Feedback-Directed Differential Testing of Interactive Debuggers

Daniel Lehmann
Department of Computer Science
TU Darmstadt, Germany
mail@dlehmann.eu

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany
michael@binaervarianz.de

ABSTRACT

To understand, localize, and fix programming errors, developers often rely on interactive debuggers. However, as debuggers are software, they may themselves have bugs, which can make debugging unnecessarily hard or even cause developers to reason about bugs that do not actually exist in their code. This paper presents the first automated testing technique for interactive debuggers. The problem of testing debuggers is fundamentally different from the well-studied problem of testing compilers because debuggers are interactive and because they lack a specification of expected behavior. Our approach, called DBDB, generates debugger actions to exercise the debugger and records traces that summarize the debugger's behavior. By comparing traces of multiple debuggers with each other, we find diverging behavior that points to bugs and other noteworthy differences. We evaluate DBDB on the JavaScript debuggers of Firefox and Chromium, finding 19 previously unreported bugs, eight of which are already fixed by the developers.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; *Software maintenance tools*;

KEYWORDS

Interactive debuggers, Differential testing, JavaScript

ACM Reference Format:
Daniel Lehmann and Michael Pradel. 2018. Feedback-Directed Differential Testing of Interactive Debuggers. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236037

1 INTRODUCTION

Interactive debuggers are a powerful tool to find and correct bugs in programs. Unlike much simpler methods, such as `printf`-debugging, interactive debuggers allow the developer to directly follow the program at runtime. In particular, one can pause the execution at points of interest through *breakpoints*, closely examine control-flow through *stepping*, and inspect *intermediate program state*, such as the call stack and the values of variables.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236037

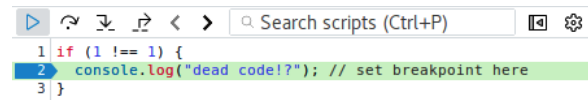

Figure 1: JavaScript debugger of Firefox 54 incorrectly pauses at a breakpoint in dead code. Bug found with DBDB.

As debuggers are a crucial tool in the development workflow, they obviously should be correct. Figure 1 demonstrates that this is not always the case, even when debugging a seemingly simple program: Since the if-condition evaluates to `false`, the statement at line 2 is never executed and nothing gets written to the console. Yet, when a breakpoint is set at line 2 in the Firefox debugger, it pauses, which gives the impression that dead code is executed.[1]

Such bugs in debuggers are very confusing because they can lead developers to believe their code is wrong even though it is correct. Even worse, bugs in debuggers can make it hard or even impossible to understand actual bugs, e.g., when a developer cannot set a breakpoint at a valid code location,[2] when variables are shown with wrong values,[3] or when the debugger obscures the actual control-flow by not pausing where it should.[4] Even when a debugger is not as blatantly wrong as shown in Figure 1, there are often subtle differences between two debugger implementations for the same programming language. Such differences are equally worrying, not just because developers are confused when behavior changes just by switching tools, but also because it shows that some of the intended behavior of debuggers is not well-specified.

Related Ideas. Finding bugs and other unexpected behavior in debuggers is a surprisingly understudied problem. The closest existing line of work addresses the correctness of compilers and interpreters. One approach is software verification, which has been successfully applied, e.g., in CompCert [21]. When the programming language is well-specified, many compilers and interpreters can also build on extensive conformance suites [12] or large test suites that are manually written by the developers [33, 38]. Unfortunately, manual tests are laborious to write and thus often insufficient. The lack of manual testing has lead to research on *automatic testing* of developer tools, where the tool under test is executed with generated programs. Such techniques have found hundreds of bugs in compilers [20, 39], interpreters [14], and other programming tools, such as refactoring engines [9].

Challenges. Unfortunately, debuggers differ from compilers and interpreters in key aspects that make it challenging to directly apply the above ideas. First, unlike compilers, debuggers take not just a program as input but also allow the user to steer the debugging

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=1370648.
[2] https://bugs.chromium.org/p/chromium/issues/detail?id=784852.
[3] https://bugzilla.mozilla.org/show_bug.cgi?id=1362432.
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=1362403.

session, often through a graphical user interface (GUI) or command-line interface. The *debugging actions* a user issues there, such as setting breakpoints and stepping, determine the debugger's behavior. A challenge for effective debugger testing is to generate such debugging actions alongside a given program-to-debug.

Second, interactive debuggers, as the name implies, differ from compilers because debuggers interact with the user, instead of taking all inputs at once in the beginning and producing outputs only at the end of the execution. After each debugger input (e.g., a step action), the debugger pauses and only then shows the next outputs (e.g., the line that is executed), waiting for the next input from the user. The problem is compounded by the fact that previous outputs determine the next possible inputs. For example, stepping is no longer possible when the execution is finished. For these reasons, automatic testing of debuggers cannot simply generate all inputs "offline" before the debugging session, unlike in compiler testing.

Third, debuggers often *lack a precise specification* of the intended behavior. Whereas many compilers and interpreters follow a language standard [1, 2, 16], we are only aware of few attempts to specify the behavior of debuggers, none of which target real-world debuggers [3, 35]. Without a specification, conformance testing and formal methods cannot be applied to debuggers.

Approach. We present **DBDB**, an automatic, differential testing technique to detect **d**iverging **b**ehaviors of **deb**uggers. The basic idea is to automatically interact with two supposedly equivalent debuggers to find a divergence of their behavior. We base our work on a simple model of interactive debuggers as finite-state transducers. In this model, debuggers take debugging actions as inputs, such as setting a breakpoint in a specific line, resuming, or stepping, and return outputs that capture the behavior of the debugger, e.g., which line is currently executing or the values of local variables. Given two debugger implementations for the same programming language, DBDB generates debugging actions, executes them in parallel in both debuggers, and compares their behavior. Since debuggers are interactive, the approach iterates between generating actions and comparing outputs until finding diverging behavior. The lack of a commonly accepted specification of debuggers results in a large number of these diverging behaviors, too many for direct manual inspection. To reduce the inspection effort, we assign each diverging behavior an equivalence class and then only inspect these equivalence classes.

Results. We evaluate DBDB on the widely used JavaScript debuggers of Firefox and Chromium. We exercised them with a total of 26,931 generated actions in 2,050 debugging sessions. After less then 20 executed actions, already 82.5% (1,692) of the debugging sessions show diverging behavior between Firefox and Chromium.

Among these diverging behaviors are at least 20 bugs, 19 of which were previously unknown and we thus reported to the respective developers. Of the five reported Chromium bugs, four are already fixed by the developers. In the Firefox debugger we reported 14 bugs, four of which are now marked as fixed. Apart from bugs, we have also found many subtle differences between the Firefox and Chromium debuggers. Understanding these undocumented differences can pave the way to a precise specification of the intended behavior of debuggers and could help different vendors to agree on a common debugger interface.

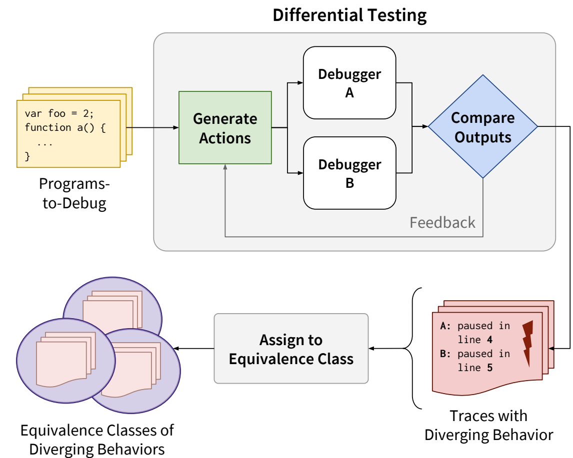

Figure 2: DBDB overview: differential testing finds diverging behaviors, which are then assigned to equivalence classes.

Contributions. In summary, this paper contributes the following:

- We identify common functionalities of interactive debuggers and model them as a finite-state transducer that translates debugging actions to outputs (Section 3.1).
- We present the first differential testing technique for debuggers. The technique uses feedback-directed test input generation to account for the interactive nature of debuggers and assigns diverging behavior to equivalence classes to reduce the manual inspection effort. (Section 3.2 to 3.4).
- We provide empirical evidence that the approach effectively reveals bugs and other diverging behavior in widely used debuggers. (Section 5).
- We make our implementation and results available to the public under https://github.com/sola-da/DifferentialDebuggerTesting.

2 OVERVIEW AND MOTIVATING EXAMPLE

This section gives an overview of our approach and explains the main steps with a simple example. Figure 2 shows the main parts of our approach. The input to DBDB is a program-to-debug and two debuggers that are supposed to provide the same debugging functionalities. For our running example, suppose these debuggers are the JavaScript debuggers of Firefox and Chromium. The approach interacts with both debuggers side-by-side by generating *debugging actions*, such as setting breakpoints and stepping through the program. During this interaction, the approach tracks the behavior of both debuggers and searches for any inconsistencies. When DBDB detects any diverging behavior, it stops the interaction and reports the difference as a pair of execution traces.

As a running example, consider the JavaScript code in Figure 3. The code iterates through an array and then executes two `if`-statements that each define a local variable. The `let` keyword indicates that these variables are local to the surrounding block scope, i.e., they are visible only within their respective then-branches. Suppose that DBDB starts debugging the code in Figure 3 by setting a breakpoint at line 1. Because both debuggers indicate that setting the breakpoint was successful, the approach generates another action by setting a second breakpoint at line 3. The Chromium

```
1  var array = [1, 2, 3];
2  for (prop in array) {
3    console.log("hi");  // Bug: Cannot set breakpoint here
4  }
5  if (array.length > 0) {
6    let firstVar = "first";
7  }
8  if (array.length > 1) {
9    let secondVar = 23; // Bug: Appears to be "first"
10 }
```

Figure 3: Running example.

debugger confirms the breakpoint at line 3. In contrast, the Firefox debugger slides the breakpoint to line 5. *Breakpoint sliding* is a common debugger feature to prevent users from setting breakpoints at, e.g., empty lines and comment-only lines. But the difference observed here is clearly unintended, as it makes it impossible to debug the statement at line 3. Our approach detects this difference, which has been fixed in Firefox 55 after we reported it to the developers.[5]

After finding a difference between the two debuggers, DBDB continues to explore more behavior. Suppose that DBDB starts debugging the code by setting a breakpoint at line 8. Then, the approach starts the program's execution and the code hits the breakpoint in both debuggers. As the next action, suppose that DBDB issues a *step in* action and the debugger subsequently pauses execution at line 9. Whenever the debugger is paused, our approach compares the program state reported by both debuggers. In this example, the comparison shows that Firefox claims `secondVar` to have the value `"first"`, whereas the Chromium debugger claims `secondVar` to be `undefined`. This surprising inconsistency is due to a bug in Firefox, which accidentally shows the value of `firstVar`, even though that variable is not in scope at line 9. We have reported this problem and the Firefox developers confirmed that it is a bug.[6]

The example illustrates that real-world interactive debuggers, as any other software, are not free of bugs. Since developers heavily rely on these tools to debug their own code, finding such problems is important. The following section presents our approach for finding unexpected and underspecified behavior of debuggers via differential testing.

3 APPROACH

This section formally defines the problem addressed by DBDB and describes our approach in detail. The basis of our approach is a finite-state model of interactive debuggers (Section 3.1). Building on top of this model, Section 3.2 defines the problem of finding diverging behaviors for a given pair of debuggers. Section 3.3 presents our algorithm to address this problem through automatic, differential testing. Finally, Section 3.4 describes a technique to assign diverging behaviors to equivalence classes, which eases the task of manually inspecting and understanding differences between debuggers.

3.1 Finite-State Model of Interactive Debuggers

Implementations of real-world interactive debuggers are complex and analyzing them is non-trivial. To keep our approach generic and independent of a specific debugger or programming language, we abstract interactive debuggers into a formal model. The model

[5] https://bugzilla.mozilla.org/show_bug.cgi?id=1362416.
[6] https://bugzilla.mozilla.org/show_bug.cgi?id=1363328.

$$
\begin{array}{rcll}
\Sigma & ::= & \mathit{BpAction} \mid \mathit{ExecAction} \mid \varepsilon & \text{(Actions)} \\
\mathit{BpAction} & ::= & \text{Set breakpoint at } \mathit{line} \mid & \\
 & & \text{Remove breakpoint at } \mathit{line} & \\
\mathit{ExecAction} & ::= & \text{Start execution} \mid \text{Resume} \mid & \\
 & & \text{Step in} \mid \text{Step out} \mid \text{Step over} & \\
\Gamma & ::= & \mathit{BpOutput} \mid & \text{(Outputs)} \\
 & & \mathit{ProgramState} \mid & \\
 & & \text{Execution finished} \mid & \\
 & & \varepsilon & \\
\mathit{BpOutput} & ::= & \text{Breakpoint set at } \mathit{line} \mid \text{Removed} \mid \text{Not removed} & \\
\mathit{ProgramState} & ::= & \langle \mathit{line}, \mathit{callStack}, \mathit{vars} \rangle & \\
\mathit{callStack} & ::= & \mathit{name}* & \\
\mathit{vars} & ::= & (\mathit{name} : \mathit{type} = \mathit{value}?)* & \\
\mathit{line} & \in & \mathbb{N}, \text{ line numbers} & \text{(Meta Variables)} \\
\mathit{name} & \in & \text{identifiers} & \\
\mathit{type} & \in & \text{types} & \\
\mathit{value} & \in & \text{primitive values} &
\end{array}
$$

Figure 4: Grammars of debugging actions and outputs.

focuses on features common to most real-world interactive debuggers and abstracts away properties of debuggers that are irrelevant for analyzing them.

Our model is based on finite-state transducers (FSTs). FSTs are a variant of finite-state machines where each transition can both consume input and produce output. They are widely used in natural language processing, e.g., for machine translation or part-of-speech tagging [27]. An FST model fits the interactive nature of debuggers, where inputs are debugging actions triggered by a user and outputs represent the resulting behavior of the debugger. More formally, we represent a debugger as follows:

Definition 1 (Debugger). A debugger for a program P is a 5-tuple $(Q, q_0, \Sigma, \Gamma, \delta_P)$ where

- Q is a finite set of states,
- $q_0 \in Q$ is the initial state,
- Σ is a set of input symbols that represent debugging actions the user can trigger,
- Γ is a set of output symbols that represent debugger behavior,
- $\delta_P : (Q \times \Sigma) \times (\Gamma \times Q)$ is a transition relation that maps the current state and some debugging action to the behavior produced by the debugger and the next debugger state.

Figure 4 summarizes the inputs and outputs considered in this work by showing the grammars for the input and output languages Σ and Γ. Debugging actions in Σ are either related to breakpoints (*BpAction*) or control the execution of the program (*ExecAction*). Debugging outputs in Γ are either breakpoint-related (*BpOutput*), provide details about the current program state (*ProgramState*), or indicate that the program has terminated. We will explain the symbols and how they relate to real-world debuggers in the following.

Figure 5 shows the states Q and the transition relation δ_P of a debugger. We model debuggers as having three states: *not running*, *running*, and *paused*. The debugger transitions between two of these states q and r when taking an action x as input and in turn exhibits some behavior represented by a debugging output y. In the figure

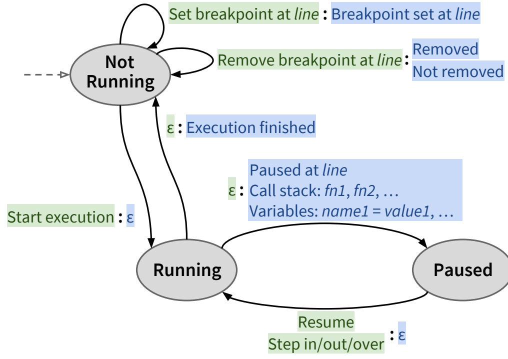

Figure 5: Interactive debuggers as finite state transducers. In transitions, ":" separates inputs (green) from outputs (blue).

and in the following definitions, we use the notation $q \xrightarrow{x:y} r$ for such a transition $(q, x, y, r) \in \delta_P$. A transition where the input is the empty word ε indicates some behavior that is not directly triggered by the user. Likewise, we assign an output ε to debugging actions that do not have an immediate output.

Besides debugging actions, the behavior of a debugger depends on the program-to-debug P. We model this dependency by indexing the transition relation δ_P with P. Since P can keep internal state, the next debugger state and output depends not only on the current debugger state and action, but also on P's state. Apart from δ_P, all other parts of our model are independent of the program-to-debug.

Initially, the program-to-debug P is not executing and the debugger is in the state q_0 = *not running*. In this state, users can set and remove breakpoints, either by clicking on line numbers in a GUI or in a prompt of, e.g., GDB or LLDB. Set breakpoint actions and remove breakpoint actions have an immediate output: the line where the breakpoint was set and whether removing a breakpoint was successful, respectively.

The start execution action transitions the debugger to the *running* state and runs the program P. In the running state, debuggers do not display any information about the execution to the user. Only when a breakpoint is hit or a step completes, the debugger transitions to the *paused* state and during that transition outputs information about the current program state. In our model, this output of program state comprises three pieces of information $\langle$*line*, *callStack*, *vars*$\rangle$:

- in which *line* the program has paused,
- the *callStack* as a sequence of function names, and
- the set *vars* of local variables, along with their types and, for primitive types, their values.

Issuing a resume action, e.g., by clicking on ▷ in a graphical debugger or by typing `continue` in a prompt, transitions back to the running state until the next breakpoint is hit. Similarly, stepping through the program continues execution until the next step of computation completes. We model three kinds of steps:

- Step in (⤓), which executes the next statement and, if it is a function call, stops at the beginning of the callee.
- Step out (⤴), which executes all statements until the end of the current function.
- Step over (↷), which executes the next statement and, if it is a function call, does not enter the function for debugging.

The execution of the program-to-debug can also just finish with a transition back to the not running state. This case happens when the program terminates without hitting any further breakpoint and without pausing after a step.

It is important to point out that it differs significantly from programming tools considered in related work, in particular, compilers. When testing compilers [20, 39] and other development tools [9], the input consists of only a program. In contrast, debuggers expect user actions as an additional input. Another difference is that existing compiler testing is non-interactive, i.e., the produced program produces only a single output, such as its exit code. In contrast, debuggers alternate between taking input and producing output.

3.2 Problem Statement

Based on the finite-state model of debuggers in Definition 1, we now define the problem addressed in this work. The overall goal is to find and understand diverging behaviors between two supposedly equivalent debuggers for the same programming language. The following formally defines diverging behavior.

Definition 2 (Diverging behavior). Given a program P, two debuggers $(Q, q_0, \Sigma, \Gamma, \delta_P)$ and $(\widetilde{Q}, \widetilde{q_0}, \widetilde{\Sigma}, \widetilde{\Gamma}, \widetilde{\delta_P})$ have diverging behavior if two sequences of transitions exist:

$$q_0 \xrightarrow{x:y} .. \xrightarrow{x':y'} r \quad \text{and} \quad \widetilde{q_0} \xrightarrow{\widetilde{x}:\widetilde{y}} .. \xrightarrow{\widetilde{x'}:\widetilde{y'}} \widetilde{r}$$

where $q_0 = \widetilde{q_0}$, $x = \widetilde{x}$, $y = \widetilde{y}$, $x' = \widetilde{x'}$, but where either $y' \neq \widetilde{y'}$ or $r \neq \widetilde{r}$.

That is, given the same sequence of actions dispatched to both debuggers, diverging behavior means that either the outputs y' and $\widetilde{y'}$ differ or that the states r and $\widetilde{r}$ reached by the debuggers differ.

The goal of our approach is to find sequences of inputs that lead to diverging behavior. We assume that the programs-to-debug are provided. The problem of finding suitable programs is related to generating inputs for compiler testing, and our approach may be combined with existing work on generating programs. We further assume that the debugger and the programs-to-debug are deterministic, which is a common assumption to make testing reproducible.

3.3 Interactive Differential Testing

We address the problem of finding diverging behavior between two debuggers through interactive differential testing. The basic idea is to compare two supposedly equivalent debuggers by continuously generating debugging actions and by checking the resulting behavior for inconsistencies.

Challenges. To motivate our interactive approach, we first outline several challenges inherent to testing of debuggers. Many typical uses of debuggers involve a combination of actions, e.g., first setting a breakpoint and then stepping once it is hit. Unfortunately, generating a sequence of such actions ahead of time based on the grammar for Σ is ineffective for three reasons. First, once the program-to-debug terminates, the debugger cannot take more actions and any remaining actions have been generated in vain. Second, some actions that are legal according to the grammar of Σ are illegal for reasons only known at runtime. Consider setting a breakpoint at line 7 of Figure 3. Both debuggers slide the breakpoint to the next line because there is no code to execute at the closing brace. Now, setting a second breakpoint in line 8 is not allowed

and our approach must not generate such an action.[7] Third, once an action causes diverging behavior between debuggers, the analysis has reached an inconsistent state that will only cause more diverging behavior. Consider again the example in Figure 3. When requesting a breakpoint at line 3, Firefox slides it to some later line due to a bug, whereas Chromium correctly sets it at line 3. Executing more actions, e.g., resumes and steps, after this first divergence is uninformative since they will likely result in more diverging behavior because the breakpoints were different. Such follow-up divergences are not relevant on their own but merely the result of having reached an inconsistent state earlier on.

Our algorithm addresses the challenge of testing interactive debuggers by issuing actions to two debuggers side-by-side and comparing their behavior after each action. The approach is feedback-driven in the sense that the behavior triggered by previous actions influences what actions to trigger next. Once the approach observes a difference in behavior between the two debuggers, it stops and reports two traces that summarize the executions.

Definition 3 (Trace). A trace t is the result of interacting with a debugger D on a program P. The trace is a sequence $\langle e_1, .., e_n \rangle$ of n events, where each event is either an action or an output, i.e., $e_i \in \Sigma \cup \Gamma \;\; \forall\, 1 \leq i \leq n$.

The goal of DBDB is to find traces t_A and t_B, one per debugger, that share a common prefix but then diverge in the resulting behavior after the final action. For an example of two such traces, consider Figure 6a. The first event in both traces is a set breakpoint action, followed by the corresponding debugger output. The traces share the same actions and outputs up to the final "Step over". Only then, the debugger behaviors diverge, with Chromium pausing in line 25 of the program, whereas Firefox pauses in line 26.

Algorithm 1 summarizes the main steps of our approach for obtaining such traces by automatically interacting with two debuggers, D_A and D_B, on a program P. The current state of a debugger D is given by $D.state$. We indicate with $D.action()$ that DBDB triggers one of the actions defined by Σ (Figure 4). We assume each triggered action is added to the trace of the corresponding debugger.

The algorithm consists of three parts: manipulating breakpoints before running the program, starting the program execution, and stepping and resuming during program execution. The first part (lines 2 to 13) sets breakpoints at randomly chosen lines and probabilistically removes them again until obtaining a configurable overall number of breakpoints. For each set breakpoint, the algorithm compares the actual breakpoint location chosen by the two debuggers. The intended and the actual breakpoint locations may differ, e.g., because debuggers slide breakpoints instead of adding them to empty lines. If the actual locations differ, the algorithm has detected diverging behavior and therefore stops and returns the trace. Otherwise, the algorithm tries to remove the breakpoints again and checks whether both debuggers agree that removing the breakpoint is possible. While it may appear obvious that removing breakpoints is possible, we found a bug in a debugger that ignored a users request to remove a breakpoint.[8]

[7] Setting two breakpoints in the same line is prevented in GUI-based debuggers because clicking on line numbers toggles a breakpoint. But in lower-level debugging APIs, e.g., of Chromium, an error is thrown when setting two breakpoints at the same line.
[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1362439.

Algorithm 1 Interactive differential analysis

Input: Debugger D_A, D_B and program P
Output: Traces t_A, t_B of actions and outputs

```
1:  Assume: D_A.state = D_B.state = not running
                                        ▷ Manipulate breakpoints:
2:  BPs ← ∅
3:  while |BPs| < max number of breakpoints do
4:    l ← randLine(P)
5:    if l ∈ BPs then
6:      continue
7:    l_A ← D_A.setBp(l); l_B ← D_B.setBp(l)
8:    if l_A ≠ l_B then
        return "Diverging behavior: Bp location"
9:    if randProb() < prob. of removing breakpoint then
10:     ok_A ← D_A.rmBp(l_1); ok_B ← D_B.rmBp(l_2)
11:     if ok_A ≠ ok_B then
          return "Diverging behavior: Bp removal"
12:   else
13:     BPs ← BPs ∪ {l_A, l_B}
                                        ▷ Start program execution:
14: D_A.startExec(); D_B.startExec()
                                        ▷ Step and resume:
15: repeat
16:   Wait until D_A.state ≠ running and D_B.state ≠ running
17:   if D_A.state = D_B.state = not running then
        return "Program finished"
18:   if D_A.state ≠ D_B.state then
        return "Diverging behavior: Termination"
19:   if D_A.programState ≠ D_B.programState then
        return "Diverging behavior: Program state"
20:   action ← randPick({resume, stepIn, stepOut, stepOver})
21:   D_A.action(); D_B.action()
22: until max number of execution actions
```

The second part of the algorithm (line 14) starts the program's execution with the debugger attached to it. The execution will continue until stopping at a breakpoint or until the program terminates.

The third part of the algorithm (lines 15 to 22) repeatedly steps through the program or resumes execution until hitting a breakpoint or program termination. The algorithm waits until both debuggers leave the running state and then checks if their behavior is consistent.[9] If one but not the other debugger has reached the end of the program, then the algorithm reports a diverging behavior. Otherwise, if both debuggers pause the execution, then the algorithm compares the program states $(line, callStack, vars)$ and $(\widetilde{line}, \widetilde{callStack}, \widetilde{vars})$ reported by the debuggers. If the states differ, e.g., because the debuggers have paused at different locations or because they show different call stacks, then the algorithm returns a trace that summarizes the diverging behavior. Finally, if there is no observable difference between the two debuggers, the algorithm triggers a randomly selected execution action (resume or steps) in both debuggers. This process continues until reaching a configurable maximum number of actions.

[9] We assume that the program-to-debug terminates. If a debugger never leaves the running state, then it has a bug that can be detected without any differential analysis. We have not encountered this case in our evaluation.

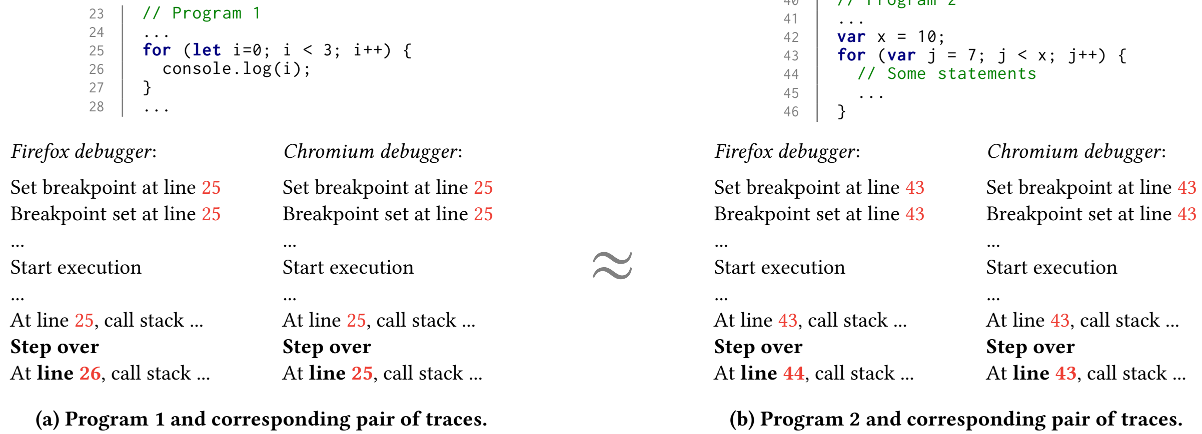

Figure 6: Example of equivalent diverging behaviors exposed by two different programs.

3.4 Equivalence Classes of Diverging Behavior

Running DBDB once with a single program-to-debug may or may not expose diverging behavior. For effective testing, we apply the approach to multiple programs and repeatedly debug each program with different random seeds. The random seed controls all non-deterministic decisions made by Algorithm 1, such as which actions to trigger. As we show in Section 5, repeatedly comparing real-world debuggers produces thousands of traces with diverging behaviors in a few minutes. While this abundance of diverging behavior shows the effectiveness of our differential testing approach, it also leads to the non-trivial challenge of inspecting the diverging behaviors. Manually inspecting all diverging behaviors is practically infeasible.

One way to address this challenge is to uniformly sample all diverging behaviors and to inspect only a subset of them. However, we find that many divergent behaviors are similar and likely have the same root cause. As a motivating example, consider Figure 6, which shows two programs-to-debug that each expose diverging behavior between the JavaScript debuggers of Firefox and Chromium. Below Program 1 and 2, the respective pairs of traces that expose the diverging behavior are shown. Even though the diverging behaviors are caused by different programs, three key characteristics are common between Figure 6a and 6b: (1) the last action is a "Step over", (2) that action is issued at a for-loop, and (3) the diverging behaviors are due to program state, in particular, the lines where the debuggers pause.

We avoid manually inspecting too many of these similar cases by first dividing diverging behaviors into equivalence classes:

Definition 4 (Equivalence classes of diverging behavior). Let P_1 and P_2 be two programs-to-debug, and let D_A and D_B be two debuggers. Suppose that the pairs of traces (t_A^1, t_B^1) and (t_A^2, t_B^2) both expose a diverging behavior, where t_X^i is the result of debugging program P_i in debugger D_X. The two diverging behaviors are in the same equivalence class if

- the last debugging action is the same in t_A^1, t_B^1, t_A^2, and t_B^2,
- the AST node type of the source code line where the last action was triggered is the same for both P_1 and P_2, and
- the type of diverging behavior between t_A^1 and t_B^1 is the same as for the diverging behavior between t_A^2 and t_B^2.

To compute the AST node type of a source code line l, we parse the program-to-debug and then search the lowest AST node that includes all tokens in l. We compare AST node types and not program lines directly, because AST node types abstract away program-specific properties, such as concrete line numbers or identifiers.

The type of diverging behavior is determined by the return value of Algorithm 1. Specifically, there are six types of diverging behaviors (line numbers refer to Algorithm 1):

- Breakpoints are set at different locations (line 8).
- Breakpoints cannot be consistently removed (line 11).
- One but not the other debugger terminates (line 18).
- The program state in both debuggers is different (line 19), subdivided by which part exactly differs (see Figure 4):
  - the debuggers paused at different lines, or
  - the function names on the call stack differ, or
  - the variables and their types and values differ.

Then, we draw our samples for manual inspection from the equivalence classes in a round robin manner instead of uniformly sampling all diverging behaviors. The technique is a heuristic that is independent of the programming language and we show in Section 5 that it results in a more diverse set of inspected diverging behaviors, which ultimately leads to more found bugs.

Reiterating on the example in Figure 6, our technique assigns the two diverging behaviors to the same equivalence class because all three conditions from Definition 4 are met. In this particular case, the two diverging behaviors are even caused by the same root cause: the Chromium debugger always pauses at each substatement when stepping over a for-loop header, whereas in Firefox a "Step over" always goes to the next line.

4 IMPLEMENTATION

We have implemented DBDB in TypeScript and run it, after compilation to JavaScript, in Node.js. Firefox and Chromium offer a programmatic interface to their debuggers through their respective remote debugging protocols (RDP). That is, we do not test the debuggers, for example, by clicking in the GUI, but directly exchange RDP messages with the debuggers via a WebSocket connection.

For Chromium, we build on the *chrome-remote-interface* library,[10] which already offers a simple abstraction, e.g., to attach the debugger to a JavaScript program, set breakpoints, and perform other debugging actions. For Firefox, we implement the RDP ourselves since no up-to-date RDP library has been available. On top of these low-level remote debugging protocols, we implemented a common, higher-level API for both debuggers. To make sure that every found diverging behavior is not just visible at the remote debugging protocol level, we also manually reproduced every bug in the debuggers' GUIs. In particular, all our bug reports include videos of the visibly wrong behavior in the GUI. The implementation is publicly available under https://github.com/sola-da/DifferentialDebuggerTesting.

5 EVALUATION

5.1 Experimental Setup

We apply DBDB to two popular JavaScript debuggers: Firefox 54.0 and Chromium 59.0.3071.109. We run DBDB on a laptop with 8 GB of system memory and an Intel Core i5-5200U CPU. The operating system is Ubuntu 16.10 64-bit.

We use 41 JavaScript programs-to-debug that are obtained from three sources: First, we use 26 programs from SunSpider, a JavaScript benchmark, version 1.0.2, which according to the original announcement cover a "wide variety of numerical, array-oriented, object-oriented, and functional idioms" [30]. Second, we use 11 JavaScript puzzles for students from a program analysis lecture at TU Darmstadt. The puzzles cover corner cases of the language, making these programs also a potential challenge for developers of debuggers. Third, we use four programs written by us that cover newer language features, such as `let` or `const`, as these features are not used in the other programs. To ensure that the programs-to-debug are deterministic, we fix the current time and replace `Math.random` with a deterministic function.

5.2 Qualitative Analysis

Applying DBDB to the 41 programs-to-debug reveals various diverging behaviors between debuggers. They range from clear bugs in one of the debuggers to underspecified behavior, where neither of the debuggers is clearly wrong, but that is nevertheless interesting. We have found 20 clear bugs, 19 of which were previously unknown and which we have subsequently reported to the developers of Firefox and Chromium. Seven of the reported bugs have already been fixed. The following discusses a selection of diverging behaviors in addition to the ones presented earlier in the paper.

Chromium Issue 730177. Example 1 in Table 1 shows a bug related to breakpoints in the Chromium debugger. A breakpoint is set in the last line of the `3d-cube.js` SunSpider program, shown as an excerpt. Although line 4 is empty, we would expect the breakpoint to stay there since the program ends with this line. Firefox exhibits the correct behavior, but Chromium moves the breakpoint to the first line of the program. This diverging behavior was caught by DBDB, we subsequently submitted a bug report, which has been confirmed and fixed.[11]

Firefox Issue 1362403.[12] Example 2 demonstrates a bug in the pausing and stepping behavior of a debugger. By comparing pause locations between debuggers, DBDB has found that the Firefox debugger does not step through each iteration of a for-in-loop in this excerpt from the SunSpider program `regexp-dna.js`. In particular, when paused at the loop header (line 2), issuing only two Step In actions (⤓) takes the debugger past the loop to line 5, even though its body executes more than once. Chromium behaves correctly and pauses twice (and more often) at line 3.

Firefox Issue 1362432.[13] In Example 3, the debugger severely misrepresents the actual program state during execution. It was found by DBDB when debugging one of the aforementioned JavaScript puzzlers. JavaScript allows developers to repeat parameter names when declaring functions. Inside the function `foo`, `param` should be bound to the second supplied argument (`"second"`). The runtime behavior is correct (confirmed by the output), but the Firefox debugger shows the variable with the wrong value `"first"`.

Besides clear misrepresentations of the actual runtime behavior, we have also found other diverging behaviors between the Firefox and Chromium debuggers. These diverging behaviors are also valuable to detect; firstly, because diverging behaviors are confusing when switching debuggers and secondly, because they indicate that the intended debugger behavior is not well-specified.

Possible Breakpoint Locations. Several instances of underspecified behavior are related to the question where it should be possible to set breakpoints. Example 4 in Table 1 demonstrates that Firefox allows to set a breakpoint at the literal `true` in line 4 (and subsequently pauses there when execution is started), whereas Chromium slides the breakpoint to the next line. Neither is clearly wrong and it is open for specification whether setting breakpoints at all function arguments should be possible (for consistency and so that developers can inspect each individually) or only at non-literals (because it is unclear what is actually executed at literals).[14] DBDB found several more such cases, e.g., Firefox allows to set breakpoints at `while(true)`, but Chromium does not.

Step Semantics and Whitespace. Another large class of underspecified behaviors is related to steps. While Firefox and Chromium agree on stepping over a single function call, it is not clear what the correct behavior should be when stepping over other statements. The last example in Table 1 shows that a single step in Firefox jumps over multiple statements if they are in a single line. Chromium, on the other hand, steps over each statement individually and thus pauses multiple times in the same line. Even to a Firefox developer, the correct intended behavior was not clear.[15] Phrased more generally, it is open whether debugging should be altogether "independent" of whitespace (or other non-semantic tokens). That is, should, e.g., the number of steps to reach some statement always be the same, even if such tokens are inserted?

Overall, the examples illustrate that the diverging behaviors found by DBDB affect all kinds of actions and outputs our debugger model

[10] https://github.com/cyrus-and/chrome-remote-interface/
[11] https://bugs.chromium.org/p/chromium/issues/detail?id=730177
[12] https://bugzilla.mozilla.org/show_bug.cgi?id=1362403
[13] https://bugzilla.mozilla.org/show_bug.cgi?id=1362432
[14] See comment at https://bugzilla.mozilla.org/show_bug.cgi?id=1370641#c4.
[15] See comment at https://bugzilla.mozilla.org/show_bug.cgi?id=1370655#c2.

Table 1: Examples of detected diverging behaviors.

ID	Affected debugger	Excerpt of program-to-debug	Excerpt of trace with detected **difference**	Description

1	Chromium	`1 // Beginning of program`
		`2 DisplArea = null;`
		`3 EOF`
			Set breakpoint at line 3
			Breakpoint set at **line 1**
				Breakpoint set in empty last line wraps around and is set at beginning of program.

2	Firefox	`1 for (k in subs)`
		`2   dnaInput = ...`
		`3`
		`4 var expectedDNAOutputString = ...`
			Paused at line 2...
			Step in
			Paused at line 3...
			Step in
			Paused at **line 5**
				Step in pauses only once at for-in-loop body, even though multiple iterations are executed. Gives impression that loop is executed only once.

3	Firefox	`1 function foo(param, param) {`
		`2   console.log(param);`
		`3 }`
		`4 foo("first", "second");`
			Paused at line 2 ...
			Variables:
			param = **"first"**
				Function parameter with repeated name is shown with wrong value (but execution is correct).

4	Both	`1 function bar() { ... }`
		`2 function foo(a, b) {}`
		`3 foo(`
		`4   true,`
		`5   bar()`
		`6 );`
			Set breakpoint at line 4
			Breakpoint set at **line 4/5**
			...
			Start execution
			Paused at **line 4/5**
				Firefox allows setting breakpoints at literals and also pauses there, Chromium does neither.

5	Both	`1 a = "foo"; b = "bar"; c = "baz";`	Paused at line 2 Step over Paused at **line 3/2**	Step over in Firefox pauses at next line, even if current line has multiple statements. Chromium steps once per statement.

considers. Out of the 20 found bugs, eight are related to setting and removing breakpoints, e.g., when breakpoints cannot be set but are slided away from valid program constructs, or when breakpoints cannot be removed. Seven bugs are related to stepping and pausing, e.g., when the debugger does not halt at breakpoints or after steps, or when the debugger pauses too often or even at dead code (see Figure 1). Finally, five bugs are related to other program state, e.g., when the debugger does not show some variables at all or shows them with wrong values.

5.3 Quantitative Analysis

5.3.1 Effectiveness of Finding Diverging Behavior. We evaluate the effectiveness of DBDB by means of three questions:

- How often does the approach find diverging debugger behavior?
- How many actions are required to find diverging behavior?
- Which types of diverging behavior are the most common?

We can answer all three questions with the help of Figure 7. The x-axis shows how many actions have been generated. In the leftmost case, only breakpoint actions were generated and the program-to-debug is not yet running. At $x = 1$, only the start execution action is generated and from $x > 1$ resumes and steps are generated as well. The y-axis shows how many test runs have (cumulatively) completed after x actions. A differential test run completes either because the program-to-debug finishes executing without finding diverging behavior ("program finished", green part of the bar) or because of diverging debugger behavior (the rest of the bar, all other colors).

Effectiveness. After at most $x = 20$ generated execution actions (i.e., start execution, resume, and steps), we have found some type of diverging behavior in 82.5% of the runs, which substantiates our claim that differential testing is effective for debuggers. The high number of found diverging behaviors (1692 in total) has also motivated assigning them into equivalence classes for more effective manual inspection.

Number of Actions. We see at $x = 0$ that only by setting breakpoints and without starting the programs-to-debug, 47.9% (983) of the test runs already find diverging behavior between debuggers. After a single start execution action ($x = 1$), an additional 6% (124) of the test runs find diverging debugger behavior not due to breakpoints. Several programs-to-debug (5.4%, 112) also immediately finish execution, presumably because all breakpoints were set in not executed code. With each additionally generated action, more diverging behaviors are found, which affirms that debugger testing is more effective when combining several debugging actions. After five actions, however, the graph quickly saturates and additional actions find only marginally more diverging behaviors. After 20 generated execution actions, 99.7% (2236) of the runs have completed and only 14 runs have to be stopped because the maximum number of actions is reached. This makes clear that 20 execution actions is sufficient in most executions.

Difference Types. As said before, most diverging behaviors (47.9%, 983) are found without ever executing the program-to-debug, but simply by comparing breakpoints between debuggers. When comparing the program state between debuggers from $x \geq 1$ on, we see that the majority of differences are either because one debugger already finished executing while the other is still running (light blue, 8.7% at $x = 20$) or because the debuggers do not agree on their pause location (dark blue, 18.8%). Both cases are indicative

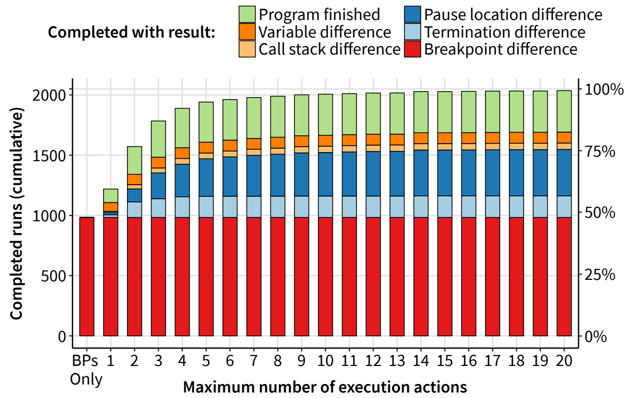

Figure 7: Completed test runs (and their results) per maximum number of generated execution actions.

Table 2: The top 6 equivalence classes of diverging behavior.

Diff. Type	Last Action	AST Node	Size	in %
Set breakpoint	Set breakpoint	`Program`	538	31.8%
Set breakpoint	Set breakpoint	`ArrayExpr`	72	4.3%
Set breakpoint	Set breakpoint	`BinaryExpr`	64	3.8%
Paused line	Step in	`ForStmt`	61	3.6%
Set breakpoint	Set breakpoint	`Property`	55	3.3%
Paused line	Step over	`ForStmt`	51	3.0%

of a debugger either not pausing where it should or pausing too often. Only a minority of the differences come from comparing call stack and variables (2.5% and 4.4%, respectively), which correlates with the smaller number of bugs we have found related to these debugger features.

5.3.2 Effectiveness of Equivalence Class Sampling. The large number of found diverging behaviors cannot all be manually inspected and we thus need to restrict ourselves to a subset. In Section 3.4, we argued that more diverse diverging behaviors can be found by first assigning each diverging behavior to an equivalence class and then sampling these classes in a round robin manner for manual inspection. We evaluate the effectiveness of this heuristic through two questions:

- How many equivalence classes exist, and is it feasible to inspect at least one diverging behavior per class?
- Does sampling diverging behaviors from these classes, instead of sampling uniformly over all diverging behaviors, help finding more unique root causes per inspected diverging behavior?

Equivalence Classes. In total there are 112 equivalence classes, which is a manageable number to manually inspect, in particular compared to the 1,692 total number of traces with diverging behavior. This strong reduction is mainly caused by some large equivalence classes, the top six are shown in Table 2. The largest equivalence class contains almost a third of all divergent behaviors. As indicated by the difference type and last action, these diverging behaviors are caused by setting a breakpoint at the `Program` AST node. `Program` is the root of the JavaScript AST, so this means a breakpoint was set in an empty or comment line. Almost all divergent behaviors in this class are instances of a single bug in Chromium related to breakpoint sliding for empty lines and comments.[16] Similarly, the equivalence classes of setting breakpoints at arrays, binary expressions, or object properties are large because Firefox allows to set breakpoints at literals (which are common at these AST nodes), whereas Chromium does not. Finally, we see two large classes related to stepping and `ForStmt`, which instances of the difference in step semantics between Firefox and Chromium that we explained in Section 5.2.

More Effective Manual Inspection. In response to the second question, we evaluated the number of unique root causes we find when uniformly sampling diverging behaviors compared to when we sample out of the equivalence classes. A root cause is either an ID from the issue trackers for bugs that we reported, or an identifier for consistent semantic differences we found between Firefox and Chromium. Figure 9 shows that manually inspecting 50 diverging behaviors from equivalence classes have led us to identify 24 unique root causes, whereas uniformly sampling from all diverging behaviors uncovers only 12. One explanation for this is rooted in the large equivalence classes discussed before. When uniformly sampling diverging behaviors, we get many instances out of these large classes that are mostly caused by just a single bug.

Regarding the time spent on analyzing diverging behaviors from equivalence classes and reporting all bugs (including manually creating minimal test cases), we estimate the effort as less than a week for one person. We do not evaluate the true positive rate of DBDB because defining true positives is difficult in the absence of a clear specification of expected debugger behavior.

5.3.3 Performance. We evaluate the runtime of our approach to assess whether DBDB could scale to many more programs-to-debug. Since the assignment of diverging behaviors to equivalence classes is performed just once after all differential testing completed and takes less than two seconds, we exclude it from our measurements. To evaluate the differential testing of our approach, we debug each of the 41 programs 50 times with different seeds. That is, even though the program execution itself remains the same, DBDB sets different breakpoints and performs, e.g., different steps or resume actions. Figure 8 summarizes the runtimes of the approach. The average test run took 448 ms, so performance-wise our approach can be applied to many more programs-to-debug. Obviously the runtime includes execution of the program-to-debug itself, which is why some programs taking consistently more time than others.

6 RELATED WORK

Compiler Testing. Compiler testing has a long history [5, 11, 34]. More recent approaches include CSmith [39], which randomly creates C programs, and "equivalence modulo inputs" [20], which creates variants of such programs with supposedly equivalent behavior. Other work empirically compares compiler testing techniques [7], prioritizes generated test programs to execute bug-revealing programs earlier [6], generates test programs guided by a type- and effect-system [26], and enumerates all test programs within some bounds [40]. Lidbury et al. [22] and Donaldson et al. [10] target compilers for OpenCL and GLSL programs running on GPUs. All of these approaches put an emphasis on generating programs as

[16] https://bugs.chromium.org/p/chromium/issues/detail?id=784852

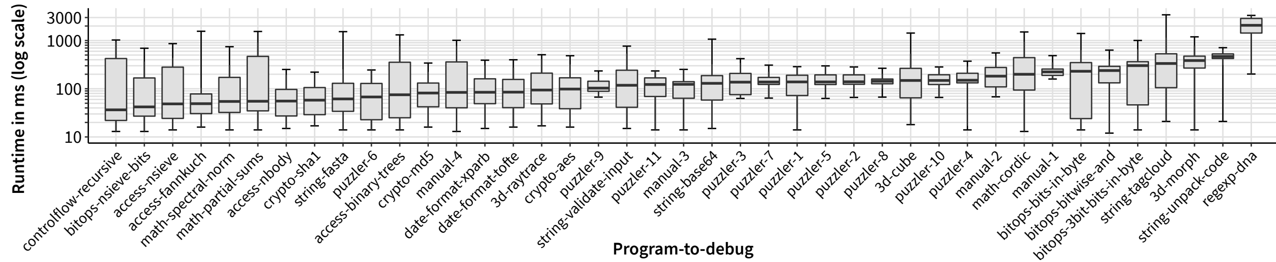

Figure 8: Runtime of the differential testing per program-to-debug. Since actions are randomly generated, medians (middle band) and first and third quartile (bottom and top of box) are shown after 50 runs. Whiskers indicate minimum and maximum.

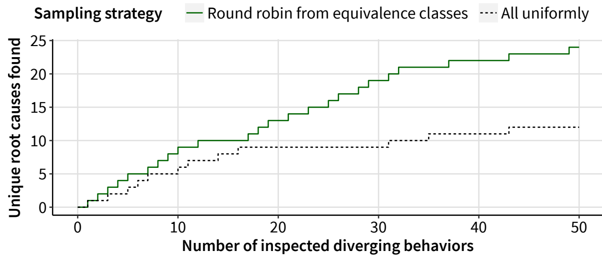

Figure 9: Found unique root causes when sampling difference equivalence classes compared to raw differences.

inputs for testing, whereas our work on debuggers also considers the problem of how and when to create user actions.

Differential Testing. The term “differential testing” was originally coined by McKeeman [24]. Apart from compilers, it has been applied to other development tools, such as refactoring engines [9], symbolic execution engines [17], x86 disassemblers [28], and binary lifters [18]. Our work differs by generating not only programs but also debugger actions associated with the program, and by using execution feedback to guide the generation. Beyond developer tools, differential testing can, e.g., be applied to code clones [41], similar library implementations [37], and supposedly behavior-preserving subclasses [32]. A related concept is *N-version-programming*, where multiple versions of a program are created to increase fault tolerance. Experiments with it have shown that similar errors were made by independent programmers [19], hinting at a potential shortcoming of our approach: if both debuggers have the same bug, DBDB cannot find it.

Inspecting Warnings. A recurring problem in automated testing is the high number of test cases that need to be manually inspected. Our heuristic of drawing diverging behaviors from equivalence classes instead of uniformly sampling relates to work by Podgurski et al. [31], who try to cluster software faults. For evaluating our equivalence classes, we use a method similar to Chen et al. [8]. Another way to improve how users inspect warnings reported by a tool is to ask specific questions that may eliminate false alarms [42].

Interactive Testing. Unlike compilers, debuggers are interactive, which precludes generating all testing inputs beforehand. Our interactive approach that alternates between generating inputs and comparing outputs is similar to prior work on automated testing of GUIs [13, 15, 23], which are also interactive.

(Cross-)Browser Testing. Our implementation of DBDB targets the debuggers in Firefox and Chromium. As such, it is also in the line of many works that perform cross-browser testing. Roy Choudhary et al. [36] visually compare websites to identify issues in rendering by different browsers and Mesbah and Prasad [25] additionally incorporate user interaction into the comparison of browser behavior. TreeFuzz compares JavaScript implementations of browsers by fuzz-generating JavaScript programs [29].

Debugger and Tooling Correctness. An important step to improve the correctness of debuggers is specifying their intended behavior. Bernstein and Stark [3] formally define the semantics of a debugger for a small functional language, but not for a debugger used in practice. The differences discovered by DBDB may help to identify situations that require more precise specification. Similar in spirit to our work is the vision of Cadar and Donaldson [4]. They postulate that even though much effort goes into developing program analyzers, the tools themselves are often not put under enough scrutiny. They call for “analyzing the program analyzer” and our first step is to find bugs in debuggers.

7 CONCLUSION AND OUTLOOK

This paper presents the first approach for automatically testing interactive debuggers. DBDB compares the behavior of two debuggers by exercising them with generated sequences of debugging actions. Our work builds upon a finite-state model that captures common features of real-world debuggers. We evaluate DBDB with the JavaScript debuggers of Firefox and Chromium, where we find 20 clear bugs and several other noteworthy differences. Eight of these bugs have already been fixed by the respective developers, and our results have spurred discussions about the intended semantics of debuggers. While testing compilers has received significant attention, we hope this work motivates more researchers to also put other development tools under scrutiny. It is worrying that debuggers, a fundamental development tool for understanding programs, are themselves buggy and even more so that the intended semantics of seemingly simple features, such as stepping, are not precisely specified. Finding behavioral differences between existing debuggers is a first step towards fixing these problems.

ACKNOWLEDGMENTS

This work was supported by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the German Research Foundation within the ConcSys and Perf4JS projects.

REFERENCES

[1] Technical Committee ISO/IEC JTC 1/SC 22. 2011. *ISO/IEC 9899:2011 – Information technology – Programming languages – C* (third ed.). Standard. Available from: https://www.iso.org/standard/57853.html. Accessed: 2017-11-15.

[2] Technical Committee ISO/IEC JTC 1/SC 22. 2014. *ISO/IEC 14882:2014 – Information technology – Programming languages – C++* (fourth ed.). Standard. Available from: https://www.iso.org/standard/64029.html. Accessed: 2017-11-15.

[3] Karen L. Bernstein and Eugene W. Stark. 1995. Operational Semantics of a Focusing Debugger. *Electronic Notes in Theoretical Computer Science* 1 (1995), 13–31. https://doi.org/10.1016/S1571-0661(04)80002-1

[4] Cristian Cadar and Alastair F. Donaldson. 2016. Analysing the Program Analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 765–768. https://doi.org/10.1145/2889160.2889206

[5] David Callahan, Jack J. Dongarra, and D. Levine. 1988. Vectorizing compilers: a test suite and results. In *Proceedings Supercomputing '88, Orlando, FL, USA, November 12-17, 1988*. 98–105. https://doi.org/10.1109/SUPERC.1988.44642

[6] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 700–711. https://doi.org/10.1109/ICSE.2017.70

[7] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 180–190. https://doi.org/10.1145/2884781.2884878

[8] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming Compiler Fuzzers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 197–208. https://doi.org/10.1145/2491956.2462173

[9] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1287624.1287651

[10] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133917

[11] Jack J. Dongarra, Mark Furtney, Steven P. Reinhardt, and J. Russell. 1991. Parallel loops - a test suite for parallelizing compilers: description and example results. *Parallel Comput.* 17, 10-11 (1991), 1247–1255. https://doi.org/10.1016/S0167-8191(05)80036-5

[12] Technical Committee 39 Ecma International. 2017. Test262: ECMAScript Test Suite (ECMA TR/104). https://github.com/tc39/test262. Accessed: 2017-11-15.

[13] Markus Ermuth and Michael Pradel. 2016. Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events. In *International Symposium on Software Testing and Analysis (ISSTA)*. 82–93.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (USENIX Security 12)*. USENIX Association, Berkeley, CA, USA, 38–38.

[15] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. ACM, New York, NY, USA, 77–83. https://doi.org/10.1145/1982595.1982612

[16] Ecma International. 2017. *Standard ECMA-262 – ECMAScript 2017 Language Specification* (8th ed.). Standard. Available from: https://www.ecma-international.org/publications/standards/Ecma-262.htm. Accessed: 2017-11-15.

[17] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 590–600.

[18] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 353–364.

[19] John C. Knight and Nancy G. Leveson. 1986. An Experimental Evaluation Of The Assumption Of Independence In Multi-Version Programming. *IEEE Transactions on Software Engineering* 12 (1986), 96–109.

[20] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[21] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[22] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2737924.2737986

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[24] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[25] Ali Mesbah and Mukul R. Prasad. 2011. Automated Cross-Browser Compatibility Testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 561–570. https://doi.org/10.1145/1985793.1985870

[26] Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *PACMPL* 1, ICFP (2017), 15:1–15:23. https://doi.org/10.1145/3110259

[27] Mehryar Mohri, Fernando Pereira, and Michael Riley. 2002. Weighted finite-state transducers in speech recognition. *Computer Speech & Language* 16, 1 (2002), 69–88. https://doi.org/10.1006/csla.2001.0184

[28] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 265–274.

[29] Jibesh Patra and Michael Pradel. 2016. *Learning to Fuzz: Application-Independent Fuzz Testing with Probabilistic, Generative Models of Input Data*. Technical Report TUD-CS-2016-14664. TU Darmstadt.

[30] Filip Pizlo. 2013. Announcing SunSpider 1.0. https://webkit.org/blog/2364/announcing-sunspider-1-0/. Accessed: 2017-11-15.

[31] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated Support for Classifying Software Failure Reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*. IEEE Computer Society, Washington, DC, USA, 465–475. http://dl.acm.org/citation.cfm?id=776816.776872

[32] Michael Pradel and Thomas R. Gross. 2013. Automatic Testing of Sequential and Concurrent Substitutability. In *International Conference on Software Engineering (ICSE)*. 282–291.

[33] LLVM Project. 2017. LLVM Testing Infrastructure Guide. https://llvm.org/docs/TestingGuide.html. Accessed: 2017-11-15.

[34] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972), 366–375.

[35] Norman Ramsey. 1994. Correctness of Trap-based Breakpoint Implementations. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/174675.175188

[36] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/ICSM.2010.5609723

[37] Marija Selakovic, Michael Pradel, Rezwana Karim Nawrin, and Frank Tip. 2018. Test Generation for Higher-Order Functions in Dynamic Languages. In *OOPSLA*.

[38] GCC team. 2017. GCC Testing Efforts. https://gcc.gnu.org/testing/. Accessed: 2017-11-15.

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[40] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*.

[41] Tianyi Zhang and Miryung Kim. 2017. Automated transplantation and differential testing for clones. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 665–676. https://doi.org/10.1109/ICSE.2017.67

[42] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective interactive resolution of static analysis alarms. *PACMPL* 1, OOPSLA (2017), 57:1–57:30. https://doi.org/10.1145/3133881