Complete Shadow Symbolic Execution with Java PathFinder

Yannic Noller
Humboldt-Universität zu Berlin
yannic.noller@informatik.hu-berlin.de

Hoang Lam Nguyen
Humboldt-Universität zu Berlin
nguyenhx@informatik.hu-berlin.de

Minxing Tang
Humboldt-Universität zu Berlin
tangminx@informatik.hu-berlin.de

Timo Kehrer
Humboldt-Universität zu Berlin
timo.kehrer@informatik.hu-berlin.de

Lars Grunske
Humboldt-Universität zu Berlin
grunske@informatik.hu-berlin.de

ABSTRACT

Regression testing ensures the correctness of the software during its evolution, with special attention on the absence of unintended side-effects that might be introduced by changes. However, the manual creation of regression test cases, which expose divergent behavior, needs a lot of effort. A solution is the idea of *shadow symbolic execution*, which takes a unified version of the old and the new programs and performs symbolic execution guided by concrete values to explore the changed behavior.
In this work, we adapt the idea of *shadow symbolic execution* (SSE) and combine complete/standard symbolic execution with the idea of four-way forking to expose diverging behavior. Therefore, our approach attempts to comprehensively test the new behaviors introduced by a change. We implemented our approach in the tool SHADOW$_{\text{JPF+}}$, which performs complete shadow symbolic execution on Java bytecode. It is an extension of the tool SHADOW$_{\text{JPF}}$, which is based on Symbolic PathFinder. We applied our tool on 79 examples, for which it was able to reveal more diverging behaviors than common shadow symbolic execution. Additionally, the approach has been applied on a real-world patch for the Joda-Time library, for which it successfully generated test cases that expose a regression error.

Keywords

Regression Testing; Symbolic Execution; Symbolic PathFinder

1. INTRODUCTION

Real-world software *evolves* in order to catch up with a continuously changing environment, which makes it necessary to fix incorrect behavior or introduce new functionality. These changes are usually denoted as *patches*. However, patches bare the risk to introduce new errors [6, 16]. To mitigate this risk, *regression testing* is used to explore the changed behavior, which was introduced by the applied patches. *Regression testing* cannot verify the absence of errors, but it can provide confidence that the patches follow their intention. Several regression testing techniques [7, 5] select and run a subset of the test cases from the program's existing test suite or automatically generate test cases with high coverage of the changed code [11]. However, full statement or branch coverage achieved by the set of regression test cases may not lead to the desired divergence revealing test inputs.

Recently, Palikareva et al. [13] have introduced a dynamic symbolic execution-based technique, which they refer to as *shadow symbolic execution* (SSE). Instead of generating a high-coverage test suite, their technique is designed to generate test inputs that reveal new program behaviors introduced by a patch. SSE executes both the old (buggy) and new (patched) versions in the same symbolic execution instance, with the old version *shadowing* the new one. Therefore, it is necessary to *merge* both programs into a change-annotated, unified version. Based on such a unified version and driven by the idea of four-way forking, the technique detects divergences along the execution path of an input that exercises the patch. However, if an execution path does not immediately expose a divergence, it may no longer be explored. While this approach significantly reduces the program search space by pruning a large number of execution paths, it might miss divergences that could expose regression errors.

In this work we present the combination of complete symbolic execution with the idea of four-way forking for the generation of regression test cases. This approach tries to test the new program version more comprehensively by exhaustively exploring the execution tree in order to detect divergences that expose new program behavior. We implemented our approach as an extension of the tool Symbolic PathFinder (SPF) [14], called SHADOW$_{\text{JPF+}}$, which performs SSE on Java bytecode and is able to effectively generate divergent test cases on a unit level. Therefore, we extended our SSE implementation for Java bytecode, called SHADOW$_{\text{JPF}}$ [12]. The main contributions of this work are:

1. The combination of complete symbolic execution with the idea of four-way forking, as a technique to generate regression tests that expose changed program behavior.
2. The tool SHADOW$_{\text{JPF+}}$ as an extension of the SHADOW$_{\text{JPF}}$.
3. The application of SHADOW$_{\text{JPF+}}$ on various examples, including a patch for the Joda-Time library in order to evaluate its test case generation capabilities. Furthermore, our approach is compared to SHADOW$_{\text{JPF}}$ to assess the effectiveness of our improved search exploration strategy.

2. BACKGROUND AND MOTIVATION

2.1 Symbolic Execution

The key idea behind symbolic execution [9, 3] is to run a program with *symbolic* input values instead of concrete ones. The values of the program variables as well as the output values are expressed as *symbolic expressions* over the symbolic input. As a result, the behavior of a program can be described as a symbolic execution tree, where the nodes represent (symbolic) program states and the edges represent the program transitions. A *symbolic state* S is defined as tuple (σ, PC, IP), where σ is a mapping from variables to symbolic expressions, PC is the current *path condition* and IP denotes the *instruction pointer* to the next instruction. A path

condition is a conjunction of constraints over the symbolic inputs, which must be satisfied in order to follow a particular execution path. The mapping σ needs to be updated at each assignment accordingly. The path condition gets updated at every conditional statement and gets checked with a constraint solver for its satisfiability. Symbolic execution has been implemented by several tools, which support different programming languages and target different applications [2]. For our evaluation we focus on Symbolic PathFinder (SPF) [14], which combines symbolic execution with model checking, also called *generalized symbolic execution*, on Java bytecode, and which has been applied in academia as well as in the industry. SPF is based on the Java PathFinder (JPF) [15] and extends and overwrites certain bytecode instructions in order to enable symbolic execution of Java bytecode.

2.2 Shadow Symbolic Execution

Palikareva et al. [13] introduced *shadow symbolic execution (SSE)* as symbolic execution-based technique that aims to reveal behavioral differences between two program versions, and hence, find newly introduced regression bugs. Specifically, their techniques searches for divergences between a buggy version (also denoted as *old* version) and a subsequent, patched version (also denoted as *new* version) that fixes the previous known bug. By using initial test inputs from an existing test suite they dynamically execute both versions in the same symbolic execution instance. Therefore, [13] manually transforms the two versions into one `change()`-annotated program by applying simple annotation rules. Each `change()` annotation represents a function call `change(oldExpression, newExpression)`, where the arguments show the old and the new expressions respectively.

As an example Listing 1 shows the function `foo`. The modification in line 4 can be unified as:

`y=change(-x,x*x).`

The addition in line 8 can be unified as:

`y=change(y,y+1),`

in which an addition can be simulated as a modification from the dummy assignment `y=y` to `y=y+1`. The presented changes in Listing 1 represent a patch for the method `foo()` that fixes the assertion error (line 13) for x=-1, but it introduces a regression bug with a new assertion error for x=0. The desired result would be two test inputs that trigger changed behavior: one for the fixed path, and one for the regression bug. Figure 1 shows the complete four-way forking symbolic execution tree for the unified version of the program in Listing 1. The approach by [13] is driven by the concrete inputs to limit the search space, and hence, it will explore only a subset of the shown symbolic states (depending on the used test inputs).

```
1   int foo(int x){
2     int y;
3     if(x < 0){
4-      y = -x;
4+      y = x * x;
5     } else{
6       y = 2 * x;
7     }
8+    y = y+1;
9     if(y > 1){
10        return 0;
11    } else {
12        if(y == 1)
13            assert(false);
14    }
15    return 1;
16  }
```

Listing 1: Motivating Example.

They perform a dynamic symbolic execution on the unified program in two steps: (1) detect possible divergences along a concrete path (denoted as *concolic* phase), and (2) from the discovered divergence points it performs a bounded symbolic execution (BSE) on the new program version only. During the concolic phase, branching instructions, like if-statements, are handled in special way: instead of forking the execution in two executions for the *true* and the *false* branch, SSE applies the notion of a *four-way* forking to investigate all four combinations of *true* and *false* branches for both program versions. As long as the concrete executions take the same decisions at the branching instructions, SSE follows the so-called $same_{True/False}$ path, however, at each branching instruction SSE also checks the feasibility of the $diff_x$ paths as part of the four-way forking. As long there is no concrete divergence, SSE continues until the end of the program. As soon as SSE hits an addition or a removal of straightline code, which is represented by the change annotations `if(change(false, true))` and `if(change(true, false))`, it immediately triggers a divergence point. This leads to an over-approximation of the *diff* paths because the added / deleted code may not necessarily lead to an actual divergence. Their tool SHADOW is implemented on top of the KLEE symbolic execution engine [1], in the remainder of the paper we will refer to it as $\text{SHADOW}_{\text{KLEE}}$.

2.3 Need for further Research

In particular, there are two limitations in the exploration strategy of [13] that we want to address.

Deeper divergences might be missed in the BSE phase: The BSE phase of [13] aims to find additional test inputs that trigger divergent behavior by exploring the execution tree of the new version starting from each divergence point found in the concolic phase. This implies that each BSE run inherits the path condition prefix from the initial input from the start to the divergence point. For example in Listing 1, for the input x=-1, which fully covers the changed statements, [13] would only generate the test case representing the fixed path and it would miss the path with the new introduced assertion error. The reason for that is that the collected path condition prefix limits the exploration space of BSE. The collected path condition up to line 9 is: $X < 0$. In order to get to the assertion error in line 13 with the new version, BSE needs to follow the *false* branch at line 9 with the condition: $(X^2 + 1 \leq 1)$, which is only possible for $X = 0$, but this contradicts with the current path condition $(X < 0)$ (cf. node 2 in Figure 1). With this precondition [13] cannot find the new introduced assertion error.

The initial input has to cover potential divergence points: Since the concolic phase of [13] only searches for divergences along the path of a concrete input that exercises the patch, divergences along alternative paths will be missed if there is no satisfiable divergence at the branching point. This means that if the concrete input does cover the changed statements, but does not cover the potential divergence point, then [13] is not able to find the *diff* path. In the program in Listing 2 a potential divergence can happen only in line 4 because it is the only condition that depends on a changed variable. The condition in line 3 is never a divergence point because the variables in the condition are not affected by the patch. To discover the divergence, the initial input has to actually reach line 4, meaning that any initial input `x` and `y` with `x+y` $\neq 5$ would miss the divergence because concrete execution would follow only the *false* branch of the conditional statement in line 3 and discard the other paths.

```
1  int bar(int x, int y){
2    int z = change(x,y);
3    if((x+y) == 5){
4      if(z == -100)
5        assert(false);
6    }
7    return 0;
8  }
```

Listing 2: Limitation example.

3. APPROACH

In order to tackle the limitations discussed in Section 2.3, we follow a conservative approach that executes the change-annotated

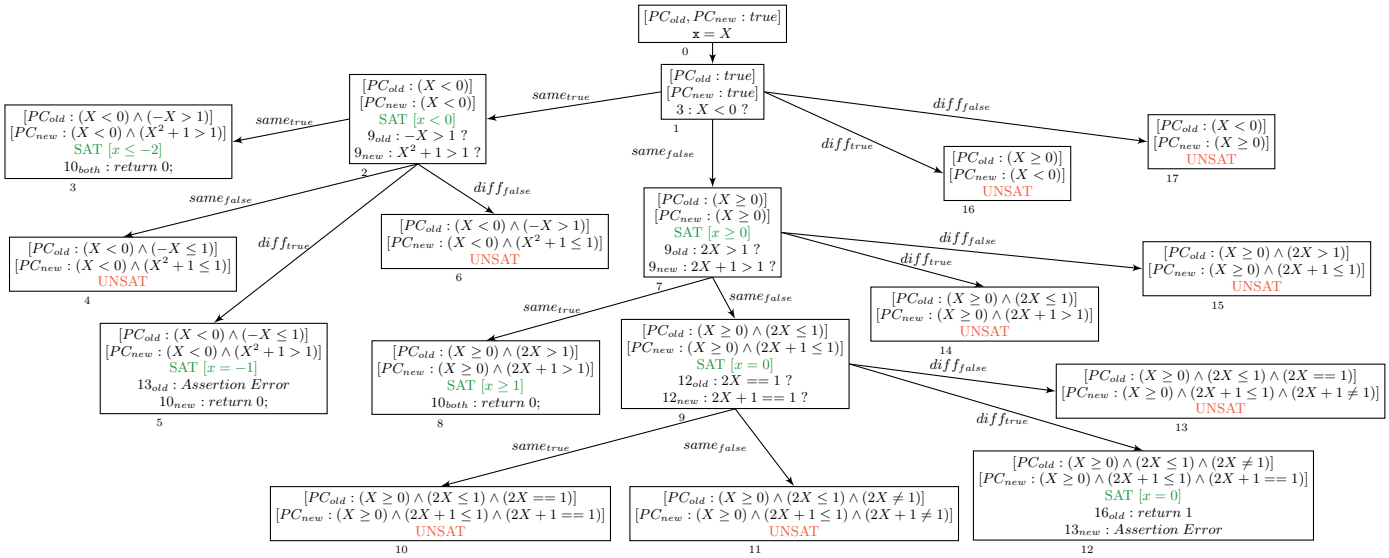

Figure 1: Complete four-way forking symbolic execution tree for the combined execution of the old and the new versions of the program in Listing 1. Each node represents a state in the symbolic search space, where each state holds the combined information of the old and the new symbolic executions.

program with bounded symbolic execution and four-way forking in a depth first manner, while detecting divergences on the fly. That is, instead of searching for divergences only along the path of a concrete input, the execution is forked into four different paths at *each* branching point ($same_{false}$, $same_{true}$, $diff_{false}$ and $diff_{true}$). The symbolic execution tree in Figure 1 represents exactly the states being explored by our approach. As soon as the executions of the two program versions diverge, i.e., we start with the exploration of a $diff_x$ path, the execution of subsequent conditional statements forks execution into two paths based on the execution of the new version *only*. Nevertheless, since we also explore paths where both program versions do not yet diverge ($same_{false}$ and $same_{true}$ paths), it is possible to detect divergences at every depth of the execution path. As soon as the execution terminates or a user-specified depth is reached, we check if the current path exercised divergent program behavior (i.e., if the path is a *diff* path) and generate a concrete test input in that case.

Adding and removing straightline code: In [13], the special case of adding and removing straightline code is handled by using the annotations like `if(change(false,true))`, which immediately trigger a *diff* path and terminate SSE, even though both versions might still exhibit the same branching behavior. Our approach continues SSE in this case, but *only* updates the symbolic information of the respective version. This is done by using the separately handled annotation `if(execute(version)){...}`, where `version` denotes the program version of the enclosed code (`OLD` or `NEW`). Any conditional statement inside such a code block forks execution into two separate paths based on the symbolic values of the specified version *only*. A divergence inside an `if(execute(version))` block is only possible if it contains a return statement, i.e., if the particular program version returns early. Therefore, our handling of added/removed code blocks is more precise as it does not lead to spurious divergences.

Implementation: We implemented our approach in SHADOW$_{\text{JPF+}}$ by augmenting our preliminary work on SHADOW$_{\text{JPF}}$ [12] with our new ideas on the search heuristic. In particular, we extended the implementation of the conditional bytecode instructions (e.g., `IFEQ`, `IF_ICMPGT`) such that the execution is forked into four paths (or two paths, while exploring a *diff* path). The arithmetic bytecode instructions (e.g., `IADD`, `IINC`) have also been updated to specifically support the newly introduced `if(execute(version))` annotation. The execution of SHADOW$_{\text{JPF+}}$ on the program presented in Listing 1 results in the two test inputs x=-1 and x=0 (cf. Table 3) for the expected *diff* paths, which fits in with the four-way forking symbolic execution tree in Figure 1.

Directed Exploration: Our basic approach will explore *unchanged* paths, i.e. paths without any change-annotation, until the very end, where it will eventually prune them because they do not represent a *diff* path. This can slow down the symbolic exploration, which should only explore paths that can truly reach a changed statement. Therefore, we propose an optimization, which combines complete shadow symbolic execution with directed symbolic execution. We leverage the existing work on control-flow guided symbolic exploration strategies [10] to limit our exploration to paths that can reach a changed statement only. We added this directed exploration as an optional extension to our tool SHADOW$_{\text{JPF+}}$, since the computation overhead for the ICFG might not be worthwhile for some smaller applications.

4. EVALUATION

Our implementation and all evaluation artifacts are available on GitHub: `https://github.com/hub-se/jpf-shadow-plus`.
We evaluated our approach with the following research questions:
RQ.1: Can SHADOW$_{\text{JPF+}}$ reveal more divergent behaviors than SHADOW$_{\text{JPF}}$?
RQ.2: How does SHADOW$_{\text{JPF+}}$ compare to SHADOW$_{\text{JPF}}$ in terms of performance?
RQ.3: Can SHADOW$_{\text{JPF+}}$ expose real-world regression bugs?

Subjects: We selected the following software artifacts as our experimental subjects from the official SPF repository[1] (with the corresponding LOC): Rational.abs (30), Rational.gcd (40), Rational.simplify (51), WBS.update (234) and WBS.launch (242) and generated in total 79 mutants with the Major mutation framework [8] (similar as [12]) with the following change types: Relational Operator Replacement (ROR), Operator Replacement Unary (ORU), Arithmetic Operator Replacement (AOR) and Statement Deletion (STD). Since Major only generated single mutants

[1] `https://github.com/SymbolicPathFinder/jpf-symbc`

per class, we manually combined a randomly chosen subset of them to get complex mutants with multiple changes per class. Additionally, we inspected several open source projects on GitHub to find real regression bugs. In the Joda-Time library we found the issue #328[2], which fixes a regression bug that was introduced with the fix for the issue #190[3].

Subject	Type	Time [s]		# States		# Paths (diff)	
		SJ	*SJ+*	*SJ*	*SJ+*	*SJ*	*SJ+*
Rational.abs_1	ROR	<1	<1	21	32	1	1
Rational.abs_2	ROR	<1	<1	21	32	1	1
Rational.abs_3	ROR	<1	<1	13	20	1	1
Rational.abs_4	ORU	<1	<1	5	6	0	0
Rational.abs_5	ORU	<1	<1	5	6	0	0
Rational.gcd_1	ROR	<1	<1	42	220	0	0
Rational.gcd_2	ROR	<1	<1	23	48	2	4
Rational.gcd_3	ROR	<1	<1	40	234	3	3
Rational.gcd_4	STD	<1	<1	43	223	3	3
Rational.gcd_5	ROR	<1	<1	27	174	1	2
Rational.gcd_6	ROR	<1	<1	27	610	1	2
Rational.gcd_7	ROR	<1	<1	87	692	1	16
Rational.gcd_8	STD	inf	inf	-	-	-	-
Rational.gcd_9	ROR	<1	<1	45	434	0	0
Rational.gcd_10	ROR	<1	<1	57	626	3	48
Rational.gcd_11	ROR	<1	<1	15	42	1	2
Rational.gcd_12	ROR	<1	<1	104	308	3	6
Rational.gcd_13	ROR	<1	<1	104	642	3	14
Rational.gcd_14	ROR	<1	<1	43	236	1	6
Rational.gcd_15	AOR	<1	<1	43	178	4	10
Rational.gcd_16	AOR	<1	<1	39	170	4	10
Rational.gcd_17	AOR	<1	1	60	342	8	36
Rational.gcd_18	STD	<1	<1	37	166	2	6
Rational.gcd_19	AOR	<1	4	49	198	5	18
Rational.gcd_20	AOR	<1	<1	49	198	5	18
Rational.gcd_21	AOR	1	94	83	386	9	34
Rational.gcd_22	STD	<1	<1	49	198	5	18
Rational.simplify_1	ROR	<1	<1	55	284	4	6
Rational.simplify_2	ROR	<1	<1	63	370	3	3
Rational.simplify_3	ROR	<1	<1	71	252	4	6
Rational.simplify_4	ORU	<1	<1	28	280	2	8
Rational.simplify_5	ROR	<1	<1	42	364	0	1
Rational.simplify_6	ROR	<1	<1	31	96	3	7
Rational.simplify_7	ROR	<1	<1	63	366	4	4
Rational.simplify_8	STD	<1	<1	19	355	1	4
Rational.simplify_9	ROR	<1	<1	31	222	1	3
Rational.simplify_10	ROR	<1	<1	73	770	1	3
Rational.simplify_11	ROR	<1	<1	67	588	1	17
Rational.simplify_12	STD	inf	inf	-	-	-	-
Rational.simplify_13	ROR	<1	1	45	578	0	1
Rational.simplify_14	ROR	<1	<1	61	898	3	49
Rational.simplify_15	ROR	<1	<1	15	74	1	3
Rational.simplify_16	ROR	<1	<1	104	388	3	7
Rational.simplify_17	ROR	<1	<1	104	674	3	15
Rational.simplify_18	ROR	<1	<1	34	280	1	7
Rational.simplify_19	AOR	<1	<1	47	274	4	11
Rational.simplify_20	AOR	<1	<1	43	266	4	11
Rational.simplify_21	AOR	<1	1	72	550	8	37
Rational.simplify_22	STD	<1	<1	37	246	2	7
Rational.simplify_23	AOR	<1	6	49	230	5	19
Rational.simplify_24	AOR	<1	<1	49	230	5	19
Rational.simplify_25	AOR	<1	95	83	418	9	35
Rational.simplify_26	STD	<1	<1	49	230	5	19
Rational.simplify_27	AOR	<1	<1	29	338	0	1
Rational.simplify_2_16	ROR^2	<1	<1	138	420	6	9
Rational.simplify_2_27	ROR,AOR	<1	<1	63	370	3	3
Rational.simplify_3_11	ROR^2	<1	<1	108	368	3	12
Rational.simplify_16_27	ROR,AOR	<1	<1	104	388	3	7
Rational.simplify_2_16_27	ROR^2, AOR	<1	<1	138	420	6	9

Table 1: Experimental results for the Rational subjects.

SHADOW$_{JPF}$ needs initial test inputs, so we generated test inputs that each test case covers at least one change-statement, similar to the assumption in [13]. For SHADOW$_{JPF+}$ we do not need these concrete inputs. We added the change-annotations to the mutants and executed them with both: SHADOW$_{JPF}$ and SHADOW$_{JPF+}$. Afterwards, we manually compared the resulting path conditions. For our experiments we disabled the guided symbolic exploration because due to the small sizes of the mutated subjects it did not provide any time benefit.

[2] `https://github.com/JodaOrg/joda-time/issues/328`
[3] `https://github.com/JodaOrg/joda-time/issues/190`

Subject	Type	Time [s]		# States		# Paths (diff)	
		SJ	*SJ+*	*SJ*	*SJ+*	*SJ*	*SJ+*
WBS.update_1	ROR^8	<1	1	70	880	2	24
WBS.update_2	ROR^8	<1	<1	73	428	2	12
WBS.update_3	ROR^7, AOR	<1	<1	51	554	2	24
WBS.update_4	ROR^6, AOR, STD	<1	<1	97	618	4	18
WBS.update_5	ROR^7, AOR	<1	<1	109	266	6	12
WBS.update_6	ROR^8	<1	<1	135	632	6	24
WBS.update_7	ROR^6, AOR, STD	<1	<1	123	618	6	28
WBS.update_8	ROR^5, AOR^2, STD	<1	<1	147	232	8	8
WBS.update_9	ROR^5, AOR^2, STD	<1	<1	89	576	4	12
WBS.update_10	ROR^7, AOR	<1	<1	118	914	4	7
WBS.launch_1	ROR^8	4	121	11724	281080	576	13824
WBS.launch_2	ROR^8	<1	2	1083	12944	36	432
WBS.launch_3	ROR^7, AOR	7	120	20701	248354	1152	13824
WBS.launch_4	ROR^6, AOR, STD	3	47	10208	111876	628	5472
WBS.launch_5	ROR^7, AOR	<1	1	1717	3506	111	222
WBS.launch_6	ROR^8	11	76	32508	195176	1600	9600
WBS.launch_7	ROR^6, AOR, STD	7	146	22414	313930	1152	16128
WBS.launch_8	ROR^5, AOR^2, STD	2	14	7313	15232	512	896
WBS.launch_9	ROR^5, AOR^2, STD	3	56	7585	143819	745	7109
WBS.launch_10	ROR^7, AOR	30	193	48460	497118	2404	15204

Table 2: Experimental results for the WBS subjects.

Subject	Time [s]		# States		# Paths (diff)	
	SJ	*SJ+*	*SJ*	*SJ+*	*SJ*	*SJ+*
Foo	<1	<1	11	18	1	2
Joda-Time	<1	<1	37	40	9 (6)	6

Table 3: Experimental results for the motivating example and the presented Joda-Time regression bug.

Infrastructure: The experiments were conducted on a machine with macOS 10.14.6 (2.9GHz Intel Core i5, 16 GB RAM). As constraint solver for the symbolic execution we use Z3 [4] with the version 4.5.0. We used Java v1.8.0_211 and configured the symbolic execution with an unbounded depth limit and a timeout of one hour.

4.1 Results and Analysis

Tables 1 and 2 show the detailed results of the mutant evaluation for the Rational and WBS subjects. The first column names the corresponding class and method that were tested together with an id, which specifies each mutant. Column *Type* contains the mutation change type. The following columns describe the execution time in seconds, the number of visited states during the symbolic exploration, and the number of resulting path conditions for SHADOW$_{JPF}$ (*SJ*) and our extension SHADOW$_{JPF+}$ (*SJ+*). Table 1 contains two mutations (Rational.gcd_8 and Rational.simplify_12), for which the mutated version results in an infinite loop, hence, we marked them with *inf* and omitted them from the analysis. We also present in Table 3 the detailed execution results for the method `foo()` from Listing 1 and the Joda-Time regression bug.

RQ.1 Effectiveness: In order to answer RQ.1 we compared the number of test cases, i.e., resulting path conditions, identified by SHADOW$_{JPF}$ and SHADOW$_{JPF+}$ (see Table 1, 2, and 3). In almost all cases SHADOW$_{JPF+}$ was able to identify the same or a greater number of *diff* paths than SHADOW$_{JPF}$. The exception is the result for the subject Joda-Time, for which SHADOW$_{JPF}$ identified 9 *diff* paths and SHADOW$_{JPF+}$ identified 6 *diff* paths. However, SHADOW$_{JPF}$ is affected by the over-approximation mentioned in Section 2.2, and hence, it identifies incorrectly three paths as *diff* paths. For the rest, SHADOW$_{JPF}$ is often limited by the concrete values, which constraint the current path condition at a divergence point. Therefore, SHADOW$_{JPF+}$ can identify significantly more *diff* paths and at the same time is more accurate because it mitigates the over-approximation problem. Note that in our experiments, SHADOW$_{JPF+}$ was able to find all possible *diff* paths (except for the subjects marked with *inf*), since there was no further bound on the exploration depth.

RQ.2 Performance: In order to answer RQ.2 we compared the run-time and the number of visited states during exploration, which SHADOW$_{JPF}$ and SHADOW$_{JPF+}$ needed to identify the *diff* paths (see Table 1, 2, and 3). Due to the complete four-way symbolic execution, SHADOW$_{JPF+}$ needs in general more symbolic states than SHADOW$_{JPF}$. The run-time strongly depends on the number of explored states, and hence, on the size of the generated path conditions. Therefore, also the used constraint solver has a strong influence on the run-time, but with Z3 we are using one of the state-of-the-art constraint solvers. Needing more states in exploration, and hence a longer run-time, is the trade-off one has to decide on if a more accurate regression analysis is targeted.

RQ.3 Exposing Real-World Regression Bugs: In order to show that SHADOW$_{JPF+}$ can expose real-world regression bugs we applied our approach on a patch in Joda-Time, which introduced a regression bug. The `ZonedChronology` class in Joda-Time contains a method `localToUTC()`, which allows to convert a local instant to a standard UTC instant with the same local time. An instant object stores the number of milliseconds from the standard Java epoch of 1970-01-01T00:00Z. Prior to a patch, the method would calculate the UTC instant by subtracting the appropriate time offset from the local instant. A possible overflow caused by this operation was not specifically handled by the method. A patched version was supposed to fix this behavior by checking for an overflow. However, this patch introduced a regression bug that would cause the method to return wrong results under certain circumstances. Some minor refactorings had to be applied to make the program ready for our symbolic analysis. SHADOW$_{JPF+}$ generated 6 test inputs that trigger a divergent behavior, from which 4 divergences were expected based on the patch. However, two divergences are unexpected (i.e., unintended introduced semantic changes), and hence, represent a regression bug.

5. CONCLUSION & DISCUSSION

In this work, we presented an approach to generate test inputs exposing the divergences between two program versions. We provided a complementary exploration strategy to the existing technique by [13] and implemented SHADOW$_{JPF+}$ as an extension of the SHADOW$_{JPF}$ tool. In order to evaluate the effectiveness of our approach, we performed experiments on 79 generated mutants and compared the results with SHADOW$_{JPF}$. Additionally, we applied our approach on a patch for the Joda-Time library to show the capability to expose real-world regression bugs. We acknowledge that for the real-world applicability of regression testing it is very important to reduce the search space of symbolic execution to improve its scalability. Nevertheless, it is important to find the appropriate balance between keeping the search space practicable and still finding all crucial regression bugs. Our approach is more accurate and more precise than SHADOW$_{JPF}$, but more computational expensive. In future we plan to address this disadvantage by combining SHADOW$_{JPF+}$ with other techniques in a hybrid analysis approach.

References

[1] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[2] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[3] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, Sept 1976.

[4] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, Apr. 2001.

[6] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 55–64, New York, NY, USA, 2010. ACM.

[7] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.

[8] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[9] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[10] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks. *Directed Symbolic Execution*, pages 95–111. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[11] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[12] Y. Noller, H. L. Nguyen, M. Tang, and T. Kehrer. Shadow symbolic execution with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 42(4):1–5, Jan. 2018.

[13] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1181–1192, New York, NY, USA, 2016. ACM.

[14] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[15] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[16] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 26–36, New York, NY, USA, 2011. ACM.