Constraint-based Event Trace Reduction

Jie Wang
University of Chinese Academy of Sciences
Institute of Software, Chinese Academy of Sciences, Beijing, China
wangjie12@otcaix.iscas.ac.cn

ABSTRACT

Due to JavaScript's dynamic features, it is challenging to debug JavaScript-based web applications. Thus record-replay techniques are developed to facilitate web application debugging. However, it is time-consuming to inspect all recorded events that reproduce an error. To reduce the cost of debugging, dynamic slicing is used to remove error-irrelevant events by tracking program dependence. However, it cannot remove irrelevant events that the error has program dependence on. In this paper, we propose an effective and efficient approach to remove error-irrelevant events in the event trace. Our approach builds constraints among events and the error (e.g., a variable can read any of its earlier values), to search for a minimal event trace that satisfies these constraints. Our evaluation on 6 real-world web application errors shows that our approach can remove 98% of irrelevant events, and 70% of resulted events by dynamic slicing can be further removed.

CCS Concepts

•Software and its engineering→Software testing and debugging

Keywords

JavaScript; record-replay; event trace reduction

1. INTRODUCTION AND MOTIVATION

To help diagnose JavaScript-based web application errors, various record-replay techniques [1][2] are developed. However, web applications are becoming more complicated and may generate a long event trace after running for a while. It is time-consuming and exhausting to debug with such a long trace. According to a recent study [3], a short event trace for an error can significantly increase programmers' efficiency in error diagnosis, fault localization and fault correction. Thus, event trace reduction techniques [3][4] are proposed to automatically reduce error-irrelevant events.

Two common techniques (delta debugging [3][5] and dynamic slicing [4]) are used to reduce event traces in web applications. The approach in [3] adapts delta debugging to reduce event traces of web application errors. It deletes some events that do not influence the occurrence of an error in each iteration, until no further events can be deleted. However, delta debugging relies on trial-and-error (black box), and does not scale to huge event traces due to (1) the large search space and (2) re-executing every blindly generated event trace. In our experiments, it costs 3 minutes for an event trace with only 341 events.

Our previous work JSTrace [4] adopts dynamic slicing (white box) to trace the precise program dependence and discards the events that are not depended by an error. However, not all remaining events in JSTrace are necessary to reproduce the error. Let's see an example in Figure 1. This code snippet shows the event handler when an item is added to a shopping list. Considering the following event trace: (1) e_1: add an item named "book1"; (2) e_2: add an item named "book2"; (3) e_3: add an item named "book1". An error will occur if two added items have the same name (e.g., "book1"). Only

```
1.    function onAddItem(){
2.       var item = new Item(getElement('item_name').value);
3.       shoppingList = shoppingList || [];
4.       shoppingList.push(item); // Throw an except when item exists.
5.    }
```

Figure 1. Event handler for adding items to shopping list.

e_1 and e_3 are enough to trigger this error. However, based on dynamic slicing, e_3 depends on e_2 (e_3 uses variable *shoppingList* written by e_2 at line 3) and e_2 depends on e_1 (e_2 uses variable *shoppingList* written by e_1 at line 3). As a result, we cannot delete e_2 although it is unnecessary to reproduce this error.

In this paper, we propose a novel constraint-based approach to *effectively* and *efficiently* remove error-irrelevant events in the event trace that leads to an error. First, we relax the program dependence constraint (e.g., a variable can read any of its earlier value), thus irrelevant events (e.g., e_2) can be removed (effectiveness). Second, we use constraints (e.g., variables should be defined before used) to filter out event traces that cannot reproduce the error (efficiency).

2. BACKGROUND AND RELATED WORK

We focus on those work that concern record-replay in web applications and techniques for trace/test reduction.

Record-replay in web applications. Mugshot [1] captures all events and all non-deterministic information such as random API calls and timers to make sure the replay phase behaves the same. Timelapse [2] further supports interactive record-replay.

Trace/test reduction. Dynamic slicing [4] can be used to reduce event traces for web applications, and also for other programs [6][7][8]. However, it faces the problem of unnecessary dependency. Andreas [5] proposed delta-debugging to simplify a failing test case to a minimal one. The work [3] adopts delta debugging to simplify web application event traces. SimpleTest [9] reconstructs a test to a simpler one by repeatedly replacing referred expressions in each statement with other alternatives. While [10] applies partial-order and def-use relationship between events to identify redundant event traces. However, SimpleTest and While cannot be used for event trace reduction.

3. APPROACH AND UNIQUENESS

Overview. Figure 2 shows our approach overview. Our approach consists of three phases: (1) *information collecting*. We instrument the source code to collect runtime information while replaying the original event trace; (2) *trace generating*. We construct constraints according to the collected information, and generate possible event traces that can reproduce the error (candidate traces); (3) *trace validating*. Each candidate trace is validated to check if it can reproduce the error, thus the invalidated ones will be pruned.

The constraints for candidate event traces come from two main observations: (1) The selected events should at least be *feasible* (i.e., a variable must be defined before used). (2) The exact value of a variable v may not be critical to the error. Thus, a variable v could be relaxed to any of its earlier values that are type-compatible with v. For example in Figure 1, we require that *shoppingList* in e_3 reads the value written in e_1. Thus, we can possibly remove e_2.

Uniqueness. The advantage of our approach is that (1) by relaxing the dependency of a variable, rather than the exact dependency used in dynamic slicing, the unnecessary dependency (and related events) could be removed; (2) by making constraints on the target event trace, rather than blindly trial, the search space is greatly narrowed down compared to delta-debugging [3].

3.1 Information Collecting

Because *shared variables* that are accessed by multiple events can affect the execution of an event trace, we only trace shared variables in the event trace. We instrument the source code to collect necessary runtime information.

We collect type and def-use information for each shared variable v so that we can build a minimal syntax constraint (Section 3.2.1) and the value of v can be relaxed to its any value that is type-compatible with v (Section 3.2.2). For each shared variable v: 1) The type of v can be "undefined", "number", "string", "boolean", and "object". For a DOM element, we mark it as "DOM" type instead of "object". 2) The events that operate on v and define v are recorded.

We symbolize each shared variable v and trace their symbolic expressions so that we can use them to check if a specific program state is satisfied (Section 3.3). Symbolic expressions are collected similar to any dynamic symbolic execution [11][12][13].

3.2 Event Trace Generating

Given an original event trace $\tau = \{e_1, e_2, \dots, e_n\}$ that can reproduce an error, our approach could generate a subset τ' of τ that can still reproduce the error. The trace τ' should be as short as possible. Let *select*(e_i) denote whether event e_i is selected by τ'. If e_i is selected by τ', e_i = 1. Formally, *select*(e_i) is:

$$select(e_i) \equiv (e_i{==}1).$$

The trace generating formula Φ is constructed by a conjunction of three sub-formulas: $\Phi \equiv \Phi_d \wedge \Phi_c \wedge \Phi_m \wedge \Phi_e$, where Φ_d denotes the minimal syntax constraint, Φ_c denotes the type-compatible constraint, Φ_m denotes the length constraints, and Φ_e denotes that the error-triggering event must be selected.

3.2.1 Minimal Syntax Constraint (Φ_d)

The minimal syntax constraint (Φ_d) ensures that a variable is used after necessary definition. Specifically, Φ_d requires that: (1) A local variable should be explicitly defined. A global variable could be used without definition, but a local variable must be explicitly defined using keyword *var*. Thus, if event e is selected, then all events that define the variables used by e should be selected. (2) An event handler should be called after its registration. Otherwise, an event will fail to trigger the event handler. Thus, if event e is selected, then the events that register the event handler of e should be selected. We can regard an event handler as a variable v, and the registration of v as its definition. Let *use*(e) be the set of variables used by e, *def*(v) be the event that defines the variable v. Formally, Φ_d is:

$$\Phi_d \equiv \wedge_{e\in\tau}(select(e) \Rightarrow \wedge_{v\in \mathrm{use}(e)}\, select(def(v))).$$

3.2.2 Type-Compatible Constraint (Φ_c)

Type-compatible constraint (Φ_c) is used to ensure that each variable reads the same type as recorded, although their exact value may be different. By relaxing the dependency of a variable, we can generate more simplified trace. Φ_c requires that if an event e is selected, then for all variables used by e, at least one of its type-compatible modification events is selected. Let *CEvent*(v) be the set of events that contain type-compatible modifications to v. Formally, Φ_c is:

$$\Phi_c \equiv \wedge_{e\in\tau} select(e) \Rightarrow \wedge_{v\in use(e)} (\vee_{ej\in CEvent(v)}\, select(e_j)).$$

We can directly compare the type information collected by symbolic recorder to decide if a previous modification is type-compatible. However, if v is marked as "DOM" type, we need to subtly models its *CEvent*(v) because DOM is build-in object in browser and has complicated (native) APIs. For DOM type, we say v_1 is type-compatible with v_2 when the DOM tree of v_1 has the same structure as the DOM tree of v_2.

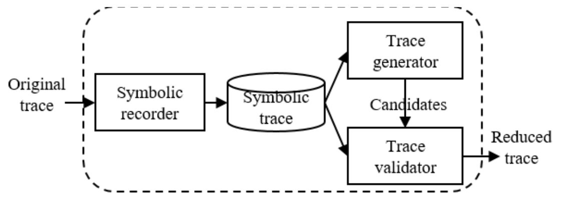

Figure 2. Approach overview.

3.2.3 Length Constraint (Φ_m)

Length constraint (Φ_m) is used to restrict the length of candidate event traces. Let *length* be the required maximal length of candidate traces. We could generate candidate traces by increasing *length* from 1. Formally, Φ_m is:

$$\Phi_m \equiv (\textstyle\sum_{e\in\tau} e_i) == length.$$

3.3 Event Trace Validating

The generated candidate traces are checked whether they can reproduce the error, and thus prune the false ones. Instead of simply replaying the candidate traces as delta debugging, we utilize the collected symbolic expressions to check the validation.

Our observation is that the execution of a valid event trace may follow the same path conditions and hit the same error as the original event trace does. That is, for each valid trace, the following constraints should be satisfied: (1) Path constraint (Φ_p). All the path conditions hold the same value as recorded. (2) Error constraint (Φ_a). Error assertions tell if the error will occur. We calculate the value of each symbolic expression for a given candidate trace and further to check if the symbolic expression for $\Phi_p \wedge \Phi_a$ is satisfied.

4. RESULTS AND CONTRIBUTIONS

Our evaluation answers the following two questions: (1) whether our approach can *effectively* remove error-irrelevant events; (2) how is our approach compared to existing approaches (dynamic-slicing [4] and delta-debugging [3]). Our approach is evaluated on 6 real-world errors in terms of reduction rate, search space and time overhead. The result shows that our approach can effectively remove 98% of error-irrelevant events, and can further remove 70% of resulted events by dynamic slicing [4]. The average reduction time overhead is less than 1 minute. Our approach narrows down the search space to 3.2% compared to delta debugging [3], while keeping high reduction rate. The contributions of this paper are as follows:

- We propose a novel approach that transforms event trace reduction problem into a constraint solving problem.
- The evaluation on 6 real-world web application errors show our approach can effectively and efficiently remove error-irrelevant events.

5. REFERENCES

[1] J. Mickens, J. Elson, and J. Howell, “Mugshot : Deterministic Capture and Replay for JavaScript Applications,” in *Proceedings of the 7th 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] 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 the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software (ESEC/FSE)*, 2015, pp. 333–344.

[4] J. Wang, W. Dou, C. Gao, and J. Wei, “Fast reproducing web application errors,” in *Preceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015, pp. 530–540.

[5] 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.

[6] J. Krinke, “Context-Sensitive Slicing of Concurrent Programs,” *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE)*, pp. 178–187, 2003.

[7] D. Giffhorn and C. Hammer, “Precise Slicing of Concurrent Programs: An Evaluation of Static Slicing Algorithms for Concurrent Programs,” *Automated Software Engineering*, vol. 16, no. 2, pp. 197–234, 2009.

[8] X. Zhang, S. Tallam, and R. Gupta, “Dynamic Slicing Long Running Programs Through Execution Fast Forwarding,” *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 81–91, 2006.

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

[10] 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.

[11] K. Sen, “Concolic Testing,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE)*, 2007, pp. 571–572.

[12] G. Li, E. Andreasen, and I. Ghosh, “SymJS: Automatic Symbolic Testing of JavaScript Web Applications,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering(ICSE)*, 2014, pp. 449–459.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution,” in *IEEE Symposium on Security & Privacy (S&P)*, 2010, pp. 317–331.