Abstraction-Aware Inference of Metamorphic Relations

AGUSTÍN NOLASCO, University of Rio Cuarto, Argentina
FACUNDO MOLINA, IMDEA Software Institute, Spain
RENZO DEGIOVANNI, Luxembourg Institute of Science and Technology, Luxembourg
ALESSANDRA GORLA, IMDEA Software Institute, Spain
DIEGO GARBERVETSKY, University of Buenos Aires and ICC/CONICET, Argentina
MIKE PAPADAKIS, University of Luxembourg, Luxembourg
SEBASTIAN UCHITEL, Imperial College, United Kingdom and University of Buenos Aires and ICC/CONICET, Argentina
NAZARENO AGUIRRE, University of Rio Cuarto, Argentina and CONICET, Argentina
MARCELO F. FRIAS, The University of Texas at El Paso, USA

Metamorphic testing is a valuable technique that helps in dealing with the oracle problem. It involves testing software against specifications of its intended behavior given in terms of so called *metamorphic relations*, statements that express properties relating different software elements (e.g., different inputs, methods, etc). The effective application of metamorphic testing strongly depends on identifying suitable domain-specific metamorphic relations, a challenging task, that is typically manually performed.

This paper introduces MemoRIA, a novel approach that aims at automatically identifying metamorphic relations. The technique focuses on a particular kind of metamorphic relation, which asserts equivalences between methods and method sequences. MemoRIA works by first generating an object-protocol abstraction of the software being tested, then using fuzzing to produce candidate relations from the abstraction, and finally validating the candidate relations through run-time analysis. A SAT-based analysis is used to eliminate redundant relations, resulting in a concise set of metamorphic relations for the software under test. We evaluate our technique on a benchmark consisting of 22 Java subjects taken from the literature, and compare MemoRIA with the metamorphic relation inference technique SBES. Our results show that by incorporating the object protocol abstraction information, MemoRIA is able to more effectively infer meaningful metamorphic relations, that are also more precise, compared to SBES, measured in terms of mutation analysis. Also, the SAT-based reduction allows us to significantly reduce the number of reported metamorphic relations, while in general having a small impact in the bug finding ability of the corresponding obtained relations.

CCS Concepts: • **Theory of computation → Program specifications**; • **Software and its engineering → Software testing and debugging**; **Dynamic analysis**.

Additional Key Words and Phrases: Metamorphic testing, oracle problem, grammar-based fuzzing.

∗Renzo Degiovanni's participation in this work started when he was affiliated to the University of Luxembourg, Luxembourg.

Authors' addresses: Agustín Nolasco, University of Rio Cuarto, Rio Cuarto, Argentina, nolasco@dc.exa.unrc.edu.ar; Facundo Molina, IMDEA Software Institute, Madrid, Spain, facundo.molina@imdea.org; Renzo Degiovanni, Luxembourg Institute of Science and Technology, Luxembourg, Luxembourg, renzo.degiovanni@uni.lu; Alessandra Gorla, IMDEA Software Institute, Madrid, Spain, alessandra.gorla@imdea.org; Diego Garbervetsky, University of Buenos Aires and ICC/CONICET, Buenos Aires, Argentina, diegog@dc.uba.ar; Mike Papadakis, University of Luxembourg, Luxembourg, Luxembourg, michail.papadakis@uni.lu; Sebastian Uchitel, Imperial College, London, United Kingdom and University of Buenos Aires and ICC/CONICET, Buenos Aires, Argentina, sebastian.uchitel@gmail.com; Nazareno Aguirre, University of Rio Cuarto, Rio Cuarto, Argentina and CONICET, Buenos Aires, Argentina, naguirre@dc.exa.unrc.edu.ar; Marcelo F. Frias, The University of Texas at El Paso, El Paso, USA, mfrias@itba.edu.ar.

ACM 2994-970X/2024/7-ART21
https://doi.org/10.1145/3643747

ACM Reference Format:
Agustín Nolasco, Facundo Molina, Renzo Degiovanni, Alessandra Gorla, Diego Garbervetsky, Mike Papadakis, Sebastian Uchitel, Nazareno Aguirre, and Marcelo F. Frias. 2024. Abstraction-Aware Inference of Metamorphic Relations. *Proc. ACM Softw. Eng.* 1, FSE, Article 21 (July 2024), 23 pages. https://doi.org/10.1145/3643747

1 INTRODUCTION

While it is now relatively easy to produce large sets of software behaviors, e.g., via automated test sequence or input generation, or by automatically executing and monitoring the executions of deployed software, it is hard to use the generated behaviors for finding business logic bugs. This is due to the fact that deciding, for a given system, if its executions correspond to correct/desired software behavior as opposed to defective or in some sense anomalous behavior, is an inherently manual task. This situation has caused the so-called *oracle problem*, i.e. the problem of effectively distinguishing correct from incorrect software behavior, to receive significant attention by the software engineering community [5]. Still, with the current state-of-the-art, precise software oracles largely depend on manually produced specifications, or when automatically synthesized, often involve implicit oracles that only check very general properties, such as deadlock freedom and no null dereference.

To better deal with the oracle problem, metamorphic testing has been proposed [32, 43, 44]. Instead of manually describing the expected behavior for every single execution scenario, a weaker form of oracle that applies to a large family of executions is provided at once. This oracle is known as a metamorphic relation, and describes an expected property that relates two or more elements of the software under test (SUT) [43, 44]. The resulting "metamorphic" oracle is in general weaker than a precise and complete specification of the software under analysis, but at the same time is also easier to define and provide. Moreover, metamorphic properties capture properties that are specific to the software under analysis, as opposed to the general properties that implicit oracles describe. As concrete examples of metamorphic properties, consider the following. If a program `p` is an implementation of a mathematical commutative function, then an expected metamorphic property of the implementation is that `p(x,y) = p(y,x)`, for all instances of `x` and `y`. Similarly, for an object-oriented (unbounded) stack implementation, the execution of `o.push(e); o.pop()` from any non-null object `o` and element `e`, should take the object `o` back to its original state.

A key issue in applying metamorphic testing is the identification of suitable domain-specific metamorphic relations [3, 11, 43, 44]. While most approaches rely on user-defined relations [9], the difficulty of finding expressive domain-specific execution properties is directing attention to the problem of automatically discovering, or synthesizing, metamorphic relations [8, 12, 21, 42, 48, 50, 51]. Some recent approaches gather likely metamorphic relations from code comments [8], or from software trace generation and monitoring [21]. Unfortunately, these approaches have issues that limit their application in practice [3]. For instance, MeMo [8], the approach that gathers metamorphic properties from code comments, can be very efficient, but its application is limited to very well documented and mature software projects. On the other hand, SBES [21] runs a search algorithm in order to identify different SUT method executions that lead to the same result, which makes SBES costly since it needs to analyze many combinations of SUT executions in order to discover metamorphic relations.

In this paper, we present MemoRIA, an approach to automatically generate metamorphic relations that describe the current behavior of a software under analysis. MemoRIA focuses on a particular kind of metamorphic relation, that asserts equivalences between methods and method sequences of the SUT. As opposed to techniques such as MeMo, MemoRIA does not require descriptive code comments, as it generates metamorphic relations solely from the code of the SUT. At the same time, MemoRIA is more efficient and effective than techniques such as SBES, thanks to the use of automatically generated abstractions, that guide the search of candidate metamorphic relations. More precisely, our technique starts by automatically generating a finite state machine abstraction of the SUT, known as *enabledness-preserving abstraction* (EPA) [13, 15]; the EPA has abstract states that group together concrete states of the SUT in which the same methods of the SUT are *enabled* (i.e., can be invoked), and transitions that correspond to SUT's methods, and indicate how routines of the SUT transition between the abstract enabledness-based states. The EPA over-approximates legal method execution sequences, and is used by MemoRIA to guide a fuzzer to effectively generate candidate "EPA-aware" metamorphic relations. Then, MemoRIA uses run-time analysis to validate candidate properties and discard those deemed invalid, i.e., falsified during the execution of the SUT. Finally, in order to report a concise set of metamorphic relations better suited for inspection by developers, MemoRIA implements a SAT-based analysis to detect and discard redundant metamorphic relations.

We implement MemoRIA and evaluate it on a dataset of 22 Java classes taken from the literature. Our results show that EPAs can effectively guide MemoRIA to reduce the search space of candidate metamorphic relations. We observe that on average 83% of randomly generated candidate method sequence equivalences do not comply with the corresponding EPA, and thus are guaranteed to be invalid (MemoRIA prevents their generation by producing only EPA-coherent candidate method sequence equivalences). MemoRIA is also able to report compact sets of candidate metamorphic relations: the SAT-based mechanism to identify and discard redundant metamorphic relations allows us to reduce by 70%, on average, the number of inferred metamorphic relations.

Finally, we compare MemoRIA with the metamorphic relation inference technique SBES [21]. Our experimental evaluation shows that MemoRIA is 4.9 times faster, and produces metamorphic relations with a good mutation killing ability, considering that metamorphic relations are inherently weak oracles: MemoRIA detects on average 18.5% of mutants, compared to the 11% of mutants that SBES detects on average. Interestingly, these numbers go to 59.1% and 48.7%, respectively, when the inferred metamorphic relations are used to augment regression suites produced by EvoSuite to cover the EPA.

2 BACKGROUND

2.1 Metamorphic Relations

Metamorphic testing [43] is a property-based testing technique whose aim is to mitigate the *oracle problem* [5], i.e., the problem of effectively deciding whether the actual behavior of a software under test is consistent with its expected behavior or not. Indeed, while the typical way of realizing a test oracle is by running the SUT under specific inputs and comparing the actual outputs with the expected ones, metamorphic testing proposes to provide a weaker form of test oracle via the so-called *metamorphic relations*. Intuitively, a *metamorphic relation* (MR) is a relation capturing an expected property of one or more sequences of invocations of the SUT [43]. For instance, a numeric function implementing the sine function is expected to satisfy the following metamorphic relation `-sin(x) = sin(-x)`, for every value of `x`. Notice that this metamorphic oracle captures the expected output of `sin(x)` in terms of another invocation to the same function, but with a different input (`sin(-x)`), for any arbitrary value of `x`. Then, metamorphic testing can generate effective test cases by running the function with different values for `x`, and using the metamorphic property as a test oracle. It is of course also possible to define metamorphic relations that involve different method invocations from the SUT. For instance, for the Stack implementation in the Java standard library, it is expected that, for every non-empty stack `s`, calling `s.pop()` is equivalent to calling `s.remove(size()-1)`; thus, the equivalence of these two methods is an expected metamorphic property of non-empty stacks. These kinds of metamorphic properties have various applications.

In particular, Carzaniga et al. [9] exploit the functional redundancy in Java components to express metamorphic properties that capture these redundancies (as in the Stack example), and exploit these to generate what they called *cross-checking oracles*.

Identifying suitable domain-specific metamorphic relations is one of the main limitations for the application of metamorphic testing in practice [3, 43]. For the case of metamorphic relations that describe equivalences of methods and method sequences, their identification has applications not just in metamorphic testing, but also in other software engineering problems such as cross checking oracle construction [9], and run-time fault recovery [10], among others. Precisely, MemoRIA focuses on the inference of metamorphic relations that express (conditionally) equivalent method sequences, similar to techniques such as SBES [21].

2.2 Enabledness-Preserving Abstractions

The idea of Enabledness-Preserving Abstractions (EPAs), first introduced in [16], is closely related to the concept of *object protocol*. An object protocol is a restriction on the order in which the methods of a particular object can be invoked [6]. For instance, the interface `java.util.ListIterator` [38] in Java imposes a call protocol in which invocations to methods `remove` or `set` are enabled only if preceded by successful calls to methods `next` or `previous`. Enabledness-Preserving Abstractions are essentially behavior models that capture object protocols. They over-approximate the sequences of successfully executing method calls, by providing a state machine behavior model whose (abstract) states correspond to concrete object states in which exactly the same set of methods of the SUT, are enabled. Transitions represent methods of the SUT, that transition from an EPA state S_i to another EPA state S_j if and only if there exists an execution of the method from a concrete state in S_i, that leads to a state in S_j.

Let us more formally introduce the notion of EPA.

Definition 1 (Enabledness-preserving Abstractions). *Given a class C with a set $M = m_1, \ldots, m_n$ of methods, an EPA of the object protocol of C is a finite labeled transition system $\langle M, S, s_0, \delta \rangle$, where:*

- *S is composed of the subsets of the set of methods, i.e., $S = 2^M$. Each element of S is an abstract state of the EPA, which captures all enabledness-equivalent states of C with respect to the methods in S: a concrete state c of C belongs to the abstract state S iff there exists at least one successful (i.e., non-failing) execution of each method in S, from state c.*
- *The initial state s_0 represents the pseudo program states of C where the constructors of C can be called (these are pseudo states of C, since the execution of a constructor is what leads to a proper state of C objects).*
- *Tuple (S_i, m, S_j) belongs to δ iff there exists a successful (i.e., non-failing) execution of method m from a concrete state c that corresponds to S_i (i.e., where all the methods in S_i are enabled), that terminates in a concrete state c' corresponding to the abstract state S_j (i.e., where all the methods in S_j are enabled).*

As a specific example, consider class `Stack` from `java.util`. The labeled transition system in Figure 1 represents the EPA of this class.

EPAs can be automatically computed from the source code of a class or component, using different approaches, including constraint solving [14], and run-time analysis [20]. They have many applications, including behavior validation [14] and test generation [20]. In this paper, we will employ EPAs as a way of guiding the search for candidate metamorphic relations that capture equivalences of method sequences. Furthermore, EPAs will also help us partition the set of candidate metamorphic relations: as we will see later on, each candidate metamorphic relation will state an equivalence between two sequences of method calls *from a specific abstract EPA state*.

2.3 Grammar-Based Fuzzing

Fuzzing is one of the most successful testing techniques, with an increasing adoption both in research [1] and industry [22–24]. A *fuzzer* automatically generates values, typically in a random way, that can be used for many purposes, specifically as program inputs for testing and bug finding. Fuzzers can often produce thousands of inputs very efficiently, leading to effective mechanisms for inducing program crashes (e.g., due to wrongly handled invalid inputs), finding security vulnerabilities, and in general for bug finding [33].

Grammar-based fuzzing takes a grammar capturing the inputs language and can produce syntactically-valid inputs by traversing the production rules of the grammar [49]. A *grammar-based fuzzer* starts from a string representing the initial grammar symbol, and systematically replaces every non-terminal symbol through the application of some production rule of the corresponding non-terminal, usually randomly selected, until all symbols in the string are terminals. Among the vast applications of grammar-based fuzzing, e.g., in the search for vulnerabilities [26, 34], it has also been used for generating candidate program assertions from an assertion grammar, which are then validated to automatically produce program specifications [35].

As we will describe in detail later on, MemoRIA will also use a grammar-based fuzzer to produce *candidate metamorphic relations*, constituted by equivalences of method sequences. The grammar will be based on the Enabledness Preserving Abstraction of the SUT, in a way that will allow the fuzzer to produce EPA-aware method sequences, thus discarding method sequences that are not consistent with the EPA, and that would therefore necessarily correspond to invalid method sequencing.

3 ILLUSTRATIVE EXAMPLES

The application of metamorphic testing and other related techniques, and their corresponding effectiveness, depend heavily on being able to identify suitable domain-specific properties of the software under analysis. Since such identification is typically costly and non-trivial, some techniques have been proposed, to aid with this task. One such technique is SBES [21], that aims at inferring metamorphic relations expressing equivalent method sequences. SBES is a search-based technique, that is able to synthesize sequences of method invocations that are equivalent to a target method, according to a finite set of execution scenarios (test cases). Given a class C with a target method m, SBES first executes EvoSuite [19] to generate initial test scenarios whose last executed method is an invocation to m. Then, the process continues with two phases. The first phase generates a new class C' with all the methods of C but without m, and uses EvoSuite to produce sequences of methods likely to be equivalent to m. The second phase takes the likely equivalent method sequences, and executes EvoSuite again, this time with the goal of finding counterexamples to the candidate equivalent sequences (i.e., executions that witness discrepancies between m and the candidate equivalent sequence). Thus, SBES aims at inferring metamorphic relations that have a common pattern: sequences of methods that are equivalent to the subject method m (these will be tested on *all* the collected scenarios, and if passed, will be assumed to hold in all object states where m is executable).

Let us consider, as an example, the `Stack` implementation [39] from the Java Standard Library, that implements an *unbounded* stack data structure, with a very simple object protocol. When the stack is empty, some methods from the class (`pop()`, `peek()`), and other methods from parent classes that require the presence of at least one element (such as `remove(E)`) cannot be executed. When the stack is not empty, all methods of the class are enabled. This causes methods such as `push(E)` or the inherited `addElement(E)` to be enabled from any (non-null) Stack state. Figure 1 illustrates the EPA that precisely captures the object protocol of this class. Analyzing this class using

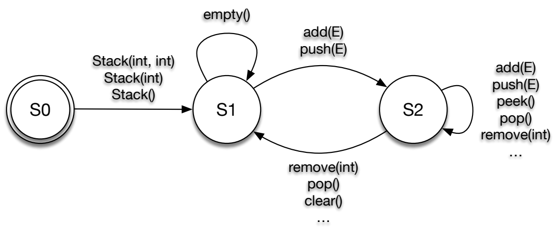


Fig. 1. EPA of class java.util.Stack.

Table 1. Sample of metamorphic relations involving method Stack.pop(), inferred by SBES and MemoRIA.

SBES
`pop() = peek(); removeElementAt(size() - 1)`
`pop() = remove(size() - 1)`
MemoRIA
$\{S1\} \Rightarrow \epsilon =$ `push(item);pop()`
$\{S2\} \Rightarrow$ `clear() = pop();clear()`

SBES allows us to find various method sequence equivalences. For instance, SBES identifies that method `pop()` is equivalent to `remove(size()-1)` in all the collected scenarios where `pop()` is executable, and thus reports this equivalence, among others. Table 1 summarizes the metamorphic relations involving `pop()` that SBES produces.

Now consider a very similar example, the implementation of a *bounded* stack taken from [20], and shown in Figure 2. It is relatively straightforward to see that not all the operations of class `MyBoundedStack` can be executed in any object state. For instance, if the stack is full, an invocation to the method `push(Object)` will throw an exception. Similarly, if the stack is empty, calling method `pop()` will also throw an exception. Figure 3 illustrates the EPA that summarizes the object protocol of `MyBoundedStack`. When analyzing this class with SBES, the technique is not able to report *any* metamorphic relation. This is so even though we can manually spot some method sequence equivalences (e.g., `pop()` being equivalent to `push(x); pop(); pop()` when the stack is not full). The reason here is many-fold: SBES does not involve the subject method in the candidate equivalent sequences (so, you cannot see `pop()` being involved in a candidate equivalent sequence for `pop()`). Moreover, candidate sequences will be evaluated on *all* the collected scenarios where the subject method is executable, without considering the enabledness of the candidate sequence itself. For classes with more complex object protocols, since most methods involved in candidate sequences would only be enabled in a reduced set of states, candidate sequences will more scarcely result to be "enabled" as a whole (notice that this is even the case for `MyBoundedStack`, whose object protocol is just slightly more complex than that of `java.util.Stack`).

Our first idea is to take advantage of the object protocol described by the EPA to produce metamorphic relations in the form of *conditionally equivalent method sequences*, conditional in the sense that they state equivalent method sequences but only for executions starting at *specific* (abstract) states in the EPA. Moreover, we can further exploit the EPA to guide the search of candidate method sequence equivalences; intuitively, if a method sequence is not *EPA-compliant*, then it is guaranteed to fail, and thus cannot participate in any metamorphic relation; similarly, if two method sequences lead to different abstract EPA states (from a same origin), then they are guaranteed to lead to different concrete states, and therefore cannot be equivalent. For instance, for the `MyBoundedStack` example, sequences `MyBoundedStack()` and `MyBoundedStack();push(x)` cannot be equivalent from abstract state `S0`, as they lead to different abstract states (`{S1}` and `{S2}`, respectively). As we will further describe later on, our technique uses the EPA to produce, using a grammar-based fuzzer, candidate metamorphic properties that are necessarily EPA-compliant. The generated candidate EPA-compliant metamorphic relations are validated using a run-time checking stage: test cases are generated and executed to *test* the candidate metamorphic equivalences; the non-falsified metamorphic relations are kept, and the invalidated ones discarded. Finally, to retrieve a minimal set of metamorphic relations, with as little redundancy as possible, MemoRIA implements a logical satisfiability checking that allows us to discard metamorphic relations that are subsumed (logically

```
public class MyBoundedStack {
    private final Object[] elements;
    private int index;

    public MyBoundedStack() {
        elements = new int[3];
        index = -1;
    }

    public void push(Object o) {
        if (isFull()) throw new IllegalStateException();
        elements[++index] = o;
    }

    public Object pop() {
        if (isEmpty()) throw new IllegalStateException();
        return elements[index--];
    }

    public boolean isFull() { return index == elements.length - 1;}

    public boolean isEmpty() { return index ==  -1;}
}
```

Fig. 2. Implementation of class MyBoundedStack.

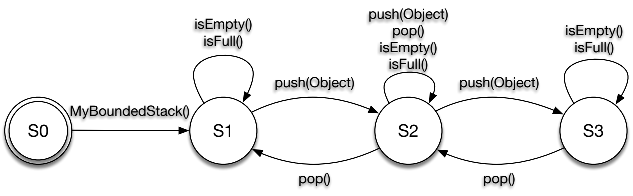


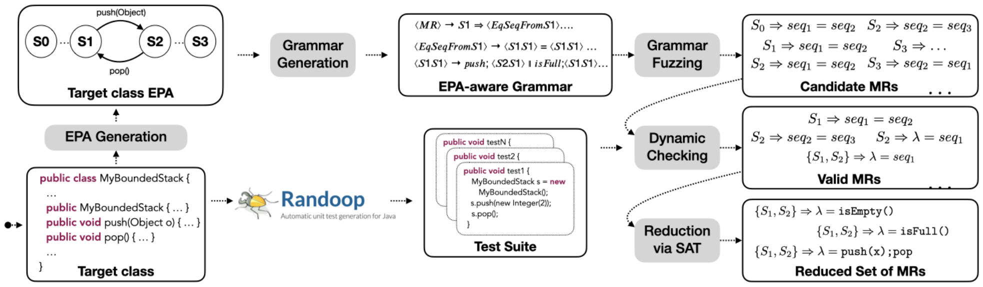
Fig. 3. EPA of class MyBoundedStack.

$\{S1, S2, S3\} \Rightarrow \epsilon =$ isEmpty()
$\{S1, S2, S3\} \Rightarrow \epsilon =$ isFull()
$\{S1, S2\} \Rightarrow \epsilon =$ push(Object);pop()
$\{S2\} \Rightarrow \epsilon =$ push(Object);isEmpty();pop()

Fig. 4. MRs generated by MEMORIA for `MyBoundedStack`.

implied) by others also being reported. This leads MEMORIA to producing more concise sets of metamorphic relations.

Table 1 shows a sample of metamorphic relations inferred by MEMORIA for the unbounded `Stack`, involving method `pop()`. The first states that, if the stack is empty, performing a `push` followed by a `pop` takes back the stack to the initial state, i.e., `pop` reverts `push` (ϵ denotes the empty method sequence). The second states that when the stack is non-empty, then performing `clear()` will lead to exactly the same object state as first doing `pop()` and then `clear()` (i.e., both sequences always take the object to the empty stack state). Neither of these sequences would be inferred by SBES: the first involves the empty sequence and does not follow the "subject method" pattern; the second would be invalidated when trying to execute the candidate sequence (right-hand side sequence) if the empty stack is one of the scenarios collected for the execution of `clear()`. Figure 4 also shows a sample of the metamorphic relations that MEMORIA generates for `MyBoundedStack`. Notice that, as motivated earlier, our inferred metamorphic relations are conditional: they have an antecedent stating the relevant abstract EPA state where the equivalence applies, and a consequent where the two sequences which are deemed to be equivalent are stated.


push(Object)
S0
S1
S2
S3
pop()
Target class EPA
Grammar Generation
$\langle MR\rangle \rightarrow S1 \Rightarrow \langle EqSeqFromS1\rangle \ldots$
$\langle EqSeqFromS1\rangle \rightarrow \langle S1S1\rangle = \langle S1S1\rangle \ldots$
$\langle S1S1\rangle \rightarrow push; \langle S2S1\rangle \mid isFull; \langle S1S1\rangle \ldots$
EPA-aware Grammar
Grammar Fuzzing
$S_0 \Rightarrow seq_1 = seq_2$ $S_2 \Rightarrow seq_2 = seq_3$
$S_1 \Rightarrow seq_1 = seq_2$ $S_3 \Rightarrow \ldots$
$S_2 \Rightarrow seq_1 = seq_2$ $S_3 \Rightarrow seq_2 = seq_1$
Candidate MRs
EPA Generation
Dynamic Checking
$S_1 \Rightarrow seq_1 = seq_2$
$S_2 \Rightarrow seq_2 = seq_3$ $S_2 \Rightarrow \lambda = seq_1$
$\{S_1, S_2\} \Rightarrow \lambda = seq_1$
Valid MRs
Randoop
Automatic unit test generation for Java
Test Suite
Target class
Reduction via SAT
$\{S_1, S_2\} \Rightarrow \lambda =$ isEmpty()
$\{S_1, S_2\} \Rightarrow \lambda =$ isFull()
$\{S_1, S_2\} \Rightarrow \lambda =$ push(x);pop
Reduced Set of MRs

```
public class MyBoundedStack {
  ...
  public MyBoundedStack { ... }
  public void push(Object o) { ... }
  public void pop() { ... }
  ...
}
```

```
public void testN {
public void test2 {
public void test1 {
  MyBoundedStack s = new
    MyBoundedStack();
  s.push(new Integer(2));
  s.pop();
}
```

Fig. 5. Overview of MemoRIA.

4 THE TECHNIQUE

MemoRIA uses enabledness-preserving abstractions (EPAs), grammar-based fuzzing, run-time checking, and SAT-based analysis, to infer metamorphic relations of Java classes. Figure 5 shows an overview of MemoRIA's workflow. Given a Java class C, MemoRIA first automatically computes an enabledness-preserving abstraction EPA_c describing the object protocol of C, and extracts an EPA-aware grammar G_c. This grammar is provided as input to a grammar-based fuzzer in order to efficiently obtain candidate metamorphic relations (all compliant with the EPA, by construction). Then, MemoRIA performs a run-time checking process in which it validates every candidate metamorphic relation against an automatically produced test suite. The candidate relations that are not falsified are considered valid and preserved, while the remainder (invalidated by at least one test case) are discarded. Finally, in order to reduce the number of metamorphic relations reported to the engineer, and to detect and discard redundant ones, MemoRIA implements a satisfiability (SAT) based approach to approximate the minimal set of relations that subsume (logically imply) the set of all the inferred metamorphic relations. A detailed description of each of these stages is provided below.

4.1 EPA Generation

MemoRIA uses the tool Evo+EPA [20] to automatically compute the EPA for the target class C. Evo+EPA is an extension of EvoSuite [19] that produces an under-approximation of the actual EPA of a given class, using search-based dynamic analysis. It starts from an initial model with a unique initial state, where only class constructors are enabled. When a new test is created, the test is run and Evo+EPA checks, for each public method m, whether its precondition (a pure - no side effects - boolean method) is satisfied on the resulting object, thus determining the states of the EPA that are traversed during a test execution. In this way, the technique identifies the EPA source and target states for transitions labeled with the public methods involved in test cases, with test cases being paths in this dynamically generated EPA. The search-based test generation attempts to maximize the coverage of the EPA. As a result, the tool generates an EPA EPA_c, whose states and transitions are those covered by the generated test suite, thus under-approximating the actual EPA of class C. The test suite that led to the construction of the EPA is also returned by the tool; we will later on use these test suites as part of the tests used during the dynamic analysis stage, to evaluate MRs and discard those deemed invalid.

4.2 EPA-Aware Grammar Generation

Given the target class C and its EPA EPA_c, MemoRIA proceeds to automatically produce a grammar G_c capturing the language of candidate MRs over C. The language, as illustrated before, corresponds

$\langle MR \rangle \rightarrow S_1 \Rightarrow \langle EqSeqFromS_1 \rangle$
$\langle EqSeqFromS_1 \rangle \rightarrow \langle S_1S_1 \rangle = \langle S_1S_1 \rangle \parallel \langle S_1S_2 \rangle = \langle S_1S_2 \rangle \parallel \langle S_1S_3 \rangle = \langle S_1S_3 \rangle$
$\langle S_1S_1 \rangle \rightarrow$ `push;` $\langle S_2S_1 \rangle \parallel$ `isFull;` $\langle S_1S_1 \rangle \parallel$ `isEmpty;` $\langle S_1S_1 \rangle \parallel$ `isFull ; null` $\parallel$ `isEmpty ; null`
$\langle S_1S_2 \rangle \rightarrow$ `push;` $\langle S_2S_2 \rangle \parallel$ `push; null` $\parallel$ `isFull;` $\langle S_1S_2 \rangle \parallel$ `isEmpty;` $\langle S_1S_2 \rangle$
$\langle S_1S_3 \rangle \rightarrow$ `push;` $\langle S_2S_3 \rangle \parallel$ `isFull;` $\langle S_1S_3 \rangle \parallel$ `isEmpty;` $\langle S_1S_3 \rangle$
$\langle S_2S_2 \rangle \rightarrow$ `pop;` $\langle S_1S_2 \rangle \ldots$

Fig. 6. Fragment of the EPA-aware Grammar for class MyBoundedStack.

to conditional equivalences of method sequences. More precisely, each candidate metamorphic relation will have the form:

$$\{S_i, \ldots, S_j\} \Rightarrow Sq_1 = Sq_2$$

where $S_i, \ldots, S_j$ are EPA abstract states, and Sq_1 and Sq_2 are sequences of public methods of C. The above MR schema expresses that:

> *for every concrete state st of an object of C, if st belongs to one of the abstract states $S_i, \ldots, S_j$, then executing Sq_1 from st leads to the same concrete object state as executing Sq_2 from st.*

The grammar G_c allows us to obtain only candidate metamorphic relations that are consistent with the object protocol of C as captured by EPA_c, i.e., method sequences Sq_1 and Sq_2 are reproducible from states $\{S_i, \ldots, S_j\}$ in EPA_c, and lead to compatible sets of states in EPA_c (notice that an EPA is typically a non-deterministic labeled transition system). This stage automatically converts EPA_c into a context-free grammar (it can actually be transformed into a regular grammar) as follows. First, it defines $\langle MR \rangle$ as the starting symbol. Then, for each pair of states $(S_i, S_j) \in S \times S$, it incorporates to the grammar the following production rule:

$$\langle MR \rangle \rightarrow S_i \Rightarrow \langle S_iS_j \rangle = \langle S_iS_j \rangle$$

where the non-terminal $\langle S_iS_j \rangle$ represents valid sequences of method names (the alphabet of EPA_c) that start in state S_i of EPA_c and can end in state S_j of EPA_c.

Next, for each pair of states (S_i, S_j) in the EPA, MEMORIA generates production rules for the corresponding non-terminals $\langle S_iS_j \rangle$. That is, for each transition in δ of the form (S_i, m, S_k), the following rule is added:

$$\langle S_iS_j \rangle \rightarrow m; \langle S_kS_j \rangle$$

Additionally, we also incorporate the rule $\langle S_iS_k \rangle \rightarrow m; null$ to allow for sequences that end with method call m, where *null* is a special token we use to denote the end of the sequence. For instance, considering the states S_1 and S_2 and method `push` in the EPA for `MyBoundedStack` (Figure 3), we would add the following rules:

$$\langle S_1S_2 \rangle \rightarrow push; \langle S_2S_2 \rangle$$
$$\langle S_1S_2 \rangle \rightarrow push; null$$

Notice that the production rules prevent infeasible sequences in the EPA. For instance, from the EPA in Figure 3, the pair (S_2, S_0) is infeasible (no method sequences allow us to get from S_2 to S_0) and thus our process will have no production rules for $\langle S_2S_0 \rangle$. Figure 6 shows a fragment of the grammar generated for class `MyBoundedStack` (the full grammar for each subject is available in our replication package site [2]).

4.3 Fuzzing Candidate MRs

Once the grammar G_c is generated, MEMORIA proceeds to generate candidate metamorphic relations. To do so, it employs a grammar-based fuzzer that, given the grammar G_c, is able to produce candidate metamorphic relations for our target class C. The fuzzer we use is a Java implementation of the grammar-coverage fuzzer introduced in The Fuzzing Book [49]. The fuzzer produces strings of the grammar language $\mathcal{L}(G_c)$ by starting with the initial symbol $\langle MR \rangle$, and then by iteratively replacing non-terminal symbols until all non-terminals have been expanded into terminals. Moreover, the fuzzer tries to systematically cover all expansions of the grammar at least once, with the goal of maximizing variability. It is also possible to configure the fuzzer to generate up to a given maximum number of sequences (in our experiments we generate up to 1000 candidate metamorphic relations). We remark that the generated candidate metamorphic relations are EPA-consistent, as the grammar G_c captures method sequences consistent with EPA_c.

4.4 Dynamic Checking of Candidate MRs

To determine whether generated EPA-aware candidate metamorphic relations hold for the target class C, MEMORIA relies on dynamic analysis. In addition to the suite generated by Evo+EPA during the EPA generation phase (that maximizes EPA coverage), MEMORIA also uses the test generation tool Randoop [17] to generate further test cases. Both automatically generated suites are combined and used as follows. First, MEMORIA collects a set O_c of objects of class C generated by the suites. As our candidate metamorphic relations are associated to specific abstract states of EPA_c, MEMORIA first identifies, for each object $o \in O_c$, the abstract EPA state S_i it belongs to, in order to determine which candidate MRs should be checked on which objects. MEMORIA then checks the candidate MRs on the corresponding objects from O_c, discarding those falsified by at least one object; the surviving properties are considered valid and reported by this stage.

The specific mechanism to check a candidate MR on a given object works as follows. Let $S_i \Rightarrow seq_1 = seq_2$ be a candidate MR. Let o be an object generated by the test generation tools (Evo+EPA or Randoop), and seq_o the sequence of methods leading to object o (notice that both tools, Evo+EPA and Randoop, save the method sequences rather than the objects). We first check if o corresponds to S_i by evaluating which of the public methods of C can be called in o. If so, we execute method sequences $seq_o; seq_1$ and $seq_o; seq_2$, and compare the corresponding concrete object states obtained, say o_1 and o_2, respectively. The equality of these states (checked using `equals`) determines if the MR holds for o or not.

4.5 SAT-Based MR Reduction

The previous stage will result in a set MR_C of likely metamorphic relations for class C. As these were generated from a fuzzer, many of the obtained relations may be *redundant*, in the sense that they might be logically implied by other relations in the set. Since the reported metamorphic relations will generally go through manual inspection by the engineer (notice that we generate metamorphic relations that are consistent with the current behavior of the SUT according to a sample of SUT executions, which may not coincide with the *intended* behavior of the SUT), it is important to produce a compact set of metamorphic relations.

We therefore perform a reduction stage, where redundant relations are identified and removed. Our approach takes the EPA, interprets abstract states as sets of concrete states, and methods as uninterpreted partial functions between the corresponding EPA states. In this context, metamorphic relations are transformed into formulas. For instance, metamorphic relation:

$$\{S1, S2\} \Rightarrow \epsilon = \texttt{isEmpty()}$$

for `MyBoundedStack` is interpreted as the formula that states that function `isEmpty()` is contained in the identity function (since it is a partial function), when restricted to state sets $\{S1, S2\}$. Then, implications between these formulas are checked with a constraint solver, in order to discard metamorphic properties implied by others. As an example, given the above metamorphic property for `MyBoundedStack`, property:

$$\{S1\} \Rightarrow \texttt{push(Object); isEmpty()} = \texttt{push(Object)}$$

becomes redundant. We implemented the check for implications between metamorphic properties in Alloy [25]. Alloy performs a SAT-based bounded exhaustive analysis of the implications.

Since reducing a set of metamorphic properties to the minimal set of properties implying the rest is inherently combinatorial (and implication checks are costly), we proceed with a heuristic, that favors shorter (and thus more legible) metamorphic properties. We sort the metamorphic properties by decreasing length, and check in order, for each metamorphic property, if the others imply it. If they do, we discard the current property (remove it from the set), and continue with the next. If not, we keep the property.

It is worth remarking that although we only discard metamorphic properties that are logically implied by others, we may lose bug finding ability by leaving out these “weaker” properties. Although this is counterintuitive, this is actually the case because metamorphic properties are not state properties, but properties of executions. In our above example, for instance, a bug in `isEmpty()` that is triggered only after a `push` may be identified if both properties are maintained, but not if only the stronger is. Our experimental results will provide further details in this respect.

4.6 Examples of Candidate MRs

Table 2 shows five different candidate MRs for class `MyBoundedStack` (cf. Figure 2), that our approach supports. Particularly, since the first MR is not EPA-compliant (see Figure 3), MemoRIA will not produce this property, since the EPA is integrated into the grammar for fuzzing, and only EPA-compliant properties are generated in the first stage of our approach. The other four candidate MRs are EPA-compliant. MemoRIA would discard the second property during dynamic analysis, when testing the candidate property by pushing an item different from the one that is popped. The three remaining MRs (i.e., properties 3, 4 and 5 in Table 2) are considered valid, since these are consistent with the EPA and the dynamic analysis process did not invalidate them. Then, MemoRIA will apply the SAT-based reduction step and discard MR 3 because it is redundant with respect to MRs 4 and 5. Notice that, from a logical point of view, if both properties 4 and 5 are valid in abstract states $S1$ and $S2$, then we can prove the validity of property 3 in both states $S1$ and $S2$. This is automatically checked by using constraint solving as explained above. Finally, MemoRIA reports the reduced set of MRs (4 and 5) to the engineer for validation and analysis.

Table 2. Sample of candidate MRs for `MyBoundedStack` and how they are handled by MemoRIA.

MRs	**Valid?**	**Discarded in stage**
1- $\{S1\} \Rightarrow \epsilon =$ `pop();push(Object)`	No	EPA
2- $\{S2, S3\} \Rightarrow \epsilon =$ `pop();push(Object)`	No	Dynamic checking
3- $\{S1, S2\} \Rightarrow \epsilon =$ `push(Object);pop();isEmpty()`	Yes	SAT-based reduction
4- $\{S1, S2, S3\} \Rightarrow \epsilon =$ `isEmpty()`	Yes	-
5- $\{S1, S2\} \Rightarrow \epsilon =$ `push(Object);pop()`	Yes	-

5 RESEARCH QUESTIONS

Our first research question assesses the suitability of EPAs in discarding invalid metamorphic relations:

RQ1 *How effective are EPAs in discarding invalid metamorphic relations?*

To answer this question we study to what extent invalid metamorphic relations are non-conforming with the EPAs. We produce random conditional method sequence equivalences, and then execute our dynamic checking step to determine which of these are invalid. For each metamorphic relation deemed invalid, we check if it complies or not with the corresponding EPA, since we are interested in analyzing the impact of EPAs in discarding invalid candidate properties, prior to the dynamic analysis.

Our second research question analyzes the effect of MemoRIA's SAT-based approach in reducing the set of reported metamorphic relations:

RQ2 *How effective is the SAT-based reduction in eliminating redundant metamorphic relations?*

We assess effectiveness on two aspects. First, we examine the level of reduction achieved, i.e., the ratio between overall inferred valid relations, and those identified as redundant by the SAT-based approach. Second, we compare the effectiveness of the reduced set of valid metamorphic relations with respect to the entire set of inferred valid relations. Effectiveness is measured in terms of the ability of detecting artificially seeded faults (mutants).

Finally, we compare the performance of MemoRIA with related approaches:

RQ3 *How does MemoRIA compare with alternative techniques?*

To answer this question we compare MemoRIA with SBES [21], to the best of our knowledge, the only other technique that uses dynamic analysis for synthesizing metamorphic relations. We compare MemoRIA against SBES according to various metrics: execution time, number of reported properties, and effectiveness of the inferred relations for detecting mutants when used as test oracles during metamorphic testing.

6 EXPERIMENTAL SETUP

6.1 Subjects

For RQ1, we use the subjects taken from a recent object protocol benchmark [20], comprising 15 Java classes (with up to 24 methods) for which the corresponding EPAs are publicly available [18]. Table 3 reports, for each subject, the number of public methods (#M), the size of the EPA in terms of numbers of states (#S) and transitions (#δ). All these EPAs were automatically obtained using the Evo+EPA tool [20]. For RQ2 and RQ3 we also incorporate the subjects used in the evaluation of SBES [21]. These subjects include 7 classes: the `Stack` implementation from the Java Standard Library (our motivating example) and 6 other classes from the `graphstream` [46], a Java library for the modeling and analysis of dynamic graphs. Overall, we evaluate MemoRIA in a total of 22 Java classes, containing a total of 160 methods. All these subjects as well as their corresponding EPAs can be found in our replication package [2].

6.2 Experimental Procedure

To answer RQ1, we start by randomly generating 1000 candidate equivalent method sequences (ignoring the EPA) and building conditional MRs pairing them with every EPA state precondition. This leads to a total of 1000 * #S (number of EPA states) candidate metamorphic relations, that are later on evaluated with our dynamic checking procedure (cf. Section 4.4). As a result, we obtain sets $\mathcal{V}$ and $\mathcal{I}$ of valid and invalid metamorphic relations (valid/invalid with respect to the dynamic analysis), respectively. We analyze $\mathcal{I}$ to determine which of the invalid MRs would have been discarded due to not being consistent with the corresponding EPA; the result is the set $\mathcal{D}$ of EPA discarded MRs. Using this information, we measure the *recall*, i.e., the proportion of invalid relations (relations that are invalidated by the dynamic analysis) that are discarded by the EPA. To account for the randomness involved in the process, we repeated the experiment 10 times, and report the average recall.

For the remaining RQs, we infer MRs with three techniques:

MemoRIA(all). This technique is essentially MemoRIA without the SAT-based reduction. That is, this technique generates the EPA and the EPA-aware grammar, fuzzes candidate metamorphic relations, and filters them out using dynamic analysis. We fuzzed 1000 candidate MRs per subject, and set Randoop to produce a maximum of 2000 tests per subject.

MemoRIA(sat). This technique corresponds to the full MemoRIA approach. It employs exactly the same stages described in the previous technique, and then applies the SAT-based reduction on the produced valid MRs. The fuzzing and dynamic checking processes are performed with the same configuration as MemoRIA(all).

SBES. The Search-based Equivalent Sequences approach [21] synthesizes, from a target method, equivalent method sequences. We execute SBES on every public method of each subject class, using the same configuration for SBES used in [21]: 180 seconds for each of the two phases, and 30 executions (repetitions) for each method under analysis.

After executing each technique, we generate mutants with Major [28] and then analyze whether the set of MRs inferred by each of the techniques are able to detect the mutants, which represent seeded faults.

To answer RQ2, we focus on the output of the MemoRIA(all) and MemoRIA(sat) techniques. We study the overhead of using the SAT-based analysis, the reduction achieved in terms of the number of discarded MRs, as well as how the mutation score is affected by the SAT-based reduction.

Finally, to answer RQ3, we compare MemoRIA with SBES in terms of execution time, number of inferred properties, and the mutation score achieved. We also analyze to what extent the inferred MRs can complement the fault detection capabilities of the regression suites produced by Evo+EPA, which adapts EvoSuite to generate test suites and regression assertions covering the corresponding EPA.

As in RQ1, the execution of MemoRIA as well as the mutation analysis were repeated 10 times to cope with the stochastic nature of the process.

7 EXPERIMENTAL RESULTS

7.1 (RQ1) EPAs for Discarding Invalid MRs

Table 3 shows how effective EPAs are in discarding invalid MRs. We produce a total of *37,594* invalid MRs (across all subjects), of which 89% (*33,674*) do not comply with the EPA. On average per subject, 83% of the invalid MRs were also invalid with respect to the EPA. Furthermore, we observe that every MR invalidated by the EPA is actually invalid, i.e., there is no metamorphic property violating the EPA that is valid in the class under analysis. This result indicates that a prior EPA-awareness analysis can considerably reduce the cost of the dynamic analysis employed by MR inference techniques. In fact, this is what motivated the design of our mechanism for candidate generation: producing a grammar capturing the language of EPA-compliant method sequences, and then fuzzing from that grammar. It is worth remarking that non EPA-compliant candidate relations are necessarily invalid (confirmed by the precision in our experiments), implying that EPAs do not miss valid MRs.

Table 3. Ratio of EPA detected Invalid MRs. It reports number of public methods (#M), EPA size (number of states #S and transitions #δ), and number of invalid ($\mathcal{I}$) MRs *discarded* by the EPA ($\mathcal{D}$).

Subject	**#M**	**EPA**		**MRs**		
		#S	**#δ**	#$\mathcal{I}$	#$\mathcal{D}$	**Rec. (%)**
JDBCResultSet	24	9	127	6,185	5,939	96
ListItr	9	8	65	3,861	3,520	91
MyBoundedStack	4	4	13	1,358	1,095	81
NumberFormatStringTokenizer	4	4	13	1,371	1,035	76
NumberFormatStringTokenizer_m	4	3	9	998	719	72
SMTPProcessor	7	5	20	2,337	2,226	95
SMTPProcessor_h	9	12	85	6,119	5,801	95
SMTPProtocol	9	3	18	1,786	1,433	80
SftpConnection	20	5	29	2,944	2,790	94
Signature	10	4	20	2,013	1,879	93
Socket	7	7	20	3,780	3,752	99
StackAr	6	3	14	1,248	807	65
StringTokenizer	3	3	12	1,201	799	67
ToHTMLStream	7	2	8	932	474	51
ZipOutputStream	4	4	9	1,462	1,406	96
Total	127	76	462	37,594	33,674	89

7.2 (RQ2) Effectiveness of the SAT-Based Reduction

Table 4 shows the results of performing MR inference with MemoRIA, with SAT-based reduction enabled and disabled, and starting from 1000 candidate MRs per subject. MemoRIA ran successfully on all the subjects from [21], except for `MultiNode`, for which Evo+EPA failed to produce an output. Below we discuss the results according to each metric we used for comparison.

Time. The SAT-based reduction has a relatively low negative impact on the execution time for the majority of the subjects. In 4 cases the execution time increases significantly, but still remains in the order of few seconds up to 9 minutes per subject. We have only one outlier for which MemoRIA(sat) causes an increase of one to two orders of magnitude in the execution time: `ListItr`. This is one of the subjects with the highest number of transitions in the model (see Table 3), which in some cases can affect the performance of the SAT-based analysis. Overall, considering all subjects, the SAT-based analysis adds a total of *10,470* seconds. Even considering this time increase, our full approach is still much more efficient than SBES (see Section 7.3).

Inferred MRs. In most subjects, applying the reduction allows us to go from hundreds of MRs to just a few dozens. We obtain no more than 53 MRs (13 on average) per subject, after applying the SAT-based reduction. Our reduction mechanism achieves on average a reduction of approximately 70%. This is significant, taking into account that the automatically inferred MRs are devised to be an input for the engineer, who will examine these to confirm that they capture the intended behavior of the software under analysis.

To assess the precision of MemoRIA in inferring MRs, we analyze the false positive rate, i.e., the ratio of inferred MRs that are in fact invalid, with respect to the total number of inferred MRs. We do so by resorting to our SAT-based property reduction. More precisely, we manually inspect the MRs generated by MemoRIA(sat), identify and remove the invalid properties in the reduced

Table 4. Comparison of metamorphic relations inference between MEMORIA(all) –our approach without including the SAT-based reduction–, MEMORIA(sat) – our approach with the SAT-based reduction enabled– and the SBES technique.

Subject	Time (sec.)			Inferred (#)			Mutation Score (%)		
	MEMORIA (all)	MEMORIA (sat)	SBES	MEMORIA (all)	MEMORIA (sat)	SBES	MEMORIA (all)	MEMORIA (sat)	SBES
MyBoundedStack	39	541	945	217	9	0	58.8	58.8	0
ListItr	47	8,120	1,750	104	53	3	48.5	48.2	0
JDBCResultSet	14	434	-	72	23	-	7.6	5.7	-
NmbrFStrngTknzr	12	47	1,070	151	5	1	33.3	16.7	0
NmbrFStrngTknzr_m	12	37	-	211	5	-	41.8	31.3	-
SMTPProcessor	12	140	-	85	17	-	22.8	21.2	-
SMTPProcessor_h	11	458	-	15	11	-	24.9	23.0	-
SMTPProtocol	13	13	-	3	2	-	7.4	7.4	-
SftpConnection	24	30	2,952	26	7	0	4.1	4.1	0
Signature	48	106	-	172	14	-	21.9	21.0	-
Socket	9	60	-	73	4	-	10.4	8.6	-
StackAr	138	154	1,853	145	13	2	66.8	64.7	0
StringTokenizer	9	10	662	17	1	0	6.9	6.9	0
ToHTMLStream	13	13	-	16	2	-	1.6	1.3	-
ZipOutputStream	12	21	-	73	2	-	4.1	1.6	-
Stack	47	61	7,080	129	41	41	18.0	16.7	50.1
SingleNode	14	14	2,400	0	0	12	0.4	0.4	0.4
MultiNode	14	14	2,475	0	0	12	0	0	0.4
Edge	638	695	6,309	249	41	20	1.3	1.3	0.8
Path	12	545	1,657	90	22	5	4.9	4.4	2.5
Vector2	11	12	2,121	32	9	21	12.9	12.7	54.2
Vector3	12	14	2,416	44	11	22	9.6	9.5	34.9

set. We then proceed to remove these identified invalid properties from the set of all inferred MRs (obtained by MEMORIA(all)), and apply again the SAT-based reduction. We repeat this process until no more false positives are identified. Each discarded MR was manually inspected by two authors, and a third author was involved in case of disagreement, in order to arrive to a decision on the validity or invalidity of an inferred MR.

Notably, MEMORIA produced false positives only for the `StackAr` case. An average of 19 MRs were identified as false positives across the 10 executions, which represent a 13% of the total inferred MRs (145). False positives arise due to the inherent partiality of the generated test suites, leading to invalid properties that no test in the test suite is able to invalidate. For instance, one of the inferred false positives is property `topAndPop()`^{4} = `makeEmpty()`, stating that four consecutive invocations of method `topAndPop()` (which sequentially performs `top()` and `pop()`, and returns null if the stack is empty) is equivalent to `makeEmpty()`. To invalidate this MR, our dynamic analysis process would need to generate a stack with 5 or more elements, which was not the case in our experiments.

After our manual analysis, and in order to mitigate the threat of missing some false positives, we executed Randoop to generate up to 10,000 tests, and checked whether the inferred MRs were falsified or not. In this process, we were not able to detect any additional false positives.

Mutation Score. Even though the SAT-based reduction eliminates in principle only logically redundant MRs, the bug finding ability of all MRs may be greater than that of the reduced set. We compare the corresponding mutation scores, and the results are shown in Table 4 (see columns MemoRIA(all) and MemoRIA(sat)). For 19 of the subjects, the mutation score achieved with the MemoRIA(sat) MRs is similar to the score achieved by MemoRIA(all), with 6 of them maintaining exactly the same score.

For other cases, such as `NumberFormatStringTokenizer`, the mutation score is greatly affected. A manual analysis confirmed that the main reason for this is the *bound* that was used in the SAT-based analysis with Alloy, which marks some MRs as redundant while they are not. Extending the bound in the analysis can improve the results in these cases. Overall, the MemoRIA(all) MRs achieve an average mutation score of 18.5%, while MemoRIA(sat) obtains 16.6%.

7.3 (RQ3) Comparison with SBES

Table 4 shows how MemoRIA compares to SBES [21], in terms of execution time, number of inferred MRs, and performance in terms of mutant killing ability, when using MRs as test oracles. MemoRIA ran successfully on all subjects except for `MultiNode`, while SBES failed on 9 subjects from [20] (these cases are indicated with "-" in the table). The SBES failures are all due to the tool not being able to properly handle some exceptions that the subjects throw at run-time (the stubs generated by the tool did not compile). As mentioned earlier, MemoRIA failed on `MultiNode` because Evo+EPA was unable to generate an EPA for this case.

Time. MemoRIA is more efficient than SBES in most of the subjects. For the subjects in which both tools could be run, SBES took 2 to 3 orders of magnitude more time to complete the analysis. We can thus conclude that even with the SAT-based reduction, MemoRIA in general performs significantly better than the search-based approach implemented in SBES.

Inferred MRs. SBES failed to run on various of the subjects not included in its original dataset, and it identified 6 MRs in total in `ListItr`, `NumberFormatStringTokenizer` and `StackAr`. However, a manual analysis of these MRs confirmed that they are all false positives, i.e., the properties hold in a few executions but are not valid in the general case. An example of this is in the `StackAr` subject, where SBES suggests `isFull()` and `isEmpty()` as possibly equivalent methods. This property clearly holds only when the Stack has elements and is not full, but not in the general case. When comparing the inferred MRs in the subjects for which both tools ran successfully, the techniques generate complementary MRs.

Mutation score. When using the MRs inferred by SBES as test oracles, we detected mutants for 7 subjects out of the 13 supported by the tool. When using the MRs inferred by MemoRIA, we detected at least one mutant for 20 out of the 21 supported subjects. Thus, the MRs inferred by MemoRIA have better mutant killing ability, on the analyzed subjects. On average, SBES MRs achieved a mutation score of 11%, while MemoRIA achieved 16.6% and 18.5%, with and without the SAT-based reduction, respectively.

We also analyzed the impact of the inferred MRs, when used to augment regression assertions, in fault detection, via mutation analysis. Table 5 shows, for each subject, the improvement achieved when incorporating the MRs inferred by SBES and MemoRIA to the regression assertions generated from Evo+EPA. Fig. 7 summarizes these results.

On the one hand, the MRs inferred by SBES allowed us to improve the mutation score of regression assertions in only one case, namely `Vector3`. Coincidentally, this is also the only case in which SBES outperforms MemoRIA. On the other hand, the MRs from MemoRIA improved the mutation

Table 5. Mutation score improvement achieved when incorporating the MRs.

Subject	**Evo+EPA**	**+SBES**	**+MemoRIA(all)**	**+MemoRIA(sat)**
MyBoundedStack	52.9	52.9	70.6	70.6
ListItr	86.2	86.2	86.2	86.2
JDBCResultSet	46.0	46.0	46.7	46.7
NmbrFStrngTknzr	80.9	80.9	85.7	85.7
NmbrFStrngTknzr_m	50.7	50.7	76.1	76.1
SMTPProcessor	28.4	28.4	28.4	28.4
SMTPProcessor_h	19.5	19.5	30.5	29.1
SMTPProtocol	34.2	34.2	38.8	38.8
SftpConnection	99.6	99.6	99.6	99.6
Signature	74.0	74.0	74.0	74.0
Socket	100.0	100.0	100.0	100.0
StackAr	94.1	94.1	94.1	94.1
StringTokenizer	46.7	46.7	47.7	47.7
ToHTMLStream	8.6	8.6	9.2	9.2
ZipOutputStream	23.6	23.6	24.4	23.9
Stack	99.5	99.5	99.8	99.8
SingleNode	1.9	1.9	2.0	2.0
MultiNode	5.7	5.7	5.7	5.7
Edge	0.8	0.8	1.3	1.3
Path	6.6	6.6	7.4	6.8
Vector2	97.0	97.0	97.0	97.0
Vector3	92.4	94.3	92.4	92.4

score of regression assertions in 13 out of the 22 analyzed subjects, with a significant improvement in some subjects such as `MyBoundedStack` and `NumberFormatStringTokenizer_m`.

The regression assertions obtained with Evo+EPA already have a good mutation score, with a median value of 48.7%. When SBES MRs are added, the improvement does not affect the median. On the other hand, when using the MRs inferred by MemoRIA, the median is increased to 59.1% with and without the SAT-based reduction. Overall, these results show that the MRs inferred by MemoRIA can complement the regression assertions generated by Evo+EPA and considerably improve their fault detection capabilities.

8 DISCUSSION

In this section, we will discuss some important aspects of our technique, and specifically of the MRs inferred by MemoRIA.

EPA Reliability. EPAs play a crucial role in the generation of candidate MRs by MemoRIA. Given a target class, MemoRIA employs the Evo+EPA tool to generate the EPA. Evo+EPA uses dynamic analysis to generate an under-approximation of the actual EPA. This under-approximation may not accurately capture the protocol of the class, in the sense that it may be more permissive in the allowed sequencing of methods (i.e., the class protocol) than the actual class. Thus, it may allow for the generation of candidate MRs that do not actually satisfy the protocol of the class, and therefore will necessarily be falsified during the dynamic checking stage of our technique. That

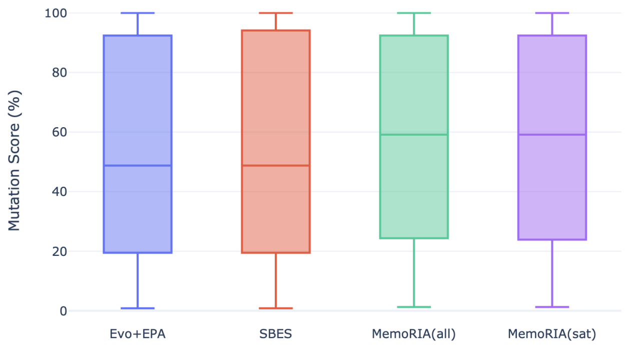


Fig. 7. Mutation score achieved by Evo+EPA regression assertions, and the increase obtained when adding MRs inferred by different techniques.

is, in the presence of an inaccurate EPA, protocol-invalid candidate MRs would be produced, thus reducing the pruning before dynamic checking, and contributing to higher computational costs. An under-approximated EPA, however, cannot reject valid MRs during generation. More precisely, the EPA cannot "leave out" during generation MRs that would "pass" the dynamic checking, assuming that we test these MRs on at least one scenario.

Expressiveness. To a great extent, the ability of MemoRIA in finding useful MRs or MRs that are capable of detecting faults depends on the expressiveness of the MRs language. MemoRIA concentrates on a very particular kind of metamorphic properties: conditional method sequence equivalences. These MRs are suitable for stateful classes, i.e., classes that capture non-trivial object protocols, but may be inadequate or ineffective for other subjects. Our supported pattern of conditional method sequence equivalences has its specific limitations too; in particular, it disregards specific method parameters and conditions on these. Thus, for instance, the `java.util.Stack` valid MR:

$$\{S2\} \Rightarrow \texttt{pop()} = \texttt{remove(size()-1)}$$

which states that performing the pop operation is equivalent to removing the last element, cannot currently be inferred by MemoRIA.

The above observation implies that, compared to manually written MRs, where one is not constrained by a specific MR language or pattern, MemoRIA is inherently less expressive. Also, the expressiveness of our approach can be enhanced by extending and/or adapting the language of candidate MRs, to support other properties of interest.

Soundness. Another relevant aspect is the soundness of the inferred MRs. Regarding this, it is important to remark that MemoRIA infers MRs solely from a program's *actual* (i.e., implemented) behavior, and thus these may not be sound in the sense of capturing the *intended* program behavior. This is an issue common to all specification inference techniques that only consider the subject's implementation as input (e.g., [17, 21, 35–37, 45, 47]). In general, two kinds of incorrect MRs may arise from MemoRIA's inference process: false positives, i.e., relations that are incorrect but are consistent with the tests the program is assessed on; and relations that, due to bugs in the program, are consistent with the program but do not correspond to the intended behavior. In our experiments we analyzed the ratio of incorrect MRs produced by MemoRIA by manually inspecting the produced MRs to identify false positives, and we found that all false positives belonged to the same subject (`StackAr`), where 13% of the inferred MRs were incorrect (see Section 7.2).

Applications. Although we did not assess our technique in the context of some downstream analysis task, it is worth mentioning that tools like MemoRIA, that produce specifications from the program behavior, typically require engineers to manually inspect the inferred assertions (MRs in our case), and discard the invalid ones before applying the resulting specification for a certain analysis task. This manual inspection makes compactness of the inferred specifications critical, which motivates our reduction strategy.

A direct use of MRs inferred by MemoRIA from the subject's implementation is for regression and differential testing. In this scenario, MRs are inferred from a reference implementation (e.g., a functionally acceptable, but suboptimal and/or legacy, implementation), and are then contrasted with future or alternative implementations, in the search of regression or differential bugs. Inferred MRs can support the validation of an implementation too, by allowing a developer to inspect the inferred relations in order to identify reported relations that should not hold, as well as missing relations that are expected to hold.

MRs (and specifications in general) inferred from implementations have many more applications, including fault localization, automated program repair, patch correctness assessment, program comprehension, and as an aid for program verification, among many others [29–31, 40, 41].

9 THREATS AND LIMITATIONS

An important threat to validity is the applicability of MemoRIA to subjects beyond our dataset. We explored the use of MemoRIA, especially in relation to implementation limitations in the EPA generation and the dynamic analysis stages. We inspected more than 200 classes from Defects4J [27], discarding classes with complex dependencies and with inheritance (Evo+EPA does not support them). Out of these, 60 were concrete stateful classes, from which, in principle, an EPA may be computed. In 22 of these, Evo+EPA failed to generate the EPA and/or Randoop was not able to generate tests. We were thus able to run MemoRIA on 38 classes, discovering MRs for 12 classes from projects `cli`, `codec`, `compress`, `lang`, `math`, `time`, `collections` and `jacksoncore`, all from Apache Commons [4]. This provides evidence of the potential increase in the applicability of MemoRIA, through the improvement (or replacement) of some of its components (further details can be found in [2]).

Our comparison was limited to SBES [21], and in particular did not consider the more recent MeMo [8] approach. MeMo requires as input subjects that have appropriate Javadoc comments. None of the subjects from [20] have comments that follow the patterns supported by MeMo, and therefore MeMo cannot be run on these cases. For the subjects from SBES, MeMo achieves around a 10% increase in mutation killing with respect to regression assertions, a similar improvement compared to the one obtained by MemoRIA in our experiments (over a different dataset, see Fig. 7). Overall, the techniques are complementary in applicability (one requires comments, the other classes with object protocols), and they have comparable effectiveness, in their corresponding domains.

10 RELATED WORK

The oracle problem [5] has received significant attention by the software engineering research community. A great progress on automated inference of software specifications has been done recently, including techniques based on dynamic analysis [17, 45], evolutionary computation [37, 47], fuzzing [35], natural language processing [7], and machine learning [36]. These approaches typically execute a test suite of the SUT, observe executions, and infer specifications that are consistent with the observations. MemoRIA differs from these approaches in two aspects. Firstly, these approaches are typically white-box, while MemoRIA is a black-box technique that infers properties in terms of the API, through an abstraction of the SUT (the EPA). Secondly, these approaches infer executable assertions for specific program points, such as pre- and post-conditions, that should hold in every single SUT execution; MemoRIA infers *metamorphic* properties, a weaker but effective kind of oracle, that applies to a large family of executions at once.

A key issue in applying metamorphic testing is the identification of suitable domain-specific metamorphic relations [3, 43]. Carzaniga et al. [9] proposed a variety of user-defined relations, general enough to be applied in many contexts, but ineffective in finding software specific bugs. A recent approach, MeMo [8], aims at gathering likely metamorphic relations from code comments. Though very efficient, its application is limited to mature and well documented software.

Goffi et al. [21] proposed SBES, a dynamic approach that implements a search algorithm that tries to find different method sequences that have the same run-time effect as a given method. SBES is inefficient since it needs to analyze many combinations of SUT methods until it identifies some potential metamorphic relation. As opposed to SBES, MemoRIA discards, prior to any test execution, many invalid candidate metamorphic properties not conforming with the protocol exhibited by the EPA, helping the approach to achieve efficiency. Our experiments confirmed that MemoRIA is more effective and efficient than SBES. At the same time, MemoRIA is limited to some specific properties, conditional method sequence equivalences, and thus it does not support some properties that SBES is able to infer.

11 CONCLUSION

Metamorphic testing is an effective technique that allows us to mitigate the oracle problem. However, the effectiveness of metamorphic testing largely depends on the identification of suitable domain-specific metamorphic relations. This issue has been acknowledged by various researchers, and there currently exist techniques to automatically detect metamorphic relations. However, these either require specific software elements (such as comments) or are expensive due to the combinations of the SUT executions they need analyze. To better deal with the automated discovery of metamorphic relations, we introduced MemoRIA, an approach that automatically generates metamorphic relations that describe the current behavior of a software under analysis in the form of conditional equivalences of method sequence invocations. MemoRIA's approach is based on the use of enabledness-preserving abstractions (EPAs), grammar-based fuzzing, run-time checking and SAT-based analysis. Our experimental evaluation shows that MemoRIA is more efficient compared with the state-of-the-art SBES, and that the obtained metamorphic relations have better fault detection capabilities when used as test oracles, both on their own and when used to complement regression assertions.

DATA AVAILABILITY

All the scripts and data to reproduce our experiments can be found in our replication package [2].

ACKNOWLEDGMENTS

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project under Grant C19/IS/13646587/RASoRS; by Argentina's ANPCyT through grants PICT 2018-3835, 2019-1442, 2019-1973, 2019-2050, 2020-2896, and 2021-4862; by the University of Buenos Aires through grants UBACYT 2020-0233 and 2023-0134; by an Amazon Research Award; by EU's Marie Sklodowska-Curie grant No. 101008233 (MISSION); by the Spanish Government MCIN/AEI/10.13039/501100011033/ERDF through grants TED2021-132464B-I00 (PRODIGY) and PID2022-142290OB-I00 (ESPADA); and by Madrid's regional government program S2018/TCS-4339 (BLOQUES-CM); these three latter projects are co-funded by EU's ESF, EIE, and NextGeneration funds.

REFERENCES

[1] 2023. Fuzzing Research Tools and their Relatedness. https://fuzzing-survey.org/.

[2] 2024. MemoRIA implementation and replication package. https://zenodo.org/records/10683011. https://zenodo.org/records/10683011

[3] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr-Summers. 2021. Testing Web Enabled Simulation at Scale Using Metamorphic Testing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 140–149. https://doi.org/10.1109/ICSE-SEIP52600.2021.00023

[4] Apache. 2023. Apache Commons site. https://commons.apache.org/.

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[6] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. 2011. An Empirical Study of Object Protocols in the Wild. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science, Vol. 6813)*, Mira Mezini (Ed.). Springer, 2–26. https://doi.org/10.1007/978-3-642-22655-7_2

[7] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[8] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/j.jss.2021.111041

[9] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. 2014. Cross-checking oracles from intrinsic software redundancy. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 931–942. https://doi.org/10.1145/2568225.2568287

[10] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. 2013. Automatic recovery from runtime failures. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 782–791. https://doi.org/10.1109/ICSE.2013.6606624

[11] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27. https://doi.org/10.1145/3143561

[12] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *J. Syst. Softw.* 116 (2016), 177–190. https://doi.org/10.1016/j.jss.2015.07.037

[13] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2011. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 381–390. https://doi.org/10.1145/1985793.1985846

[14] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2012. Automated Abstractions for Contract Validation. *IEEE Trans. Software Eng.* 38, 1 (2012), 141–162. https://doi.org/10.1109/TSE.2010.98

[15] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013), 25:1–25:46. https://doi.org/10.1145/2491509.2491519

[16] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013), 25:1–25:46. https://doi.org/10.1145/2491509.2491519

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[18] Javier Godoy et al. 2023. EPA benchmark repository. https://github.com/j-godoy/epa-benchmark.

[19] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE.* ACM, 416–419.

[20] Javier Godoy, Juan Pablo Galeotti, Diego Garbervetsky, and Sebastián Uchitel. 2021. Enabledness-based Testing of Object Protocols. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 12:1–12:36. https://doi.org/10.1145/3415153

[21] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. 2014. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of*

Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 366–376. https://doi.org/10.1145/2635868.2635888

[22] Google. 2021. Google's ClusterFuzz. https://github.com/google/clusterfuzz.

[23] Google. 2023. Google's AFL. https://github.com/google/AFL.

[24] Google. 2023. Google's OSS-Fuzz. https://github.com/google/oss-fuzz.

[25] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis.* MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[26] Samuel Jero, Maria Leonor Pacheco, Dan Goldwasser, and Cristina Nita-Rotaru. 2019. Leveraging Textual Specifications for Grammar-Based Fuzzing of Network Protocols. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 9478–9483. https://doi.org/10.1609/aaai.v33i01.33019478

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[28] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 612–615. https://doi.org/10.1109/ASE.2011.6100138

[29] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 177–188. https://doi.org/10.1145/2931037.2931049

[30] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[31] Thanh Le-Cong, Duc-Minh Luong, Xuan-Bach Dinh Le, David Lo, Nhat-Hoa Tran, Bui Quang Huy, and Quyet-Thang Huynh. 2023. Invalidator: Automated Patch Correctness Assessment Via Semantic and Syntactic Reasoning. *IEEE Trans. Software Eng.* 49, 6 (2023), 3411–3429. https://doi.org/10.1109/TSE.2023.3255177

[32] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? *IEEE Trans. Software Eng.* 40, 1 (2014), 4–22. https://doi.org/10.1109/TSE.2013.46

[33] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[34] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-Time Temporal Logic Guided Greybox Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1343–1355. https://doi.org/10.1145/3510003.3510082

[35] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1008–1020. https://doi.org/10.1145/3510003.3510120

[36] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo F. Frias. 2019. Training binary classifiers as data structure invariants. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 759–770. https://doi.org/10.1109/ICSE.2019.00084

[37] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[38] Oracle. 2023. Documentation of List Iterator implementation from java.util. https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html.

[39] Oracle. 2023. Documentation of Stack implementation from java.util. https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html.

[40] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Trans. Software Eng.* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[41] Todd W. Schiller and Michael D. Ernst. 2012. Reducing the barriers to writing verified specifications. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer

(Eds.). ACM, 95–112. https://doi.org/10.1145/2384616.2384624

[42] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. 2022. Automated Generation of Metamorphic Relations for Query-Based Systems. In *IEEE/ACM 7th International Workshop on Metamorphic Testing, MET@ICSE 2022, Pittsburgh, PA, USA, May 9, 2022*. ACM, 48–55. https://doi.org/10.1145/3524846.3527338

[43] Sergio Segura, Gordon Fraser, Ana Belén 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

[44] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. 2020. Metamorphic Testing: Testing the Untestable. *IEEE Softw.* 37, 3 (2020), 46–53. https://doi.org/10.1109/MS.2018.2875968

[45] Anthony J. H. Simons. 2007. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Autom. Softw. Eng.* 14, 4 (2007), 369–418. https://doi.org/10.1007/s10515-007-0015-3

[46] Graphstream Team. 2023. Graphstream project site. https://graphstream-project.org/.

[47] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. https://doi.org/10.1145/3368089.3409758

[48] Javier Troya, Sergio Segura, and Antonio Ruiz Cortés. 2018. Automated inference of likely metamorphic relations for model transformations. *J. Syst. Softw.* 136 (2018), 188–208. https://doi.org/10.1016/j.jss.2017.05.043

[49] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/ Retrieved 2019-09-09 16:42:54+02:00.

[50] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 235–245. https://doi.org/10.1109/ICSME.2019.00035

[51] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-Based Inference of Polynomial Metamorphic Relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 701–712. https://doi.org/10.1145/2642937.2642994

Received 2023-09-29; accepted 2024-01-23