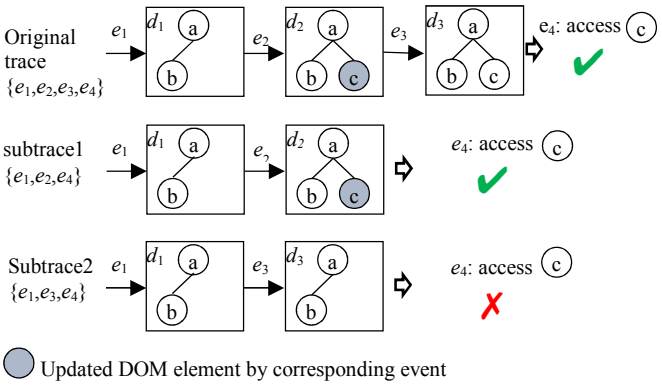


Fig. 6. An example for resolving type-compatible event for DOM variable. *Event e_1 loads the initial DOM tree, event e_2 adds an element c to the DOM tree, event e_3 does nothing to the tree, and event e_4 accesses the added DOM element c.*

initial DOM tree, event e_2 adds an element c to the DOM tree, event e_3 does nothing to the DOM tree, and event e_4 accesses the added DOM element c. For the original trace $\{e_1, e_2, e_3, e_4\}$, we cache the DOM state after the execution of each event. Suppose that the cached DOM states are $\{d_1, d_2, d_3\}$ before the execution of e_4. Under the current DOM state, its accessed DOM element c is available. If we determine type-compatible events of c, they would be e_2 and e_3, since their resulting DOM states (i.e., d_2 and d_3) contain the DOM element c. According to this type-compatible event information, our constraints may let e_4 read from e_2 and generate subtrace1: $\{e_1, e_2, e_4\}$. This subtrace is feasible, since the existence of the DOM element accessed by e_4 keeps the same as the original trace.

Generally, we use the following approach to obtain the type-compatible events for a DOM access. DOM states are cached as $D = \{d_1 \dots d_{i-1}\}$ before the execution of event e_i. d_{i-1} is the DOM state after the execution of event e_{i-1} and before the execution of event e_i. When executing event e_i, we intercept all the operations that perform on the DOM. For each intercepted DOM operation, we calculate the *selector* of accessed DOM element *ele* (see next paragraph for the algorithm of *getSelector*). And then we use the *selector* to select DOM elements on each cached DOM state in D, and compare R_k and *ele* (R_k is the selected element on d_k ($1 \leq k < i$)). If the following conditions are satisfied, then e_k is regarded as a type-compatible event for DOM element *ele*: both R_k and *ele* are empty or neither of them is empty. Note that, only comparing the existence of a DOM element is enough. If the accessed DOM element has complicated DOM structure, then as long as its child elements are accessed, our approach is able to intercept that operation and resolve the existence of the child elements.

We need to calculate a selector s for a DOM access in DOM state so that we can check the existence of the accessed element with the selected DOM element on each cached DOM state. Our algorithm (*getSelector*) to calculate a selector which keeps enough information to precisely locate a DOM element. The algorithm is described as follows: (1) If the DOM access API is *querySelector*, the algorithm returns the selector directly. (2) If the DOM access API is *getElementById*, *getElementsByClass*, or *getElementByTagName*, we calculate the XPath [22] of current element as shown in (3), and also record corresponding id, class or tagName information. (3) Otherwise, DOM is not accessed by common APIs. In this case, we calculate the XPath of the current element, but ignore the index information since it easily makes a selector broken. For example, the DOM access *document*.getElementById('name') (Line 36 in Fig. 2) is calculated as //input[@id='name']. The //input is the calculated XPath according to (3), and the [@id='name'] is the id information of the element according to (2).

Our DOM model is not completely precise (it does not always generate feasible event traces). For example, our constraints may let e_4 read from its type-compatible event e_3, and generated subtrace2: $\{e_1, e_3, e_4\}$. If we run subtrace2, we will find it infeasible, since the DOM element accessed by e_4 does not exist. This imprecision may cause our approach to explore more candidate subtraces. We observe that event e_3 has nothing to do with DOM element accessed by e_4 and thus there is no need to select e_3. Specifically, we further make the following improvement for the DOM model: For each DOM element accessed by event e_i, we record the history of events E that update it. Only the events that are in E can be regarded as type-compatible events of e_i.

Resolving constraints. We use z3py, a python front end for constraint solver z3 [18][19], to encode our constraints described in Section III.B to z3 and retrieve the result.

D. Trace Validator

Since we do not care about the concrete values of each shared variable, the candidate event subtraces we generate may not be able to reproduce the failure. Therefore, as delta debugging does, our trace validator validates a candidate subtrace by running it in a browser and observes if the failure can be reproduced. The trace validator starts a browser and uses the record-replay component to replay a candidate subtrace and checks if the candidate subtrace is failure-reproducible (i.e., the failure assertions are satisfied). It terminates if it can trigger the failure. Otherwise, it invokes the trace generator to obtain the next candidate subtrace. Our constraints avoid generating infeasible traces in most cases. But if a generated subtrace is infeasible, we do not care about if the subtrace can be replayed properly, and just dispatch the recorded events and check if the assertions are satisfied.

V. EVALUATION

We evaluate EvMin by answering the following three research questions:

RQ1: Can EvMin effectively remove failure-irrelevant events?

RQ2: What is the performance of EvMin?

RQ3: How is EvMin compared with existing approaches, e.g., delta debugging?

A. Experimental Subjects

We use two datasets to evaluate EvMin:

Dataset-1. We reused the subjects that were used by the delta debugging approach [5]. There are totaling 30 subjects in their project. We removed a subject if it satisfied one of the following conditions: (1) The number of failure related events

TABLE I. EXPERIMENTAL SUBJECTS

StackOverflow failures (Dataset-1)			
ID	Application	LOC	Error type
1	CanadaLong	105	Incorrect values
2	OnlineShoppingLong	30	Cannot add items to cart
3	AgeCaculate	114	Invalid calculation
4	CarRental	125	Unresponsive DatePicker
5	InsuranceLong	93	Form submitted with empty field
6	StudentInfo	92	Invalid input
7	Airport	44	Invalid input
8	BestCars	38	Invalid input
9	Game	68	Faulty button click
10	Numbers	118	Incorrect calculation
11	Patient form	93	Form submitted with empty field
12	Patient form	74	Invalid input

Real-world failures (Dataset-2)			
ID	Application	Size	Popularity
13	Chart.js-1[39]	105K	14,803
14	Chart.js-2[40]	105K	14,803
15	HandsonTable[41]	4.7M	4,989
16	JPushMenu[42]	1.5M	134
17	Todolist[43]	312K	19
18	FullPage[44]	882K	9,518
19	Editor.md[45]	257K	530
20	My-mind[46]	223K	1,449
21	Foundation-sites[47]	576K	22,885
22	Reveal.js[48]	424k	26,893

(including the failure event) is 1. We regard such cases as non-representative since the developers can diagnose the failure by just inspecting the single erroneous event. We prior to selecting complicated failures that are usually caused by multiple events. (2) The failures cannot be reproduced. Finally, we obtained 12 failures along with their failure revealing traces.

The upper part in Table I shows the details about the 12 failures. The involved applications are relatively small and one might argue that they do not reflect the typical JavaScript application failures. However, they covers different types of failures and were used by the existing delta-debugging approach [5]. Further, we can evaluate EvMin and compare EvMin with the existing approach on this dataset.

Dataset-2. In order to evaluate EvMin on real-world failures, we chose another 10 real-world failures from GitHub that were used by JSTrace [10]. These JavaScript application failures were selected by the criteria that they are marked as bugs, written in JavaScript language, have at least two comments and explicit reproducing descriptions. The lower part in Table I shows the details of these selected 10 real-world failures, in which, *Size* denotes the file size of JavaScript source code, and *Popularity* denotes the numbers of stargazers of the project. These involved applications are designed for different purposes and functionality, and have high popularity (weighed by the number of stargazers in GitHub). These failures are complex (multiple steps are necessary to trigger the failures).

To get the failure revealing traces for these 10 real-world failures, we first used their applications for a while, and then followed the failure reproduction steps in the bug reports to reproduce the failures. Then we used our record and replay component to record these traces. To evaluate the reduced event traces, we manually identified the expected minimal failure-inducing events in the original trace that are critical to trigger the failure.

TABLE II. RESULT ON DATASET-1

ID	Original trace		Delta debugging		EvMin	
	#All	#Expected	#Reduced	#Subtrace	#Reduced	#Subtrace
1	17	2	2	7	2	2
2	10	2	2	9	2	2
3	9	3	3	6	3	1
4	11	2	2	6	2	1
5	10	2	2	6	2	1
6	9	2	2	10	2	1
7	6	2	2	10	2	2
8	4	2	2	3	2	2
9	13	2	2	14	2	1
10	8	2	2	12	2	1
11	8	2	2	11	2	1
12	9	2	2	12	2	1
Avg.				8.8		1.3

B. Experimental Setup

To answer RQ1, we evaluated *EvMin* on Dataset-1 and Dataset-2, respectively. We check how close the final solutions are to the expected minimal events and give an analysis to the experimental result.

According to our preliminary experiment, we found that the reduced event traces of EvMin are very close to the minimal failure-inducing trace, but not minimal. We further dug into why some irrelevant events cannot be removed. The reason is as follows: The use of native JavaScript calls can cause our collected history set of events that modifies a variable incomplete (the earlier part is missing), thus our compatible context constraint cannot find an event and some irrelevant events are not removed. We have fixed this case by modeling most commonly used APIs, but not all of them. To further address this issue, we propose an *optimized EvMin*, in which we further use delta debugging to remove the remaining irrelevant events after we have used *EvMin*. In the last, we evaluate the *optimized EvMin* by checking whether it can reduce the original trace to minimal.

To answer RQ2, we evaluated how many subtraces EvMin has to explore until a failure-reproducible one is found. In addition, we evaluated the time taken to find the final solution. Since EvMin first collects context information, and then generates one candidate subtrace each time and validates it, the total time is comprised by context collecting, trace generating, and trace validating. We answer this question by evaluating the total time. We give statistics on the 3 parts respectively and explain the experimental result.

To answer RQ3, we compared EvMin with delta debugging in terms of the number of generated subtraces, the length of reduced trace and time. To compare with delta debugging [5], we implemented their algorithm and evaluated it on our experimental subjects.

Our experiments were performed on a 64-bit machine with 8G memory. In Table II and Table III, category *Original trace*

TABLE III. RESULT ON DATASET-2

ID	Original trace		Delta debugging			EvMin			EvMin with DD		
	#All	#Expected	#Reduced	#Subtrace	Time(s)	#Reduced	#Subtrace	Time(s)	+Subtrace	+Time	Total(s)
13	1051	5	5	156	1205	6	5	38	11	21	59
14	1189	5	5	147	1637	7	102	268	16	27	295
15	694	5	5	227	2771	12	18	126	13	18	144
16	342	2	2	38	150	2	1	9	3	7	16
17	1410	3	3	74	514	7	14	393	14	19	412
18	398	3	3	62	752	8	1	26	10	26	52
19	1326	2	2	48	702	8	7	165	11	34	199
20	1290	6	6	116	1034	8	1	162	7	22	184
21	367	2	2	23	148	2	6	63	7	14	77
22	617	5	5	92	1639	11	1	73	21	97	170
Avg.	868	2.7	2.7	98	1055	7.1	16	132	11	29	161

TABLE IV. TIME STATISTICS FOR EVMIN

ID	Collect info		Gen trace		Validate	
	Time(s)	%Time	Time(s)	%Time	Time(s)	%Time
13	24	63.2%	3	7.9%	11	28.9%
14	20	7.5%	21	7.8%	227	84.7%
15	51	40.5%	11	8.7%	64	50.8%
16	6	66.7%	1	11.1%	2	22.2%
17	343	87.3%	13	3.3%	37	9.4%
18	18	69.2%	2	7.7%	6	23.1%
19	152	92.1%	2	1.2%	11	6.7%
20	129	83.6%	5	3.1%	28	17.3%
21	42	66.7%	12	19.0%	9	14.3%
22	61	83.6%	2	2.7%	10	13.7%
Avg.	84.6	63.9%	0.5*	5.4%	2.6*	30.6%

* Average time cost per subtrace

shows the information about the original trace that reveals a failure. The columns *All* and *Expected* represent the length of the original trace and expected minimal number of events to reproduce the failure. The category *Delta debugging, EvMin*, *EvMin with DD* shows the reduction result of delta debugging, EvMin, and optimized EvMin with delta debugging, respectively. The corresponding columns *Reduced, Subtrace,* and *Time* present the length of reduced result, the number of subtraces to explore the solution and time needed.

C. Results and Analyses

1) RQ1: Can EvMin effectively remove failure-irrelevant events?

Table II shows EvMin's reduction result on Dataset-1. As we can see, EvMin can reduce the trace to minimal, which means all the irrelevant events are removed. Since most of failures in Dataset-1 have simple data flow and most of the events are user-input events, EvMin performs well.

The right part of Table III shows EvMin's reduction result on Dataset-2. The length of reduced event traces is only on average 2.1X of the minimal failure-inducing trace.

EvMin with DD. EvMin alone can remove most of the irrelevant events. The remaining irrelevant events can be further removed by delta debugging. The *EvMin with DD* column in Table III shows how many additional subtraces, additional time and total time needed to reduce the trace to minimal. As a result, optimized EvMin with DD can remove all irrelevant events with little cost.

2) RQ2: What is the performance of EvMin?

Event trace reduction. We use the number of subtraces to evaluate how many traces EvMin explores before the failure-reproducible subtrace is found. The *Subtrace* column under EvMin category in Table II shows this result on Dataset-1. Since most of failures in Dataset-1 have simple data flow and most of the events are user-input events, our constraints on the data flow (Section III.B) can quickly find a failure-reproducible solution within 1 or 2 subtraces.

For Dataset-2, as shown in Table III, EvMin tests 16 subtraces on average. For some failures, the failure-reproducible subtrace could be found in the first time, such as failures 18 and 22. While some failures need to try many subtraces, such as failure 14. It is because a variable may have many type-compatible values that result in many solutions.

Time overhead. We use the execution time to evaluate whether EvMin can quickly reduce an event trace. Table III shows the result for each subject in Dataset-2 (column Time under EvMin category). As a result, EvMin takes 132 seconds, and EvMin with *DD* takes 161 seconds on average.

The execution time of EvMin consists of three parts: collecting information, trace generating, and trace validating. We give the time cost for each part to further analyze the time overhead. They are showed under columns *collect info*, *gen trace*, *validate* respectively in Table IV. (1) Collecting information takes on average 63.9% of the time. It mainly comes from the overhead of instrumentation that aims to collect context information at runtime. As we can see in the table, the time for collecting information for each subject varies from 6 to 343 seconds. This is determined by the complexity of the application or the user operations (especially frequent UI operations on heavier applications). If the application has complicated data dependences, our trace collector may spend much time finding out the history of events that modified a variable of interest (especially for DOM variables). (2) The trace generating time is used for constraint solving, and it takes 5.4% of the whole time. This is determined by the performance of z3 and the number of subtraces EvMin need to generate. When we focus on each subtrace, it takes only on average 0.5 seconds. This value is not high and shows that the constraint solver is good at resolving our constraints. The reason is that each event variable in our constraints has two states: 1 for selected or 0 for not selected. We also add a length constraint (Section III.B.2) to generate candidate subtraces, which means there are exactly *n* events hold value 1. (3) The trace validating takes 30.6% of the time. Each subtrace takes on average 2.6 seconds. This value is not high because our policy to generate candidate is from short to long, and all our tried subtraces are no longer than the length of the failure-reproducible trace (which is on average 7.1 as shown in Table III).

The overhead of EvMin with DD consists of two parts, event trace reduction with EvMin, and further reduction with *DD*. They take on average 82% and 18% of the total time, respectively. This result shows that, EvMin can be prioritized to reduce the original trace to minimal with little cost.

3) RQ3: How is EvMin compared to existing tools, e.g., delta debugging?

On Dataset-1 showed in Table II, both EvMin and delta debugging can find the minimal failure-inducing events. However, EvMin generates 70% fewer subtraces compared to delta debugging.

On Dataset-2, both EvMin with DD and delta debugging can reduce the trace to minimal. However, as Table III shows, EvMin generates 84% fewer subtraces and takes 86% less time compared with delta debugging, and EvMin with DD generates 72% fewer subtraces and takes 84% less time compared to delta debugging. Although EvMin takes some time to collect runtime information, EvMin avoids exploring infeasible subtraces and the length of each subtrace is relatively short (short subtraces are firstly explored until the target subtrace is found). While delta debugging is on the opposite, it may generate many infeasible subtraces and the subtraces are relatively long (long subtraces are firstly explored until the target subtrace is found).

VI. DISCUSSION

Limitations of EvMin. (1) Our implementation of calculating type-compatible events for DOM is imprecise. This may calculate $Events(e, v)$ as the superset of the precise one, and let EvMin generate more candidate subtraces. Another problem is that it takes much memory. We leave it as future work to improve the precision and memory overhead. (2) As for the experimental result, one might argue EvMin alone cannot reduce the trace to minimal, this is mainly because our implementation does not model all the native APIs. Our reduction result is already close to the minimal one, and the performance is greatly improved. In addition, we improve it to further reduce the remaining irrelevant events using delta debugging with little time overhead.

Threats to validity. A threat to our evaluation is that only 10 real-world failures are evaluated. However, the selected applications are developed for different purpose and have high popularity. The selected failures need multiple steps to be reproduced. This implies they have reasonable complexity.

VII. RELATED WORK

We focus on the work that concern event trace reduction in JavaScript applications, delta debugging-based fault isolation, and program analysis utilizing program dependence.

Event trace reduction in JavaScript applications. Hammoudi et al. [5] adapt delta debugging [8] to reduce event traces for JavaScript applications. The algorithm repeatedly selects subtraces of the events, and replays them to determine whether these subtraces can, by themselves, reveal the failure. Their work relies on trial-and-error and does not care the relationship among events, while EvMin utilizes the context information to generate candidate subtraces, and by this way greatly narrows down the search space. JSTrace [10] adopts dynamic slicing [11][12][13] to trace the precise program dependence and remove the events that are not depended by a failure. However, not all remaining events are necessary for reproducing the failure.

Delta debugging-based fault isolation. Delta debugging is commonly used for trace minimization and fault isolation. Andreas proposes a series of delta debugging algorithms for simplifying failure inducing input [8], minimizing reproduction [23], failure-inducing thread schedules [24], and isolating cause-effect chains [7]. HDD [9] is proposed to speed up delta debugging by using the hierarchical structure of the input, such as XML, AST of a program. However, HDD does not help our case since the input of our approach is not structured. The hill climbing approaches [25][26] aim to generate test cases as diverse as possible, and do not help for our purpose. GUI event trace minimizing [27] on Android proposes a variant of delta debugging [6][8] to find smaller event traces that reach a desired activity with high probability.

Program analysis utilizing program dependence. Sriraman et al. [28] identify failure-irrelevant threads by analyzing the program's control and dataflow dependence. Lots of dynamic slicing approaches [29][30] are proposed for slicing programs. However, they face the problem that a computation may still be redundant even if it has data or control dependence to the buggy state. LEAN [31] proposes two redundancy criterions that characterizes the redundant computation in a buggy trace to simplify the concurrency buggy execution. But their redundant criterions cannot help us remove failure irrelevant events. Thin slicing [32] heuristically omits some data dependences to reduce a slice since not all statements that may affect a point of interest appear equally relevant to a human. As for observation-based slicing [33], its concept of moving deletion window is meaningless in our case, since the deletion unit in a trace for JavaScript applications is one event. Thus, the algorithm will degrade to delete one event at a time from an event trace. SimpleTest [34] presents a technique that simplifies tests at the semantic level by repeatedly replacing referred expressions in each statement with other alternatives from the test code itself. Some GUI testing tries to reduce test suites by applying event dependence analysis [35], symbolic execution [36], program slicing [37] or partial order reduction [38], to improve their performance. Different from these work, our approach only requires compatible context instead of precise program dependence.

VIII. CONCLUSION

In this paper, we propose a novel approach to remove failure-irrelevant events in an event trace for client-side JavaScript applications. To find the minimal failure-inducing events efficiently, we build constraints among events, and require that each event's context is respected before and after reduction. Thus, we can avoid generating syntactically infeasible candidate event subtraces. We have implemented our approach as a tool, *EvMin*. The evaluation on 10 real-world client-side JavaScript application failures shows that EvMin can efficiently remove failure-irrelevant events. Compared with delta debugging, EvMin generates 72% fewer candidate event subtraces and takes 84% less time overhead. In the future, we plan to apply our approach on Android applications, which usually generate long event traces, too.

ACKNOWLEDGMENT

This work was supported by National Key Research and Development Plan (2016YFB1000803), National Natural Science Foundation of China (61672506, 61732019, 61702490), Key Research Program of Frontier Sciences, CAS (Grant No. QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

REFERENCES

[1] J. Mickens, J. Elson, and J. Howell, "Mugshot : Deterministic Capture and Replay for JavaScript Applications," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation(NSDI)*, 2010, pp. 159–174.

[2] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive Record/Replay for Web Application Debugging," in *Preceedings of User Interface Software and Technology (UIST)*, 2013, pp. 473–484.

[3] S. Alimadadi, S. Sequeira, A. Mesbah, and K. Pattabiraman, "Understanding JavaScript Event-based Interactions," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2014, pp. 367–377.

[4] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript," in *Proceedings of the Joint Meeting on Foundations of Software Engineering(ESEC/FSE)*, 2013, pp. 488–498.

[5] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the Use of Delta Debugging to Reduce Recordings and Facilitate Debugging of Web Applications," in *Proceedings of Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software (ESEC/FSE)*, 2015, pp. 333–344.

[6] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?," in *Proceedings of the European software engineering conference held jointly with the ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, 1999, vol. 24, no. 6, pp. 253–267.

[7] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[8] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 183–200, 2002.

[9] G. Misherghi and Z. Su, "HDD: Hierarchical Delta Debugging," in *Proceedings of the International Conference on Software Engineering(ICSE)*, 2006, pp. 142–151.

[10] J. Wang, W. Dou, C. Gao, and J. Wei, "Fast Reproducing Web Application Errors," in *Preceedings of International Symposium on Software Reliability Engineering(ISSRE)*, 2015, pp. 530–540.

[11] B. Korel and J. Laski, "Dynamic Program Slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[12] X. Zhang, N. Gupta, and R. Gupta, "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults," *Empirical Software Engineering (ESE)*, vol. 12, no. 2, pp. 143–160, 2007.

[13] R. Gopal, "Dynamic Program Slicing Based on Dependence Relations," in *Proceedings of the International Conference on Software Maintenance(ICSM)*, 1991, pp. 191–200.

[14] "Document Object Model (DOM) Level 3 Events Specification." [Online]. Available: http://www.w3.org/TR/2011/WD-DOM-Level-3-Events-20110531/.

[15] F. S. Ocariza, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic Fault Localization for Client-side JavaScript," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 69–88, 2016.

[16] G. Li, E. Andreasen, and I. Ghosh, "SymJS: Automatic Symbolic Testing of JavaScript Web Applications," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE)*, 2014, pp. 449–459.

[17] A. J. Ko and B. A. Myers, "Extracting and Answering Why and Why Not Questions about Java Program Output," *ACM Transactions on Software Engineering and Methodology(TOSEM)*, vol. 20, no. 2, pp. 1–36, 2010.

[18] "Z3." [Online]. Available: https://z3.codeplex.com/.

[19] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems(TACAS)*, 2008, pp. 337–340.

[20] K. Bajaj and A. Mesbah, "Synthesizing Web Element Locators," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 331–341.

[21] A. Feldthaus, M. Schafer, M. Sridharan, J. Dolby, and F. Tip, "Efficient Construction of Approximate Call Graphs for JavaScript IDE Services," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2013, pp. 752–761.

[22] "XML Path Language (XPath)." [Online]. Available: https://www.w3.org/TR/xpath/.

[23] M. Burger and A. Zeller, "Minimizing Reproduction of Software Failures," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2011, pp. 221–231.

[24] J.-D. Choi and A. Zeller, "Isolating Failure-Inducing Thread Schedules," in *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis(ISSTA)*, 2002, pp. 210–220.

[25] A. Marchetto and P. Tonella, "Using Search-based Algorithms for Ajax Event Sequence Generation During Testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.

[26] H. Dan, M. Harman, J. Krinke, L. Li, A. Marginean, and F. Wu, "Pidgin Crasher: Searching for Minimised Crashing GUI Event Sequences," in *Symposium on Search-Based Software Engineering(SSBSE)*, 2014, pp. 253–258.

[27] Q. Zhang and B. Goncalves, "Minimizing GUI Event Traces," in *Preceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering(FSE)*, 2016, pp. 422–434.

[28] S. Tallam, C. Tian, X. Zhang, and R. Gupta, "Enabling Tracing Of Long-Running Multithreaded Programs Via Dynamic Execution Reduction," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2007, pp. 27–218.

[29] J. Krinke, "Context-Sensitive Slicing of Concurrent Programs," in *Proceedings of the European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2003, pp. 178–187.

[30] D. Giffhorn and C. Hammer, "Precise Slicing of Concurrent Programs: An Evaluation of Static Slicing Algorithms for Concurrent Programs," in *Proceedings of International Conference on Automated Software Engineering(ASE)*, 2009, vol. 16, no. 2, pp. 197–234.

[31] J. Huang and C. Zhang, "LEAN: Simplifying Concurrency Bug Reproduction via Replay-supported Execution Reduction," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications(OOPSLA)*, 2012, pp. 451–466.

[32] M. Sridharan, S. J. Fink, R. Bodik, M. Sridharan, S. J. Fink, and R. Bodik, "Thin Slicing," *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 112, 2007.

[33] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo, "ORBS: Language-Independent Program Slicing," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 109–120.

[34] S. Zhang, "Practical Semantic Test Simplification," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 1173–1176.

[35] S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon, "Lightweight Static Analysis for GUI Testing," in *Proceedings of International Symposium on Software Reliability Engineering(ISSRE)*, 2012, pp. 301–310.

[36] L. Cheng, J. Chang, Z. Yang, and C. Wang, "GUICat: GUI Testing as a Service," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering(ASE)*, 2016, pp. 858–863.

[37] S. Arlt, A. Podelski, and M. Wehrle, "Reducing GUI Test Suites via Program Slicing," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 270–281.

[38] P. Maiya, R. Gupta, A. Kanade, and R. Majumdar, "Partial Order Reduction for Event-Driven Multi-threaded Programs," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of System (TACAS)*, 2016, pp. 680–697.

[39] “Chartjs issue 503.” [Online]. Available: https://github.com/nnnick/Chart.js/issues/503.

[40] “Charjs issue 920.” [Online]. Available: https://github.com/nnnick/Chart.js/issues/920.

[41] “HandsonTable issue 638.” [Online]. Available: https://github.com/handsontable/handsontable/issues/638.

[42] “JPushMenu issue 1.” [Online]. Available: https://github.com/takien/jPushMenu/issues/1.

[43] “TodoList.” [Online]. Available: https://github.com/01org/webapps-todo-list.

[44] “Fullpage issue 146.” [Online]. Available: https://github.com/alvarotrigo/fullPage.js/issues/146.

[45] “Editor.md issue 18.” [Online]. Available: https://github.com/pandao/editor.md/issues/18.

[46] “My-mind issue 12.” [Online]. Available: https://github.com/ondras/my-mind/issues/12.

[47] “Foundation-site issue 7528.” [Online]. Available: https://github.com/zurb/foundation-sites/issues/7528.

[48] “Reveal issue 463.” [Online]. Available: https://github.com/hakimel/reveal.js/issues/463.