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
Various record-replay techniques are developed to facilitate web application debugging. However, it is time-consuming to inspect all recorded events that reveal a failure. To reduce the cost of debugging, delta-debugging and program slicing are used to remove failure-irrelevant events. However, delta-debugging does not scale well for long traces, and program slicing fails to remove irrelevant events that the failure has program dependence on. In this paper, we propose an effective and efficient approach to remove failure-irrelevant events from the event trace. Our approach builds constraints among events and the failure (e.g., a variable can read any of its earlier type-compatible values), to search for a minimal event trace that satisfies these constraints. Our evaluation on 10 real-world web applications shows that our approach can further remove 70% of events in the reduced trace of dynamic slicing, and needs 80% less iterations and 86% less time than delta-debugging.

CCS Concepts
•Software and its engineering→Software testing and debugging

Keywords
JavaScript; failure; event trace reduction

1. INTRODUCTION AND MOTIVATION
To help diagnose JavaScript-based web application failures, 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 to debug with such a long trace. According to a recent study [3], a short event trace for a failure can significantly increase programmers' debug efficiency. Thus, event trace reduction techniques are proposed to reduce failure-irrelevant events, e.g., delta debugging [3][4] and dynamic slicing [5].

The delta debugging technique, e.g., the [3], removes some events that do not influence the occurrence of a failure in each iteration, until no further events can be deleted. However, delta debugging is 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 27 minutes for an event trace with only 617 events.

Our previous work JSTrace [5] adopts dynamic slicing to trace the precise program dependence and discards the events that are not depended by a failure. However, not all remaining events in JSTrace are necessary to reproduce the failure. Let's see the 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". A failure will occur if two added items have the same name (e.g., "book1"). Only e_1 and e_3 are enough to trigger this failure. 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 remove e_2 although it is unnecessary to reproduce this failure.

```
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.

In this paper, we propose a novel constraint-based approach to *effectively* and *efficiently* remove failure-irrelevant events in the event trace that leads to a failure. First, we relax the program dependence (e.g., force the variable *shoppingList* in e_3 to directly read from its value in e_1), 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 failure (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. Web application record-replay tools have been developed for faithfully reproducing failures, e.g., Mugshot [1] and Timelapse [2]. However, they do not figure out which events are relevant to failures.

Delta-debugging. Delta debugging is widely used to facilitate program debugging [4][6][7][8][9][10]. Hammoudi et al. [3] adapts delta-debugging [4] to reduce web application event traces. Their approach operates by repeatedly selecting subsets of the events in a trace, and replaying these subsets to determine whether they can reveal the failure. Their work relies on blindly generating event subsets. While our approach utilizes the runtime information to restrain the target event trace generation, and by this way greatly narrows down the search space.

Program Analysis. JSTrace [5] adopted dynamic slicing [11][12] to simplify event traces. However, a computation may still be redundant even if it is depended by a failure. SimpleTest [13] reconstructs a test to a simpler one by repeatedly replacing referred expressions in each statement with other alternatives. While the work [14] applies partial-order and def-use relationship between events to identify redundant event traces.

3. APPROACH AND UNIQUENESS

Overview. Our approach consists of three phases. (1) We instrument the source code to collect runtime information. (2) We construct constraints according to the collected information, and generate candidate event traces that are likely to reproduce the failure. (3) Each candidate trace is checked if it can reproduce the failure. Phases 2 and 3 are repeated until a valid trace is generated.

We have two observations to generate candidate event traces. (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 failure. Thus, variable v could be relaxed to any of its earlier type-compatible values. For example in Figure 1, we require that *shoppingList* in e_3 directly reads the value written by e_1. Thus, we can possibly remove e_2.

Uniqueness. The advantage of our approach is that (1) more failure-irrelevant events can be removed by relaxing program dependence, rather than the exact dependence used in dynamic slicing; (2) the search space is narrowed down since our constraints restrict it by using runtime information.

3.1 Information Collecting

We collect the following runtime information, so that we can use them to guide trace generating and validating.

***Types and def-use*:** We collect type and def-use information so that we can build constraints (Section 3.2) to generate candidate traces. For each variable v: 1) The type of v is recorded. It can be "undefined", "number", "string", "boolean", and "object". Since DOM is special object, we mark it as "DOM" type instead of "object". 2) The events that operate on v and define v are also recorded.

Symbolic expressions: We symbolize each 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 [15][16][17]. Since we only calculate their values rather than solving the path constraints to explore new execution paths, our validating does not suffer from poor performance problem.

3.2 Event Trace Generating

Given an event trace $\tau = \{e_1, e_2, \ldots, e_n\}$ that can reproduce a failure, our approach generates a subset τ' of τ, which can still reproduce the failure. We require 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 four sub-formulas: $\Phi \equiv \Phi_s \wedge \Phi_c \wedge \Phi_l \wedge \Phi_e$, where Φ_s denotes the minimal syntax constraint, Φ_c denotes the type-compatible constraint, Φ_l denotes the length constraint, and Φ_e denotes that the failure-triggering event must be selected.

3.2.1 Minimal Syntax Constraint (Φ_s)

The minimal syntax constraint (Φ_s) ensures that a variable is used after necessary definition. Specifically, Φ_s 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 fails 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 events that define the variable v. Formally, Φ_s is:

$$\Phi_s \equiv \bigwedge_{e\in\tau}(select(e) \Rightarrow \bigwedge_{v\in use(e)} select(def(v))).$$

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 dependence 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 written events is selected. Let $CEvent(v)$ be the set of events that contains type-compatible written to v. Formally, Φ_c is:

$$\Phi_c \equiv \bigwedge_{e\in\tau} select(e) \Rightarrow \bigwedge_{v\in use(e)} (\bigvee_{ej\in CEvent(v)} select(ej)).$$

We can directly compare the type information collected to decide if a previous written value is type-compatible. However, if v is marked as "DOM" type, we need to subtly model its $CEvent(v)$ because it has complicated tree-like structure. 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 that of v_2.

3.2.3 Length Constraint (Φ_l)

Length constraint (Φ_l) restricts the length of candidate traces. Let *length* be the maximal length of candidate traces. Formally, Φ_l is:

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

The initial value of *length* is 1 and we increate it by 1 if there are no solutions for *length*. This process is repeated until a valid trace is found. This strategy make short traces be generated first. Thus, we could quickly find the valid trace since the failure related event trace is usually relatively short [3][18][19].

3.3 Event Trace Validating

We validate each generated candidate trace if it can reproduce the failure. Instead of replaying the candidate trace as delta debugging, we utilize the symbolic expressions to make the validation.

We observe that a valid event trace may follow the same path conditions and hit the same failure as the original event trace does. For each trace, the following constraints should be satisfied: (1) Path constraint (Φ_p). All the path conditions (i.e., the branches of the execution) hold the same value as recorded. (2) Failure constraint (Φ_f). Failure assertions tell if the failure occurs. We calculate the value of each symbolic expression for a given candidate trace and check if $\Phi_p \wedge \Phi_f$ is satisfied.

4. RESULTS AND CONTRIBUTIONS

We performed our evaluation on 10 real-world web application failures used in JSTrace [5]. The evaluation shows that our approach can further remove 70% of events in the reduced trace of dynamic slicing, and needs 80% less iterations and 86% less time than delta-debugging. Note that the time overhead of our approach includes information collecting (65.3%), trace generating (5.6%) and trace validating (29.1%). 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 10 real-world failures shows our approach can effectively and efficiently remove failure-irrelevant events.

5. ACKNOWLEDGMENTS

This work was supported in part by National Key Research and Development Plan (2016YFB1000803) and National Natural Science Foundation (61672506) of China.

6. 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] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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, To appear.

[10] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?," in *Proceedings of the 7th European Software Engineering Conference(ESEC)*, 1999, pp. 253–267.

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

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated Testing with Targeted Event Sequence Generation," in *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, 2013, pp. 67–77.