# Interactive Metamorphic Testing of Debuggers

Sandro Tolksdorf
sandro@sandrotolksdorf.de
TU Darmstadt, Germany

Daniel Lehmann
mail@dlehmann.eu
TU Darmstadt, Germany

Michael Pradel
michael@binaervarianz.de
TU Darmstadt, Germany

## ABSTRACT

When improving their code, developers often turn to interactive debuggers. The correctness of these tools is crucial, because bugs in the debugger itself may mislead a developer, e.g., to believe that executed code is never reached or that a variable has another value than in the actual execution. Yet, debuggers are difficult to test because their input consists of both source code and a sequence of debugging actions, such as setting breakpoints or stepping through code. This paper presents the first metamorphic testing approach for debuggers. The key idea is to transform both the debugged code and the debugging actions in such a way that the behavior of the original and the transformed inputs should differ only in specific ways. For example, adding a breakpoint should not change the control flow of the debugged program. To support the interactive nature of debuggers, we introduce interactive metamorphic testing. It differs from traditional metamorphic testing by determining the input transformation and the expected behavioral change it causes while the program under test is running. Our evaluation applies the approach to the widely used debugger in the Chromium browser, where it finds eight previously unknown bugs with a true positive rate of 51%. All bugs have been confirmed by the developers, and one bug has even been marked as release-blocking.

## CCS CONCEPTS

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

## KEYWORDS

Interactive Debuggers, Metamorphic Testing, JavaScript

**ACM Reference Format:**
Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive Metamorphic Testing of Debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330567

## 1 INTRODUCTION

Interactive debuggers are an integral part of software development since many developers rely on them to understand a program's behavior and for finding and fixing bugs. They are a powerful tool that allows developers to stop execution at points of interest through *breakpoints*, to inspect the intermediate program state such as the values of local variables, and to closely follow the control-flow through *single stepping*.

What happens if this essential development tool is buggy itself? First, bugs in debuggers make it harder to find and fix bugs in programs. Consider the case of a variable not shown in the debugger, although it should be. (We found such a bug with our approach in the Chromium debugger.[1]) A developer might assume that the variable is not present due to a problem in the code, while in fact this is purely a problem in the debugger. This would clearly slow down a developer investigating the issue. Second, as a consequence of a debugger misrepresenting the actual execution of a program, developers might change the program for the worse, i.e., introduce "wrong fixes". Consider the case of a debugger not pausing at a breakpoint, even though it should. (Another bug we found with our approach.[2]) This behavior might lead a programmer to believe the line with the breakpoint was not executed at all and in turn incorrectly change the program.

*Testing Debuggers.* One way to find bugs in debuggers is testing, specifically automated or random testing. There, test inputs are randomly generated and a *test oracle* [2] decides whether the test was successful or failed. Recent work has introduced an automated testing technique for debuggers [19]. As the input, it randomly generates *debugging actions* (such as setting a breakpoint, resuming execution, or steps) from a grammar and executes them on a given program-to-debug. For the test oracle, it applies *differential testing* to debuggers, which has been very successful in the past at testing other programmer tooling, such as compilers [22, 29] and refactoring engines [11]. As the name implies, the oracle in differential testing comes from comparing two different implementations against each other. E.g., the JavaScript debuggers in Firefox and Chromium can be given the same program and debugging actions and their resulting behavior (e.g., where they pause execution and which variables are shown) compared against each other.

The main requirement for differential testing is that one needs those two implementations and that they must be comparable to each other, i.e., a difference in their outputs should truly point to a bug, not simply a disagreement that is allowed as per the problem specification. E.g., in the case of compiler testing, much effort has to be put into avoiding generating test inputs that are not well-specified (e.g., C and C++ programs containing *undefined behavior*). Similarly, differential testing of debuggers suffers from underspecified behavior in debuggers. One such instance is *breakpoint sliding*. E.g., Firefox' and Chromium's debugger might disagree on where to move a breakpoint that was requested at a comment line, but neither of them is clearly wrong or right [19].

*ISSTA '19, July 15–19, 2019, Beijing, China*

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330567

[1] https://bugs.chromium.org/p/chromium/issues/detail?id=901816
[2] https://bugs.chromium.org/p/chromium/issues/detail?id=892622

```
1 ▶ // 1. paused here
2    // -> action: continue
3 foo();
4 function foo() {
5 ▶ stmt; // 2. pauses here
6 } // because of breakpoint
```

**(a) Initial test run.**

```
1 ▶ // 1. paused here
2    // -> action: step over
3 foo();
4 function foo() {
5 ▶ stmt; // 2. pauses here
6 }       //    as well
```

**(b) Follow-up test run.**

**Figure 1: Example of a metamorphic test case for debuggers.**

*Metamorphic Testing.* Some of the problems of differential testing are solved by *metamorphic testing* [7, 25], another automated testing technique. The core difference between metamorphic and differential testing is that instead of giving the same input to two different implementations (which might have differences simply due to underspecification) and comparing the results, one gives two different inputs to the same program and compares the results of the two different runs. The key realization is that oftentimes a program $P$ produces similar outputs on inputs $A$ and $B$ if the inputs are already in a specific relation $R_I$ with each other. That is, for a program under test $P$, a specific input relation implies a specific output relation: $A\ R_I\ B \Rightarrow P(A)\ R_O\ P(B)$.

If we have two such inputs that are related by $R_I$, but the outputs are not related by $R_O$, we have found a potential bug in $P$. A common way to obtain two such inputs related by $R_I$, is to transform the input $A$ into $B$ such that the relation holds. E.g., in metamorphic testing of compilers, a transformation could insert dead code into a given program and one would expect that the produced binaries (i.e., the compiler output) still behave the same.

Metamorphic testing could also work for debuggers, but to the best of our knowledge, no existing approach has applied this idea to them. The closest existing work is testing compilers [12, 17], but as we show in the following, testing debuggers entails challenges not present for compilers or other software.

Figure 1 gives an example of a metamorphic test case for debuggers. On the left, we see a debugging session, where a continue is issued at a pause in line 1. This pauses the debugger next in line 5, due to a breakpoint. In the follow-up test run, a metamorphic transformation has replaced the continue with a step over. Because of the breakpoint in line 5, the debugger behavior is the same as before. If the behavior were not equal, the breakpoint in line 5 might have been ignored, which would constitute a bug in the debugger.

*Challenges.* We apply the idea of metamorphic testing to interactive debuggers, which comes with several interesting challenges.

First, debuggers take not just a program as input, but also debugging actions such as breakpoints, steps, and continues. Since inputs consist of two parts, we also have two separate types of metamorphic transformations: *program transformations* and *action transformations*. Because debugging actions refer to code locations, e.g., for setting breakpoints, these two transformations need to be consistent with each other. E.g., if a program transformation inserts dead code, the debugging actions have to be updated as well.

Second, whereas metamorphic transformations for programs are well known from compiler testing, to the best of our knowledge, nobody has looked into metamorphic transformations for interactive debuggers. E.g., one can safely assume that inserting dead code does not change the behavior of a program in compiler testing, but the behavior of a debugger might legitimately change, e.g., because now more breakpoints can be set than before or because *breakpoint sliding* moves previous breakpoint to now different locations.

Finally, unlike compilers and many other programs, debuggers are inherently interactive. That is, debugging actions are not given to the debugger once before test execution, but continuously throughout the debugging session. For a similar reason, metamorphic transformation of the debugging actions sometimes depends on the current state of the debugger, which precludes transforming the debugging actions "offline" before the test execution, as in traditional metamorphic testing. Instead, the debugging actions must be transformed during the testing itself, i.e., in an interactive fashion.

*Approach.* This paper presents our approach on *interactive metamorphic testing* of debuggers. Similar to previous work on debugger testing [19], the input to the debugger under test is a program-to-debug and initially randomly generated debugging actions, such as setting breakpoints, continues, and steps. However, unlike in previous work that used differential testing, we employ metamorphic testing, i.e., inputs for a follow-up test run are generated by metamorphic transformations. Then, our tool compares the debugger behavior in the initial run with the follow-up run based on debugging traces that were collected during testing. If an unexpected difference is found, this might indicate a bug in the tested debugger.

As part of our approach, we present program transformations and new action transformations to test debuggers, e.g., a transformation that adds breakpoints and matching continues or one that replaces continues with steps under certain conditions. These *interactive transformations* decide depending on the current state of the debugger how an action in the initial test input shall be transformed for the follow-up run.

We implement our approach for the JavaScript debugger in the widely used Chromium browser and evaluate it on 47,342 JavaScript programs. Each input program and initial sequence of debugging actions goes through several iterations of metamorphic transformations, up to five times or until an unexpected difference is encountered. In total, our tool has produced 59 warnings, 30 of which we manually confirmed to be true positives. This results in a true positive rate of 51%, which we believe to be acceptable for an automatic approach to find bugs in such an essential tool that debuggers are. We have also reported nine bugs found by our tool to the Chromium developers. Eight of those bug reports were confirmed and one of those is even marked as release-blocking, which we take as an indication that our approach is useful in practice.

*Contributions.* In summary, this paper contributes the following:

- We are the first to adapt metamorphic testing to debuggers, for which we introduce interactive transformations that need to happen intertwined with test execution (Section 3).
- We present several metamorphic transformations on programs and actions specific to debuggers in Section 3.2. These transformations are a step towards specifying the expected behavior of debuggers, which is currently mostly given informally as part of user-level documentation.
- We implement and evaluate the approach for a widely used debugger, where it found eight previously unknown bugs.

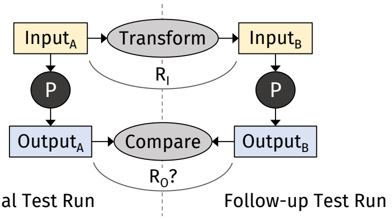


**Figure 2: Metamorphic testing overview.**

## 2 BACKGROUND

Before we come to our approach in the next section, we want to introduce the key concepts behind metamorphic testing and testing of interactive debuggers.

### 2.1 Metamorphic Testing

Figure 2 shows the main principle behind metamorphic testing [7, 25]. In the *initial test run*, shown in the left half of the figure, the program under test $P$ is executed on a given test input $Input_A$ and produces $Output_A$. For the *follow-up test run*, shown on the right half, we obtain $Input_B$ by transforming $Input_A$, such that the two inputs are related by $R_I$. In the final step of metamorphic testing, the two outputs $Output_A$ and $Output_B$ are compared against each other. If relation $R_O$ does not hold between the two outputs, metamorphic testing has found a potential bug. Note that metamorphic testing always needs two executions of $P$ (which we call *test runs*) and subequent comparison of their outputs to find a bug. In the remainder of this paper, we call this pair of test runs a *metamorphic test case*, or test case for short.

For simple transformations, e.g., the one already shown in Figure 1, the output relation $R_O$ is plain identity. That is, the debugger's behavior on the transformed, follow-up inputs should be exactly equal to the behavior on the initial inputs. However, for more complex input transformations, e.g., when the program-to-debug is modified, the output relation needs to account for changes introduced by the input transformation. An example of such changes are locations referred to by the debugger, e.g., where it is currently paused. If the input transformation changed the underlying program by inserting or removing lines, it is expected that these changes reflect in the debugger's outputs, namely its reported locations will be offset by the number of inserted or removed lines. Our comparison between initial and follow-up run must account for those differences.

### 2.2 Testing Interactive Debuggers

As described in previous work on automated testing of debuggers [19], testing debuggers is different from testing other programs, such as compilers, in several ways.

First, debuggers take two types of inputs instead of just one: the *program-to-debug* (e.g., as JavaScript source code) and *debugging actions*, or actions for short. Debugging actions are user commands that steer the debugging session, such as setting breakpoints and stepping through the program. For automated testing, these debugging actions need to be generated. In our case, the actions for the initial test run come from DBDB, an existing approach to automated debugger testing [19]. As a brief overview of the possible debugging actions, Figure 3 also gives a grammar. Note that every sequence of *Action*s must start with adding at least one breakpoint, otherwise the debugger will never pause during testing.

(Debugging Actions)

*Actions* ::= Add breakpoint at *Location*; *DebuggingAction*$^*$

*DebuggingAction* ::= *BreakpointAction* | *ControlAction*

*BreakpointAction* ::= Add breakpoint at *Location* |
Remove breakpoint at *Location*

*ControlAction* ::= Continue | Step in | Step out | Step over

*Trace* ::= *DebuggerOutput*$^*$ (Debugging Trace)

*DebuggerOutput* ::= *BreakpointOutput* | *PauseOutput*

*BreakpointOutput* ::= Breakpoint set at *Location*

*PauseOutput* ::= Paused at *Location* with *Variable*$^*$

*Variable* ::= (*name*, *value*)

*Location* ::= *line*:*column* (Common Variables)

*line* ∈ ℕ, line numbers

*column* ∈ ℕ, column numbers

*name* ∈ identifiers

*value* ∈ primitive value or JavaScript object

**Figure 3: Grammars for debugging actions and traces.**

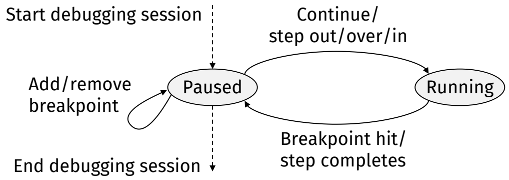


**Figure 4: Debugger as a finite state machine with two states.**

Second, unlike many programs tested previously by metamorphic testing [25], debuggers are *interactive*. That means during a debugging session, the debugging actions are issued interleaved with the execution of the program-to-debug and not all at once before starting the debugger. For more detail, Figure 4 provides a "zoomed-in" view of the debugger during testing. We see that the debugger alternates between two states, *paused* and *running*:

- Debugger testing begins and ends in the paused state, where the program-to-debug is either not yet started, paused in the middle of execution, or finished. Whenever in this state, we record the current debugger output and then issue the next debugging action (either from DBDB in the initial test run or from the transformation in the follow-up run). Adding and removing breakpoints (*BreakpointAction*s), keep the debugger in the paused state. Only after a *ControlAction*, namely a *continue* or one of three types of *step*s, execution resumes and the debugger transitions to the running state.
- The debugger remains in the running state until either a breakpoint is hit, a single step completes, or execution finishes, upon which it transitions back to the paused state.

Finally, what is the output of a debugger? That is, what is recorded in the debugger's pause state and what is compared between the two test runs? The middle part of Figure 3 gives a grammar for

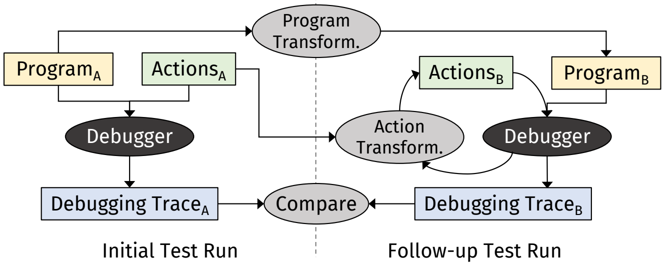


**Figure 5: Overview of metamorphic testing for debuggers.**

what we call the *debugging trace*. It contains two types of debugger outputs: First, *BreakpointOutput*, which is the result of adding a breakpoint in the debugger. While users can request breakpoints at arbitrary locations in the source code, not all of those correspond to meaningful operations. Thus, debuggers usually move the breakpoint to the next "appropriate" location, which is called *breakpoint sliding* and captured herein. Second, *PauseOutput* records the state of the debugger in a pause at a breakpoint or after a step. It contains the *Location* where it is paused and local and global variables shown in the debugger. In principle our approach is independent of the programming language, so how variables are captured depends on the implementation. In our case of JavaScript debuggers, we capture all primitive values and recursively copy objects and their properties (up to a threshold depth) into the trace.

## 3 APPROACH

In this section, we describe our approach in more detail. We start by giving an overview in Section 3.1. Then, Section 3.2 explains the metamorphic transformations we apply to the program and debugging actions. Finally, Section 3.3 introduces an extension that iteratively applies multiple transformations after each other.

### 3.1 Overview

Figure 5 shows that our approach, as in regular metamorphic testing, has two main phases: an initial test run and a follow-up test run, shown in the left and right half, respectively. Apart from the two test runs, we also see two transformations (for the program and debugging actions) and the comparison of the debugging traces after the test runs. These components execute in the following order: First, the initial test is run to obtain an initial debugging trace. Then, the program transformation delivers the first part of the input for the follow-up run. The follow-up test then runs interlocked with the action transformation, since the latter depends on the current debugger state. And finally, the two debugging traces are compared.

The input to our overall approach are the debugger to be tested and the debugger inputs for the initial test run. In our case, we do not generate the programs randomly, but instead take them from a fixed, but large and diverse test set (mostly the official test suite for ECMAScript compliance, test262, but also some benchmarks and other sources). For generating the debugging actions, we build on previous work on automated testing of debuggers [19]. This tool, called DBDB, produces a sequence of debugging actions with configurable length for the initial test run.

**Table 1: Metamorphic transformations for debuggers.**

| Transformation | Short Description |
|---|---|
| *Action Transformations:* | |
| Add breakpoint and continue | Adding breakpoint at line $l$ should have no influence other than additional pauses at $l$. |
| Replace continue by step | Control actions have a subset relation (continue ⊂ step out ⊂ step over ⊂ step into) based on where they pause. Thus, sometimes, a continue can be replaced by a step. |
| Breakpoint sliding | Setting breakpoint at line $l$ and sliding to $l'$ should be equal to directly setting it at $l'$. |
| *Program Transformations:* (Actions are transformed accordingly.) | |
| Insert dead code | Insertion of statically dead code should have no influence on pauses or program state. |
| Remove dead code | Removal of dynamically dead code should have no influence. |
| Add parameter | Additional formal parameter should have no influence other than appearing as `undefined` in the function's scope. |
| Add no-operation | Insertion of `x=x;`, where `x` is in scope, should have no influence. |
| Integer literal to expression | Replacing `n` by `(n1+n2)`, where $n = n_1 + n_2$ should have no influence. |
| Boolean literal to expression | Replacing `true` by `(x==x)` should have no influence. Analogous for `false`. |

The overall output of our metamorphic testing tool is the result of the comparison of the debugging traces. If there is an unexpected difference, our tool reports a warning to the user and saves the two debugging traces, the two programs-to-debug, and additional logging information for further analysis to disk. Although evaluation will show that the number of false positives is comparatively low, unfortunately not every warning means a clear debugger bug. Also, multiple warnings found by automated testing can point to the same root cause. Thus as the final step, we manually inspect each warning and in case of a bug, report an issue to the debugger developers (with a reduced example from the recorded traces).

### 3.2 Metamorphic Transformations for Interactive Debuggers

The input transformation in metamorphic testing of debuggers consists of two parts: a transformation of the program-to-debug and a transformation for the debugging actions. While in principle it is possible to do complex transformations on both at the same time, we instead chose metamorphic transformations that focus on one of the two parts of the input and transforms the other one only in so far, as to keep program and action consistent with each other (e.g., if there is a change in line numbers). More complex interactions between program and debugging action changes can still be obtained by combining multiple transformations in iterative metamorphic testing (Section 3.3). Table 1 gives an overview of our transformations, grouped by their main focus (program vs. actions).

```
1  function foo() {
2  ▶ bar(); // paused here
3           // -> step out
4    stmt;
5  }
6
7  foo(); // pauses here
```

**(a) Initial test run.**

```
function foo() {
▶ bar(); // paused here
         // -> step out
▶ stmt;  // interrupted!
} // -> set tmp bp at 7
  //    to resync traces
▶ foo(); // -> remove bp
```

**(b) Follow-up test run with additional breakpoint at line 4.**

**Figure 6: Add breakpoint and continue transformation is complex due to steps.**

...
*ControlAction*
Paused at $l$

**(a) Initial actions (left-aligned) and trace (right-aligned).**

Add breakpoint at $l_{new}$
...
*ControlAction*
Paused at $l_{new}$
Set temporary breakpoint at $l$
Continue
Paused at $l$
Remove temporary breakpoint at $l$

**(b) Follow-up transformed actions and expected trace.**

**Figure 7: Full *add breakpoint and continue* transformation.**

### 3.2.1 *Action Transformations.*

These transformations touch only the debugging actions and never change the program-to-debug. (Or in other words, the program transformation here is identity.)

*Add breakpoint and continue.* We expect that breakpoints are independent of each other, i.e., that adding a breakpoint to a program that already has several breakpoints set, does not modify the existing breakpoints (e.g., disables them or moves them around), nor should the new breakpoint change when existing breakpoints are hit. If there are only continues in the debugging actions, we also expect that there are only additional pauses at the inserted breakpoint, not suddenly at other program locations.

We can test this metamorphic relation with the following transformation. Our tool inserts a new action *add breakpoint at* $l_{new}$ with a random location $l_{new}$ into the actions of the initial test run. Then, during comparison of the traces between test runs, we ignore all pauses at $l_{new}$ in the follow-up trace that are not in the initial trace.

This is the first instance of an action transformation that also requires runtime information from the debugger to function. What happens if the requested breakpoint at $l_{new}$ is moved due to breakpoint sliding (e.g., because it is at an empty line)? Then, the comparison must ignore pauses at the actual location $l'_{new}$, which is only returned by the debugger during testing.

A second complication is due to step actions. The transformation so far works for breakpoints and continues. But because steps also pause at breakpoints, the full transformation needs to be more elaborate. Figure 6 gives an example. Assume the debugger is paused at line 2 in the initial run and performs a *step out*. This pauses the debugger at the call site of function `foo` in line 7, which is recorded in the initial debugging trace. Now, consider the follow-up test run in which a breakpoint was added at line 4. Again, assume the debugger is paused at line 2. The *step out* pauses at the new breakpoint in line 4, because steps are also interrupted by breakpoints. Our comparison ignores this pause at the new breakpoint. But later in the comparison we will detect a difference because the pause at line 7 of the initial test run is missing in the follow-up run. The pause is missing because the step is already "consumed" by pausing at the new breakpoint.

How do we make sure the follow-up test run pauses where the initial one paused in case of a step? Figure 7 shows the full action transformation abstractly: Whenever we are paused at the inserted breakpoint in the follow-up run (on the right), we check whether the last *ControlAction* was a step (which would be "consumed" by the new breakpoint). If so, we insert a temporary breakpoint at the line where the step in the initial run (on the left) went to. Then, we continue execution, which pauses at the temporary breakpoint and resyncs the initial and follow-up test run. Finally, the temporary breakpoint is removed again. Figure 6 also shows this concretely.

```
1  function foo() {
2  ▶ bar(); // paused here --> control action
3    stmt;   // (b) step over would pause here
4  }
5  function bar() {
6    stmt;   // (a) step into would pause here
7  }
8  foo();
9  stmt;     // (c) step out would pause here
10 ▷ stmt;  // (d) continue would pause here
```

**Figure 8: Example for the different *ControlActions*.**

**Table 2: Subset relation between the four different debugger *ControlActions* supported by most debuggers.**

| Pauses at: | Breakpoints | Next statement… | | |
|---|---|---|---|---|
| Action ↓ | | …in caller | in function | anywhere |
| Continue | ✓ | | | |
| Step Out | ✓ | ✓ | | |
| Step Over | ✓ | ✓ | ✓ | |
| Step Into | ✓ | ✓ | ✓ | ✓ |

*Replace continue by step.* In this action transformation, we test an interesting relation between the debugging actions, specifically the *ControlActions*. The underlying observation is that each *ControlAction* pauses execution at different points in the program, and that there is a subset relationship between these potential pause points. Consider Figure 8 for an example. Assume the debugger is paused at line 2 and we resume execution by a continue, step out, step over, or step into. The comments show that every action causes execution to pause at a different line of the program. However, we also see that some actions pause "earlier" than others. E.g., the step into action always goes to the next statement, even if that is inside a called function, whereas the step over action ignores statements that are deeper in the call tree than the current function. Later pausing yet again, step out ignores any statement in the current function and pauses only at the next statement in the caller of the current function. And finally, a continue will only pause at breakpoints and not at any statement mentioned earlier.

```
1  ⬚ // requested breakpoint at this comment line...
2  ▶ var x = 0; // ...is moved to next statement
```

**Figure 9: Example of breakpoint sliding in debuggers.**

Add breakpoint at $l_{requested}$
                Breakpoint set at $l_{actual}$
... (more actions)
                Paused at $l_{actual}$

**(a) Initial actions (left-aligned) and trace (right-aligned).**

Add breakpoint at $l_{actual}$
                Breakpoint set at $l_{actual}$
... (more actions)
                Paused at $l_{actual}$

**(b) Follow-up transformed actions and expected trace.**

**Figure 10: Full *breakpoint sliding* transformation.**

Table 2 summarizes this subset relationship between *ControlActions*. A continue pauses only at breakpoints. Step out pauses at breakpoints as well but additionally at the next statement of the caller of the current function. Step over pauses in the previous two cases and also at any statement in the current function. Finally, step into pauses at any statement. We can thus order *ControlAction*s by their set of "pause points", from least to most: continue ⊂ step out ⊂ step over ⊂ step into. Based on this observation, we see that any lower-order *ControlAction* can be replaced by a sequence of one or more higher-order ones, and no debugger pause will be missed. In our action transformation, we replace randomly selected continue by one of the three steps. Then, during comparison, we check that all pauses from the previous run are still present in the trace.

Similar to the add breakpoint transformation, the inserted steps themselves may not only pause at breakpoints, but additionally at their step "pause point". To re-synchronize the pauses in the traces of the initial and follow-up run, we thus additionally have to issue a continue in some cases. Whether this additional continue must be inserted, again depends on the current state of the debugger, which makes this an interactive metamorphic transformation. If after replacing the initial continue with a step, the debugger is paused at the same location, no additional continue has to be inserted. If the step did not hit a breakpoint, it paused instead at some previously unknown location. Then, an additional continue re-sychronizes the trace of the initial and follow-up run.

*Breakpoint sliding.* The last action transformation tests a behavior of debuggers known as *breakpoint sliding*. While users can set breakpoints at any line of a program, not every line corresponds to an actual operation at runtime. The simplest case of such lines are comments or empty lines, but also non-trivial programming constructs may correspond to no runtime instruction (e.g., types). As a usability feature, debuggers move the breakpoint to the next "appropriate" location, instead of simply ignoring it.

Figure 9 shows a simple example of this. It also introduces our notation for a requested breakpoint that was moved away (⬚) and the final, actual location of that breakpoint (▶). That is, in the example the debugging action was *add breakpoint at 1:1* and the debugger output was *breakpoint set at 2:1*. In the GUI, the debugger shows the actual location of the moved breakpoint, so we expect that this breakpoint behaves every bit the same as if the user would directly set a breakpoint at this final location. That is, in this example we could replace the original debugging action with *add breakpoint at 2:1*.

We can test this relation by transforming every *add breakpoint* action that results in a moved breakpoint (i.e., where the actual location returned by the debugger is not equal to the requested location), into an *add breakpoint* action where the location is directly at the actual location. This is described abstractly by Figure 10.

This transformation is also useful in the presence of iterative metamorphic testing (Section 3.3) in conjunction with program transformations. Consider again the example of Figure 9. If a later metamorphic program transformation inserts code in between line 1 and 2, the breakpoint would no longer be slided to the original statement. Instead, it could be moved inside the new code, thus no longer behaving as in the non-transformed program. To counter this unwanted interaction between breakpoint sliding and code changes, we implicitly apply this transformation to every slided breakpoint before applying any program transformation. Since all breakpoint sliding is eliminated after this transformation, it can no longer interact badly with subsequent program transformations.

#### 3.2.2 Program Transformations.

The second class of transformations focuses on modifying the source code of the program-to-debug. Debugging actions are only changed to keep them consistent with the modified program.

*Add dead code.* Here we transform the program by adding dead code at a randomly selected program location, similar to previous work on compiler testing [17]. Our inserted code is of the form

```
if (false) {
    variable = value;
}
```

where *variable* is randomly chosen from the scope where this code is added to (e.g., a function parameter) and *value* is a literal. Note that later program transformations can also increase the complexity of the dead code, e.g., a `false` literal might get replaced by an expression and no-operations or more dead code might be inserted inside the `if`'s body.

Because the debugging actions refer to program locations by lines and columns, this program transformation is additionally accompanied by an action transformation that moves the breakpoints after the insertion point by the number of inserted lines. Other than that, no changes to the actions are necessary. Because dead code by definition should have no influence on the execution, we still expect that continues and steps lead to the same pauses. During comparison, we simply need to apply a mapping from old lines to new lines for pause locations, due to the code insertion.

*Remove dead code.* Intuitively, this transformation is the inverse of the previous one. During the initial test run, we collect dynamic coverage information (the Chromium debugger supports this via `Profiler.takePreciseCoverage`). Then, we randomly select an AST node that is dead code as per this coverage information and remove it from the program, e.g., a whole function that was never called or block statements that were never executed such as in conditionals or loops.

As for the debugging actions, we only adapt all locations past the removal point and remove all breakpoints that were set in the now removed dead code. Since they could not have been hit previously, no pauses should appear and disappear in the follow-up test run.

```
function foo(p1,p2) {
➡ // p1, p2 are
   // in scope
}
foo();
```

**(a) Initial test run.**

```
function foo(p1,p2,fresh) {
➡ // now also expect
   // fresh == undefined
}
foo();
```

**(b) Follow-up test run.**

**Figure 11: Add function parameter transformation example.**

*Add spurious parameter to function declaration.* Because of JavaScript's relaxed semantics, functions can be called with less arguements than they are declared with. The formal parameters of the missing arguments are bound to `undefined`, but the call is still valid. In this program transformation we thus add a parameter to a random function declaration, but do not change any call site. Figure 11 gives an example. The inserted parameter gets a fresh name, so that it does not collide with existing identifiers.

As for transforming the actions accordingly, only breakpoints on the same line as the added parameter need to be moved by the number of inserted columns. Additionally, for the comparison between initial and follow-up run, inside the body of the modified function, we need to expect an additional variable with the freshly generated name and value `undefined`.

*Add no-op-like statement.* This transformation is similar to adding dead code, only that the code itself shall have no effect instead of being guarded by a false condition. An instance of such a statement that shall have no effect are self assignments. Currently, we always insert code of the form *variable* = *variable*; where *variable* is a random, non-`const` variable from the current scope.

Again, locations past the insertion point in debugging actions have to be changed in order to keep them consistent with the program. Similarly, the comparison accounts for the changed lines between the initial and follow-up run.

*Replace literal with expression.* This program transformation replaces integer and boolean literals by expressions. In total, it consists of six individual transformations: four transformations for integer literals and two for boolean literals. An integer literal $n$ is replaced by one of the following randomly selected expressions:

- Addition: ($n - 1$ + 1), e.g., replacing 12 by `(11+1)`.
- Subtraction: ($n + 1$ - 1), e.g., replacing 12 by `(13-1)`.
- Division: ($n$ / 1), e.g., replacing 12 by `(12/1)`.
- Multiplication: ($n$ * 1), e.g., replacing 12 by `(12*1)`.

For booleans, we replace a literal `true` by a tautological expression, namely $v$ == $v$, where $v$ is a randomly selected variable from the current scope. Special care must be taken because of IEEE754 not-a-number floating point values. The standard demands that a comparison with `NaN` is always false, even if compared against itself. For that reason, a correct replacement for `true` is (`isNaN`($v$) || $v$ == $v$) and (!`isNaN`($v$) && $v$ != $v$) for `false`.

Locations in the debugging actions and traces are only changed by the inserted number of columns in the affected line.

### 3.3 Iterative Testing

In its simplest form, metamorphic testing consists only of an initial test run and a follow-up test run, where the follow-up inputs are generated by a transformation from the initial ones. However, in our case there is no difference between initial inputs and follow-up inputs. Both consist of a program-to-debug and debugging actions. We can thus apply the transformations a second time on the follow-up test run to obtain yet another test input. On these new inputs, we can apply the program and action transformations yet again, continuing indefinitely, if necessary. Since the metamorphic relation should hold between each input and its individual follow-up, each iteration creates a new metamorphic test case.

We use this *iterative* metamorphic testing to obtain multiple follow-up test runs from a single initial test run. There are two benefits of iterative testing for debuggers. First, since we take the programs-to-debug from a fixed test set, our initial programs are limited in size. By applying multiple transformations on them, we can perform more metamorphic tests per input program than if we would only apply a single transformation. Second, iterative testing is interesting because of the complex possible combinations of program and action transformations. Examples of interesting combinations are a program transformation that inserts dead code, followed by a transformation that modifies the inserted code, or combinations of action transformations that first replace continues by steps and then insert additional breakpoints.

In our approach, we first randomly select one of the action or program transformations from the previous section. If the comparison on the debugging traces is successful, iterative testing continues by transforming the inputs of the follow-up test case once more with a randomly selected transformation. This process continues until either a metamorphic test case fails (and a potential bug is found) or until we reach a maximum number of iteration *rounds*. Continuing after a failed metamorphic test case does not make sense, as we have already reached an inconsistent state.

## 4 IMPLEMENTATION

We implement our approach as a fully automated tool that tests the Chromium JavaScript debugger. The core of the implementation is a Node.js application that controls the debugger via its remote debugging API. This API allows us to trigger debugger actions and to obtain the debugging trace. Given an initial program and debugging actions, the implementation performs up to five input transformations, and it stops as soon as any behavior is observed that is not compatible with the metamorphic output relation.

While our implementation targets the Chromium JavaScript debugger, the underlying concept of metamorphic testing of debuggers is not tied to any specific language or debugger. E.g., the action transformations assume a general model of debuggers, and the program transformations are equally applicable to other languages.

## 5 EVALUATION

We evaluate the effectiveness and efficiency of the approach by applying our implementation to a large test set of JavaScript programs. The evaluation focuses on the following research questions:

- How effective is the approach at detecting bugs? (Section 5.2)
- How important is the interactive nature of our metamorphic testing approach? (Section 5.3)
- Does iterative metamorphic testing reveal problems missed with a single iteration of the approach? (Section 5.4)
- What is the runtime of our metamorphic testing? (Section 5.5)

## 5.1 Experimental Setup

We apply our tool to a total of 47,342 JavaScript files. The vast majority of them is from the test262 ECMAScript test suite[3], which contains thousands of short code snippets that test individual features of JavaScript. We create multiple variants of the files in the test suite by running them in strict and non-strict mode, and by concatenating multiple files into larger test cases. In addition, we also use files from the Sunspider and Octane benchmark suites[4] and a set of JavaScript code puzzles from a university course. For all files, we automatically replace non-deterministic API calls, e.g., to `Date.now()`, with deterministic code, to make our results reproducible and to avoid spurious warnings.

Given the total of 47,342 JavaScript files, we create an initial sequence of debugging actions with DBDB, an existing approach for automated debugger testing [19]. Each of these initial test cases is executed twice, each time creating and executing a metamorphic test, which yields a total of 94,684 test executions. We ignore a small subset of these executions because they failed or reported incorrect results due to bugs in our implementation.

We apply our tool to find bugs in the JavaScript debugger of the widely used Chromium web browser. Unless otherwise noted in our bug reports, all experiments use version 70.0.3538.102 of Chromium. The experiments are conduced on two standard computers, an AMD Phenom II 945 with 3GHz CPU and 8GB of RAM, and an Intel i7 CPU with 4.6GHz and 16GB of RAM. On these computers, we run 16 Debian-based virtual machines to parallelize our experiments.

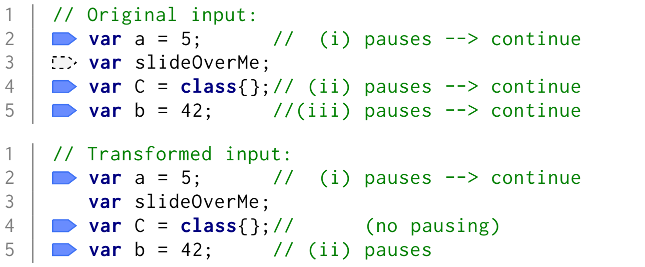
## 5.2 Effectiveness at Detecting Bugs

While generating and executing 94,684 metamorphic debugger tests, our tool reports a total of 59 warnings. For each of them, the traces obtained from the original and the transformed inputs are not in the expected output relation. We manually inspected and classified them into true positives and false positives. A true positive is a violation of the equivalence relation that indeed points to a bug in the tested debugger. We find 30 of the 59 warnings to be true positives. The remaining 29 warnings are false positives, i.e., the result of violating an assumption we make when designing the metamorphic relations. Typically, such violations are due to corner-cases of the JavaScript language. Sections 5.2.1 and 5.2.2 give examples of both kinds of warnings. Overall, we believe a true positive rate of 51% is acceptable for automatically detecting bugs in a tool (debuggers) that is both difficult to test and important for many developers. Refining the metamorphic relations to avoid specific corner-cases of JavaScript could further increase the true positive rate.

Through manual inspection, we identify a set of unique root causes for the reported warnings. In the process of our evaluation, we reported nine unique root causes as bugs to the Chromium developers. Table 3 lists the bug reports, along with their status as of June 1, 2019. Eight reports are currently "Assigned" to a developer, which, as per the Chromium bug reporting guidelines[5], means a developer has inspected our report and confirmed it as a bug. One of the reported bugs is marked as release-blocking. Another one of the reported bugs, which is about changed program behavior caused by debugging, has been marked as "Won't fix". Overall, our experience reporting these bugs shows that our approach detects relevant problems in complex, real-world software.

**Table 3: Bugs reported to the Chromium developers.**

| Issue ID | Description | Status |
|---|---|---|
| 862978 | Cannot set breakpoint | Assigned |
| 889481 | Debugger does not pause | Assigned |
| 892622 | Debugger does not pause | Assigned,<br>release-blocking |
| 892653 | Pauses at location without breakpoint | Assigned |
| 901811 | Missing variable in scope | Assigned |
| 901814 | Step-in does not enter function | Assigned |
| 901816 | Missing variable in scope | Assigned |
| 901819 | Debugger does not pause | Assigned |
| 908054 | Debugging changes program behavior | Won't fix |



```
1    // Original input:
2  ▶ var a = 5;      //  (i) pauses --> continue
3  ▷ var slideOverMe;
4  ▶ var C = class{};// (ii) pauses --> continue
5  ▶ var b = 42;     //(iii) pauses --> continue

1    // Transformed input:
2  ▶ var a = 5;      //  (i) pauses --> continue
3    var slideOverMe;
4  ▶ var C = class{};//      (no pausing)
5  ▶ var b = 42;     // (ii) pauses
```

**Figure 12: Bug that causes the debugger to not pause at a breakpoint.**

### 5.2.1 Examples of Detected Bugs.

*Debugger does not pause at breakpoint.* The following bug has been reported as issue 889481. Figure 12 shows two JavaScript code snippets along with their breakpoints. In the code on the top, our tool sets three breakpoints at lines 2, 3, and 5, respectively. Because the second breakpoint is on a variable declaration only, the debugger slides it to line 4. When running the code, the debugger pauses at all three breakpoints and moves on to the next breakpoint when triggering the `continue` action. The code at the bottom is the same, but the debugging actions differ. Instead of setting a breakpoint at line 3, which anyway slides to line 4, our tool now sets the breakpoint directly at line 4. This supposedly harmless difference causes the debugger to miss the breakpoint during the execution. After pausing at line 2 and continuing the execution, the debugger skips line 4 and only pauses again at line 5.

Such misbehavior is very misleading for developers because it seems that a statement is not executed even though the underlying JavaScript engine executes it. Our approach detects this bug by applying two transformations in a row. In the first transformation, it adds a breakpoint and a continue at line 3. In the second transformation, it eliminates breakpoint sliding by setting the breakpoint directly at line 4 instead of line 3. The two code examples in Figure 12 shows the code before and after the second transformation.

[3] https://github.com/tc39/test262/
[4] https://webkit.org/perf/sunspider/sunspider.html, https://chromium.github.io/octane/
[5] https://www.chromium.org/for-testers/bug-reporting-guidelines

```
1      // Original input:
2      function *g() {
3        //  (i) pauses --> continue
4  ▶     var foo = [ x , ...{}[ yield ] ] = [];
5      } // (ii) pauses
6      var iter = g();
7      iter.next();
8  ▶   iter.return(); // (iii) pauses
```

```
1      // Transformed input:
2      function *g() {
3        //  (i) pauses --> step-out
4  ▶     var foo = [ x , ...{}[ yield ] ] = [];
5      } // (no pausing)
6      var iter = g();
7      iter.next();
8  ▶   iter.return(); // (ii) pauses
```

**Figure 13: Bug that causes the debugger to pause without any breakpoint.**

```
1  // Original input:
2  function * t({x: ▶ y}) { // pauses, y is in scope
3    var a = function() {
4    }
5  }
6  t({x: 1});
```

```
1  // Transformed input:
2  function * t({x: ▶ y}) { // pauses, y is missing
3    var a = function() {
4      if (false) { // dead code
5        y = 5;
6      }
7    }
8  }
9  t({x: 1});
```

**Figure 14: Bug that causes the debugger to show an incorrect program state.**

*Debugger pauses at location without breakpoint.* Figure 13 shows a bug reported as issue 892653. The code example uses a combination of generator functions, marked with `*`, and the spread syntax `...`, which expands an object expression. The code at the top stops at the breakpoint in line 4, after which a `continue` action is triggered. Surprisingly, the debugger pauses again at line 5, even though this line does not have any breakpoint. Since continues are supposed to pause only at locations explicitly chosen by the developer, this behavior is clearly incorrect.

Our tool transforms the code at the top by replacing the `continue` action at line 4 with a `step-out` action. This kind of change should not remove any previously existing pause locations, but now the debugger does not pause at line 5 anymore, but instead moves on the next breakpoint at line 8. Interestingly, the input before, not after, applying the transformation, exposes the unexpected behavior in this example. However, applying the transformation is crucial to identify the behavior as unexpected, because our tool reports this warning only because the traces produced by the two executions are not in the expected output relation (namely equal pauses).

*Missing variable in scope.* The third example, reported as issue 901816, shows a bug that causes the debugger to show an incorrect program state to the developer. The code in Figure 14 uses the object destructuring syntax to unpack an object passed to function `t`. When the function gets called, its local variable `y` gets assigned the value of property `x` of the object that is passed as an argument. The testing tool sets a breakpoint at line 2 just before this assignment gets executed. At this point in time, the local variable `y` exists but still has value `undefined`. In the example at the top, the debugger correctly shows that `y` exists in the local scope of the function when reaching the breakpoint.

Our approach transforms the code at the top by inserting dead code into the nested function, as shown in the bottom of Figure 14. Even though adding dead code should not change any debugger behavior, variable `y` is now missing from the scope shown to the developer when hitting the breakpoint at line 2. Showing an incorrect program state can be very confusing for developers, because they usually use a debugger to understand the program state.

### 5.2.2 *Examples of False Positives.*

The main cause of false positives is that some assumption we make when designing the metamorphic relations does not always hold in practice, e.g., because the relation does not consider a corner-case of the JavaScript language.

One example is a warning reported when applying the "replace boolean literal with expression" transformation. In this case, the transformation replaces the literal `true` with the expression `(!isNan(obj)) && (obj !== obj)`. While the result of this expression is indeed always `true`, calling the built-in `isNaN(obj)` function has the side-effect of calling the `toString` method of `obj` as part of an implicit type coercion [23]. By default, `toString` is defined through a built-in function. In the specific code example that produces the false positive, though, the function is overridden with a user-defined function, causing the control flow to enter this function when `isNaN` gets called. This change of control flow causes the debugger to pause at new location inside the user-defined `toString`, which our metamorphic relation does not expect.

Another example is a warning caused by a program that inspects the source code of a function using the built-in `Function.prototype.toString` API and compares it to another string. When our tool applies a program transformation that changes the code of this function, this comparison is no longer true and the state observed by the debugger changes again. Because our metamorphic program transformations do not expect such a change, a warning is reported that turns out to be a false positive.

## 5.3 Influence of Interactive Testing

One of the technical novelties of our approach compared to traditional metamorphic testing is to interact with the program under test to determine the metamorphic input transformation and to determine the expected output relation. To assess the influence of this feature, we measure how many of the reported 59 warnings involve at least one interactive transformation, i.e., either "Add breakpoint and continue" or "Replace continue by step" (see Table 1). For example, finding the bug in Figure 13 involves replacing a `continue` action with a `step` action, which is an interactive transformation. In total, 29 of the 59 reported warnings involve at least one interactive transformation, showing that the interactive nature of our metamorphic testing approach is worthwhile.

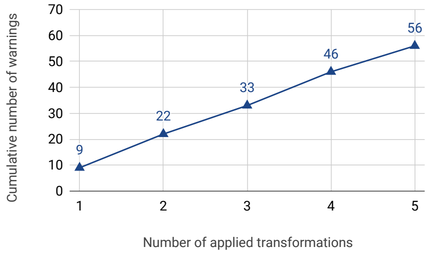


**Figure 15: Cumulative number of warnings reported by our tool after $N$ iteratively applied transformations.**

## 5.4 Influence of Iterative Testing

Our testing approach is iterative, i.e., inputs produced through a metamorphic transformation are used as the starting point for other metamorphic transformations. We see that iteratively applying transformations in our approach is useful because four of the nine reported bugs (i.e., true positives) were found after two iterations and one more true positive was found only after three iterations. Additionally, we measure how many warnings are reported after a given number of applied transformations. The results in Figure 15 show that additional rounds of applying transformations reveal additional warnings. Since the curve is increasing without any sign of saturation, it suggests that even more transformations than the the upper limit of five, which we currently use, could be useful.

## 5.5 Efficiency

Finally, we evaluate how efficient our tool is in terms of runtime. Our current implementation takes around 24 hours to generate and execute the 94,684 tests cases our evaluation is based on. This time is obtained by running 16 parallel instances in parallel on two computers. We believe that this amount of required computing resources is acceptable for a fully automated tool that detects otherwise missed bugs in a crucial piece of software.

## 6 RELATED WORK

*Metamorphic Testing.* Introduced by Chen et al. [7], metamorphic testing has been applied to various kinds of software, including implementations of partial differential equations [8], service-oriented applications [5], and machine learning models [27, 28]. Our work contributes the first metamorphic testing approach for debuggers, and it introduces interactive metamorphic testing, which differs from the traditional approach by determining both the metamorphic relation and transformation during the execution of the program under test. A study of the selection of useful metamorphic relations [9] distinguishes blackbox and whitebox approaches. According to this classification, our work is blackbox, as we design the relations without considering the implementation of the tested debugger. Liu et al. [20] study the effectiveness of metamorphic testing. A comprehensive summary of the above and other existing metamorphic testing work is available in a recent survey [25].

*Debugger Testing.* Despite the importance of debuggers as widely used developer tools, automated testing of debuggers has only recently started to receive attention. Lehmann and Pradel [19] propose a feedback-directed test generator for debuggers, called DBDB, and apply it via differential testing [22]. Our work builds on DBDB's approach for obtaining initial test inputs. The main advantage over DBDB is that we do not require two supposedly equivalent debugger implementations, but only a single debugger that gets tested against itself. This difference is particularly important because debuggers are notoriously underspecified, i.e., their expected behavior is only informally known and different debuggers often provide slightly different behavior, despite their common interface [19].

*Automated Testing of Compilers.* The problem of testing compilers has received significant attention. Several approaches mention [31] and implement [17, 18, 26] different forms of metamorphic testing. Other approaches include random program generation [29] and enumerating all programs for a given code template [30]. Chen et al. empirically study different compiler testing techniques [6]. To ease the task of understanding warnings from compiler testing, ranking of warnings [10] and test program reduction, either for a single language [24] or arbitrary languages [14] has been proposed.

*Automated Testing of Other Developer Tools.* Beyond compilers, other developer tools and program analyses are subject to automated testing, including work on differential testing of refactoring engines [11], differential testing of symbolic execution engines [16], and fuzzing to test implementations of abstract interpretation domains [3]. Cadar and Donaldson argue that more effort should be spend on testing or analyzing program analyzers[4], which our work is an example of.

*Testing of Interactive Applications.* Due to the interactive nature of debuggers, the problem of testing them resembles UI-level testing. Existing UI-level test generators, e.g., for web applications [1, 13] or Android [15, 21] can, however, not be easily adapted to debugger testing. The reason is that UI-level events, such as stepping through code, are strongly tied to the debugged code – a kind of input not consider by existing UI-level testing techniques.

## 7 CONCLUSION

This paper presents the first approach for metamorphic testing of debuggers. Given a program to debug and a sequence of debugging actions, the approach transforms both inputs in such a way that the debugging behavior is expected to change only in particular ways. To address the dynamic nature of debuggers, we introduce the concept of interactive metamorphic testing, which differs from traditional metamorphic testing by determining both the transformation and the expected output relation during the execution of the program under test. We show that our approach is effective and efficient at testing a widely used, real-world debugger, where we found several previously unknown bugs. Our work contributes a novel tool for testing debuggers to the stream of work on improving the quality of developer tools. Beyond debuggers, we envision interactive metamorphic testing to be applied to other developer tools, e.g., interactive IDEs.

## ACKNOWLEDGMENTS

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

# REFERENCES

[1] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. 2011. A framework for automated testing of JavaScript web applications. In *ICSE*. 571–580.

[2] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[3] Alexandra Bugariu, Valentin WÃijstholz, Maria Christakis, and Peter MÃijller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*.

[4] Cristian Cadar and Alastair F Donaldson. 2016. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 765–768.

[5] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. 2007. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research (IJWSR)* 4, 2 (2007), 61–81.

[6] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An Empirical Comparison of Compiler Testing Techniques. In *ICSE*.

[7] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases*. Technical Report. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong âĂę.

[8] Tsong Yueh Chen, Jianqiang Feng, and T. H. Tse. 2002. Metamorphic Testing of Programs on Partial Differential Equations: A Case Study. In *26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, 26-29 August 2002, Oxford, England, Proceedings*. 327–333. https://doi.org/10.1109/CMPSAC.2002.1045022

[9] Tsong Yueh Chen, DH Huang, TH Tse, and Zhi Quan Zhou. 2004. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*. Polytechnic University of Madrid, 569–583.

[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Conference on Programming Language Design and Implementation (PLDI)*. 197–208.

[11] 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

[12] 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

[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] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-Structured Test Inputs. In *ASE*.

[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*. ACM, 77–83.

[16] 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.

[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 216–226. https://doi.org/10.1145/2594291.2594334

[18] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, 386–399.

[19] Daniel Lehmann and Michael Pradel. 2018. Feedback-directed Differential Testing of Interactive Debuggers. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 610–620. https://doi.org/10.1145/3236024.3236037

[20] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering* 40, 1 (2014), 4–22.

[21] 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*. ACM, 224–234.

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

[23] Michael Pradel and Koushik Sen. 2015. The Good, the Bad, and the Ugly: An Empirical Study of Implicit Type Conversions in JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*.

[24] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 335–346.

[25] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[26] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. 270–279. https://doi.org/10.1109/APSEC.2010.39

[27] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 303–314. https://doi.org/10.1145/3180155.3180220

[28] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558. https://doi.org/10.1016/j.jss.2010.11.920

[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.

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

[31] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. 2004. Metamorphic testing and its applications. In *Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004)*. Software Engineers Association Xian, China, 346–351.