Comprehending Object State via Dynamic Class Invariant Learning*

Jan H. Boockmann(✉) and Gerald Lüttgen

Software Technologies Research Group,
University of Bamberg, Bamberg, Germany
{jan.boockmann,gerald.luettgen}@swt-bamberg.de

Abstract. Maintaining software is cumbersome when method argument constraints are undocumented. To reveal them, previous work learned preconditions from exemplary valid and invalid method arguments. In practice, it would be highly beneficial to know class invariants, too, because functionality added during software maintenance must not break them. Even more so than method preconditions, class invariants are rarely documented and often cannot completely be inferred automatically, especially for objects exhibiting complex state such as dynamic data structures.

This paper presents a novel dynamic approach to learning class invariants, thereby complementing related work on learning method preconditions. We automatically synthesize assertions from an adjustable assertion grammar to distinguish valid and invalid objects. While random walks generate valid objects, a combination of bounded-exhaustive testing techniques and behavioral oracles yield invalid objects. The utility of our approach for code comprehension and software maintenance is demonstrated by comparing our learned invariants to documented invariant validation methods found in real-world Java classes and to the invariants detected by the Daikon tool.

1 Introduction

Comprehending the behavior of a complex software component is challenging, but necessary for component reuse and maintenance. The object-oriented programming paradigm has enforced the principle of information hiding, which separates externally observable behavior from internal implementation. To make a component *reusable*, it typically suffices to document its external behavior and the constraints imposed on its method argument values. When following the principles of defensive programming [4], a thorough input validation at the entry of each method checks whether the constraints are satisfied. For components that lack input validation, previous work has shown that appropriate preconditions can be inferred automatically [2,8,27,30,33].

* This research is supported by the German Research Foundation (DFG) under project DSI2 (grant no. LU 1748/4-2).

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 143–164, 2024.
https://doi.org/10.1007/978-3-031-57259-3_7

To make a component *maintainable*, however, information on its external behavior alone is insufficient, because maintenance may require modifications of the component's implementation. *Class invariants* [19,20] capturing the constraints on the component's program state exhibited at runtime are essential for maintainers to ensure that their source code modifications, such as bug fixing, refactoring, or implementing new functionalities, match the assumptions implicitly encoded in the existing source code. A failure to do so may result in unpredictable behavior or even system crashes. Despite this, class invariants are rarely documented and checked even more rarely during input validation.

Approaches to dynamic assertion learning generalize from observations, e.g., object states, to synthesize assertions such as preconditions and class invariants. Related tools include *Daikon* [8], *Proviso* [2], *Hanoi* [22], and *EvoSpex* [25]. Daikon observes program states during execution and uses templates to obtain a set of candidate assertions, including class invariants, that hold at certain program locations. Proviso learns preconditions that also consider complex data types and uses a test generator as an oracle to detect invalid method arguments. Hanoi infers representation invariants for data types in a functional programming language. EvoSpex employs an evolutionary algorithm to learn postconditions from (in)valid pre/post state pairs. Overall, the exploration of approaches to dynamic class invariant learning for complex types remains relatively limited, despite the potential benefits for software maintenance.

This paper proposes a dynamic analysis approach that learns a class invariant using iterative refinements from (in)valid objects. We perform random walks in object state spaces to construct valid objects and combine bounded-exhaustive testing techniques [3,6,18] with behavioral oracles to create invalid objects. As oracles, one can either adapt the random walks or provide property-based tests [9]. We refine our candidate invariant by removing existing or introducing new assertions, which are dynamically constructed along an assertion grammar. This process iterates until all obtained (in)valid objects are classified correctly.

We have implemented our class invariant learning approach for Java in a prototype tool, called *Geminus*. Our evaluation shows, for real-world Java classes taken primarily from the the `java.util` package, that our learned class invariants are at least as accurate as, and often surpass, those detected by Daikon or documented in the code. Beyond software maintenance, class invariants also support various software development activities, including software testing [13].

Organization Section 2 introduces the notions of class invariant and bounded-exhaustive/property-based testing alongside a running example. Section 3 explains our class invariant learning approach and Section 4 evaluates it. Section 5 discusses related work, while Section 6 presents our conclusions and future work.

2 Foundations

This section reviews the concepts of class invariant in the context of the object-oriented paradigm by means of a running example. We subsequently outline how property-based and bounded-exhaustive testing relate to class invariants.

```
1 public class SimpleSquare {
2   //@ invariant w == h && w > 0;
3   private int w, h; // width and height
4
5   public SimpleSquare() { setLength(1); }
6   public void setLength(int length) {
7     if (length <= 0) { throw new IllegalArgumentException(); }
8     this.w = length;
9     this.h = length;
10  }
11
12  public int area() { return w*h; }
13  public int perimeter() { return 2*(w+h); }
14  public int aspectRatio() { return w/h; }
15
16  public SimpleRectangle toRect() {
17    return new SimpleRectangle(w, h);
18  }
19 }
```

Fig. 1: Running example Java class `SimpleSquare`.

Running Example The class `SimpleSquare` in Figure 1 models a square with a non-zero positive length using the two integer attributes *width* (`w`) and *height* (`h`). Other objects can interact with `SimpleSquare` by invoking its public methods to set the length of the square or to compute its geometric properties, or to obtain an equivalent object of class `SimpleRectangle`. Note that method `setLength` performs thorough input validation and throws an `IllegalArgumentException` if the provided method argument value is not strictly positive.

Class Invariants Objects play a fundamental role in object-oriented programming. They are created via constructors, interact with other objects via method calls, and are disposed by a destructor. Throughout method execution, an object may call methods of other objects, including itself, or alter the accessible attributes of other objects. Often, invoking a method results in a side-effect or modification of the object's state, either through modifying its primitive attributes or by modifying the object state of a referenced object.

The notion of a class invariant in object-oriented programming has first been explored in [19] and since been adapted by specification languages such as JML [16]. Understanding class invariants is crucial during development and maintenance, because they provide guarantees about the object state at the start of a *qualified method call* [20] and the end of such a call. In contrast, the class invariant may not hold for *unqualified method calls*, which the object invokes on itself. For example, calling *setLength* in the constructor is considered unqualified. Accordingly, the class invariant holds for all objects derived via a constructor or via a qualified call invoked on an object that satisfies the invariant.

```
1 @Test public void traditionalTest() {
2   SimpleSquare s = new SimpleSquare();
3   s.setLength(5);
4   assert s.area() == 25;
5 }
6
7 @Test public void propertyBasedTest(SimpleSquare s) {
8   assert s.toRect().area() == s.area();
9   s.toRect().toSquare(); // implicitly checks absence of exception
10 }
```

Fig. 2: A traditional and a property-based tests for class `SimpleSquare`.

In the running example, the assertion that the width and height are equal and strictly positive is a suitable class invariant. Accordingly, method `aspectRatio` does not need to check that attribute `h` is non-zero to avoid a division-by-zero exception, because this is implied by the invariant. Similarly, method `toRect` can assume that constructing a new `SimpleRectangle` object always succeeds.

The set of reachable objects that a class invariant has to satisfy can be constructed incrementally by performing random walks in the object state space. A random walk starts at an object state derived from a constructor and continues by invoking methods on the current object; this kind of state exploration is used in the context of fuzz testing [17] and test suite generation [10,26]. Even for finite object state spaces, an exhaustive exploration is often practically infeasible.

Property-Based Testing While traditional tests first establish a testing scenario, property-based tests [9] are parameterized over inputs supplied by a test engine. Property-based testing is primarily used in functional languages, e.g., in Haskell using *QuickCheck* [5], but can also be applied to object-oriented programs.

Figure 2 depicts a traditional and a property-based test for our running example. Note that the property-based test is parameterized over an object of the class under test and checks that the obtained rectangle has the same area as the former square. It also implicitly tests that the translation from rectangle to square via method `toSquare` does not raise an exception.

Bounded-Exhaustive Testing Deriving a representative set of objects, e.g., for property-based testing, is often a tedious and error-prone task when done manually. Bounded-exhaustive testing [6,11,21] is a testing technique that automatically tests a software for all valid inputs within specified size bounds.

While primitive types like integers are often sampled from a range of values, complex object states usually require a create-and-test approach: a systematic enumeration artificially assigns values to private and public attributes to create all object states within a provided bound, and a manually specified predicate, i.e., a class invariant, tests for validity and retains valid objects only.

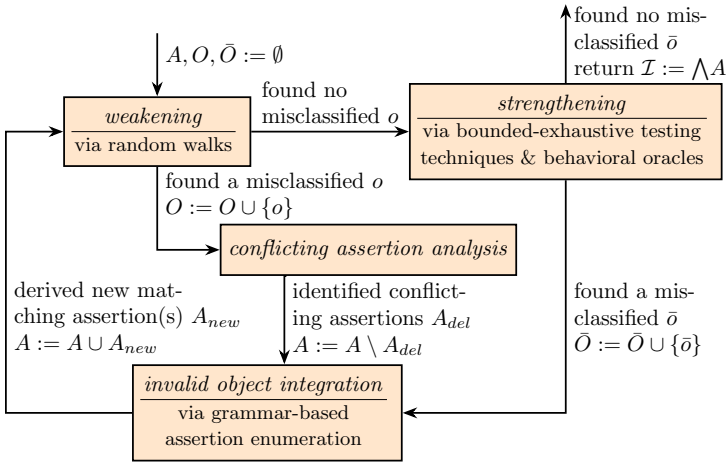

Fig. 3: Overview of our approach to dynamic class invariant learning.

3 Approach

This section introduces our approach to dynamic class invariant learning, which is depicted in Figure 3. Each step either modifies the set of collected valid (O) or invalid ($\bar{O}$) objects, or the set of assertions (A) whose conjunction forms the candidate class invariant ($\mathcal{I}$). If an object is reachable, we consider it valid. If an object is unreachable, we consider it invalid. The class invariant we aim to learn classifies all reachable objects as valid and all unreachable objects as invalid.

The *weakening* step aims to refine the candidate class invariant $\mathcal{I}$ by finding a valid object o that is classified as invalid by $\mathcal{I}$. If successful, we remove the conflicting, overly restrictive assertion(s) that caused the incorrect classification. Previously collected invalid objects that are no longer classified as invalid due to the removed assertions are reintegrated subsequently. If no valid object is misclassified, we perform *strengthening* to find an invalid object $\bar{o}$ that is misclassified. The *invalid object integration* step then derives a matching assertion that correctly classifies an invalid object as invalid but all prior found valid objects still as valid. If no $\bar{o}$ is found, we return the candidate class invariant.

Because our approach learns from a finite set of objects, the learned class invariant is only correct for the collected (in)valid objects, but not in general. However, if no assertion can be generated to distinguish a valid from an invalid object, the learned invariant correctly classifies only all identified valid objects, but mistakenly classifies some invalid objects as valid.

The high-level weakening, strengthening, and invalid object integration steps are generic and can be instantiated by different techniques. Our approach leverages random walks to generate valid objects and combines bounded-exhaustive testing techniques with behavioral oracles to obtain invalid objects. We derive assertions to distinguish valid from invalid objects using a grammar. In contrast to related approaches [25,30], our objects are guaranteed to be (in)valid.

Table 1: Intermediate states of our approach to class invariant learning in each iteration, for the `SimpleSquare` running example.

it.	current assertions A	found new $o/\bar{o}$	removed assertions A_{del}	new assertions A_{new}
1	$\emptyset$	$\bar{o}$: ⌈0 0⌉	$\emptyset$	$\{false\}$
2	$\{false\}$	o : 1 1	$\{false\}$	$\{w = 1\}$
3	$\{w = 1\}$	$\bar{o}$: ⌈1 0⌉	$\emptyset$	$\{w = h\}$
4	$\{w = 1, w = h\}$	o : 2 2	$\{w = 1\}$	$\{w > 0\}$
5	$\{w = h, w > 0\}$	$\bot$		

Table 1 shows the execution state of our approach in each iteration when learning class invariant $w = h \wedge w > 0$ for our running example `SimpleSquare`. Valid objects such as 1 1 are indicated by a solid box, while invalid objects such as ⌈0 0⌉ are shown in a dashed box. The remainder of this section uses this example to illustrate the workings of our invariant learning approach.

3.1 A Triangle of Oracles

Our approach exploits the insight that an executable implementation, a testable assumption, and an object form a closed loop of information. Assuming two elements are correct one to allows constructing a test-based oracle to assess the correctness of the third. This leads to the creation of three distinct oracles:

1. *Implementation*: Given a correct assumption and a valid object, any failure upon testing the assumption indicates a faulty implementation.
2. *Assumption*: Given a correct implementation and a valid object, any failure upon testing the assumption indicates an incorrect assumption.
3. *Object*: Given a correct implementation and a correct assumption, any failure upon testing the assumption indicates an invalid object.

The implementation oracle is leveraged in software testing to detect faulty implementations. It either encodes assumptions as traditional tests, which create objects assumed to be valid by construction and checks assertions on them, or as property-based tests, which evaluate properties on valid objects supplied by the test engine. When learning a class invariant for a given implementation, one can ignore the question of implementation correctness, because the invariant is supposed to reflect the implementation. However, a learned invariant that does not match the expectations may indicate a faulty implementation.

The assumption oracle can be employed to identify an incorrect invariant that misclassifies valid objects as invalid when considering the invariant as the assumption. By generating valid objects in our weakening step, we detect an overly restrictive, i.e., unsound, invariant. Analogously, the second oracle can

be used to identify invariants that misclassify invalid objects as valid. If an object is invalid, but the candidate invariant holds, the invariant is incomplete, which allows our strengthening step to detect overly permissive invariants. We consider an invariant/oracle *sound* if it classifies all valid objects as valid, and *complete* if it classifies all invalid objects as invalid. The objects revealing an incorrect candidate class invariant are added to the training set during weakening/strengthening, and the invariant is updated accordingly.

The object oracle can detect invalid objects if implementation and assumption are correct. Invalid objects can be used by the assumption oracle to spot overly permissive invariants. Providing assumptions to detect both valid and invalid objects is challenging and equivalent to learning the class invariant.

3.2 Generating Valid Object States via Random Walks

The weakening step leverages the assumption oracle to assess whether the candidate class invariant misclassifies valid objects as invalid. To construct valid objects, we perform random walks in object state spaces: any object derived via a sequence of qualified method calls starting from a freshly constructed object is valid. Because the implementation can be considered correct, a method invocation in a random walk may only throw *expected exceptions*, which are associated with a failed input validation such as the `IllegalArgumentException` thrown by method `setLength`. In contrast, *unexpected exceptions* are prevented by the class invariant. For example, a division-by-zero exception cannot be thrown in method `aspectRatio`, because the invariant guarantees that the height is non-zero. In practice, all checked exceptions in Java are typically expected exceptions and some unchecked exceptions are unexpected exceptions.

We parameterize the random walks using a set of *builders* and *actions*. Builders construct fresh objects using the available constructors, and actions invoke methods. Following the naming convention of [31] for methods, we use the term *observer/modifier* action to denote an action that does not/does alter the considered object's state. In our example, a single builder invoking the zero-argument constructor and a single action invoking method `setLength` with value 2 suffice. To enforce termination, we bound the random walk with respect to the number of walks and the number of method calls per walk. To ensure deterministic behavior, one may either randomly select a builder/action using a fixed seed (like Randoop [26]) or exhaustively explore all builder/action combinations up to a given depth (like EvoSpex [25]). Thus, not finding a valid object that is misclassified as invalid by the candidate class invariant does not guarantee the absence of one. The effectiveness of finding a misclassified object depends on the object state coverage achieved by the random walk.

The candidate invariant before the second iteration (*false*) in Table 1 misclassifies `1 1` obtained directly from the constructor. In contrast, the invariant at the start of the fourth iteration ($w = 1 \wedge w = h$) misclassifies `2 2`, which is obtained after invoking `setLength(2)` on the freshly constructed object. No valid object is misclassified as invalid for the invariant at the start of the fifth

Table 2: Accuracy of properties for detecting artificially created invalid `SimpleSquare` objects (• detected, ◦ undetected)

invoked method/tested property	[0 0]	[1 0]	[-1 -1]	[1 2]	[3 2]
`aspectRatio()`	•	•	◦	◦	◦
`toRect()`	•	•	•	◦	◦
`area()>0`	•	•	◦	◦	◦
`perimeter()>0`	•	◦	•	◦	◦
`aspectRatio()==1`	•	•	◦	•	◦
`toRect().toSquare()`	•	•	•	•	•

iteration ($w = h \wedge w > 0$). Hence, this invariant is sound and, as we will see later, it is also complete.

3.3 Detecting Invalid Objects via Behavioral Oracles

An object is considered invalid if it cannot be reached via a random walk. However, exhaustive state space exploration is impossible for infinite state spaces which occur, e.g., when objects use references to establish unbounded structures such as linked lists. Even for finite state spaces as exhibited by the running example, an exhaustive exploration often remains practically infeasible. In general, a partial exploration does not provide a sound oracle to determine if a supplied object is unreachable. To detect invalid objects, we instead consider *behavioral oracles* that exploit the behavior of the object under analysis exposed upon method invocations. We consider two sound but possibly incomplete behavioral oracles for detecting invalid objects: random walks and property-based tests.

Random Walks as Weak Oracles During the random walks used to generate valid objects, any thrown expected exception indicates a failed input validation and is ignored. Conversely, if an unexpected exception occurs during a walk starting from an artificially created object, it implies that all objects along the walk, including the initial object, are invalid. The use of random walks for detecting invalid objects shares similarities with *fuzz testing* [17] for identifying faulty implementations. In fuzz testing, a program is subjected to a range of different input values to cause an observable error [38], indicating a bug in the implementation. For a correct implementation, any unexpected exception indicates an invalid object. While behavioral oracles based on random walk-based are sound by construction for detecting invalid objects, they are rarely complete.

Table 2 shows the detection results of six properties for five invalid objects. The first two properties resemble observer actions during a random walk. Method `aspectRatio` throws a division-by-zero exception if the height is zero, thus detecting the first two invalid objects. Method `toRect` creates a new rectangle with the same width and height as the current square. The constructor of class

`SimpleRectangle` (not shown) validates the input width and height and throws an exception if argument values are not strictly positive, thus subsuming the `aspectRatio` method in terms of its detection capabilities. However, it fails to detect objects whose strictly positive width and height differ.

Property-based Tests as Strong Oracles Property-based tests [9] are a stronger behavioral oracle when compared to random walks. Not only can they detect invalid objects that throw unexpected exceptions, but they can also interpret the absence of an exception and method return values as an indication of object invalidity. Because property-based tests operate at a behavioral level, they do not require knowledge about internal implementation details. Information regarding expected behavior can be found in the documentation of the class under analysis and (formal) specifications, e.g., for abstract data types [12]. Because property-based tests are assumed to be sound but incomplete, a passing property-based test suite does not guarantee the validity of the object under analysis. However, a single failed test is sufficient to deem the object invalid.

The last four properties in Table 2 resemble candidate property-based tests. We may assume that the expected behavior of class `SimpleSquare` is that the area and the perimeter must be greater than zero and that the aspect ratio must be equal to one. In addition, the translation from a square to a rectangle and back to a square should be possible without raising an exception. Observe that the area property detects invalid objects with either the width, height or both equal to zero. The perimeter property detects those invalid objects where the sum of width and height is not strictly positive. Note that the aspect ratio property, in addition to its corresponding observer action, detects some states (due to integer division) where `w` and `h` differ. The last property subsumes its associated observer action and detects all invalid objects.

3.4 Generating Invalid Objects via Bounded-Exhaustive Testing Techniques

By considering invalid objects, we can not only check if the invariant is complete, i.e., sufficiently restrictive, but also automatically identify equivalent assertions [1,28]. While misclassified valid objects found during weakening widen the scope, misclassified invalid object found during strengthening narrow it.

Acquiring a representative set of invalid objects is a non-trivial task. Existing assertion learning approaches primarily derive possibly invalid objects by executing a mutated program [15,23,30] or by mutating valid program states [25,29]. Nevertheless, these approaches often assume the derived object state to be invalid without conducting further validation. Consequently, the quality of the learned assertion is compromised if a valid object state is mistakenly labeled as invalid. Using generators for complex test inputs from *bounded-exhaustive testing* (BET), such as *Korat* [3,21], enables the artificial creation of a large number of (in)valid object states. We combine these generators with behavioral oracles, and contrary to the conventional practice in BET of retaining only valid objects, we retain only those objects that are classified as invalid. Behavioral oracles can

also be applied to objects constructed using program or state mutation; however, we favor the complex test input generators from BET because they produce a larger and more representative set of invalid objects.

The five invalid object states displayed in Table 2 are included in the output of a bounded-exhaustive object state generator when supplied with a lower/upper bound of -1/3 on integer values. The invalid objects [0 _ _ 0] and [1 _ _ 0] are suitable for strengthening the candidate invariant.

3.5 Invalid Object Integration

Our approach generates new assertions on-the-fly in order to integrate so far misclassified invalid objects and classify them correctly. Each assertion is evaluated in the context of an object of the class under study. The following assertion grammar suffices for our running example:

Int	::=	*0* | *1*
Bool	::=	*true* | *false* | **Int** = **Int** | **Int** > **Int**
Int	::=$^+$	*w* | *h*

The first two rule fragments reason about integer and boolean values, while the last fragment provides access to the attributes of a `SimpleSquare` object. Terminals such as "*1*" or ">" denote constants or operators, and non-terminals such as **Int** are types. Symbol ::=$^+$ indicates that we supplement a non-terminal with new rules.

The invalid object integration step is performed after strengthening or weakening. In the former case, a single invalid object is provided, while in the latter case there may be multiple or no invalid objects. In case of a single misclassified invalid object, we search for an assertion that classifies the said object as invalid, but does not classify any previously collected valid object as invalid. For multiple invalid objects, we iteratively search for a suitable assertion.

Our invalid object integration step can be substituted with any model learning approach that accepts valid and invalid object states as input. While neural networks [24] and support vector machines [30] generally achieve high accuracy, their black-box nature makes them less ideal for program comprehension. In contrast, decision tree models [2] offer interpretability, but their internal disjunctive encoding is disparate to how developers express class invariants in code, usually as a sequence of assert statements. Hence, we favor conjunctive models for modeling class invariants in the context of comprehending object states, because they are interpretable and align with how invariants are phrased in practice.

Caching Suitable Assertions An unsuitable assertion either incorrectly detects a valid object or does not detect the candidate invalid object. Because our approach only adds objects and never removes existing ones, an assertion that incorrectly detects a valid object is not only unsuitable to integrate the currently misclassified invalid object but also for any future one. In contrast, an

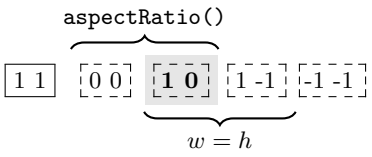

Fig. 4: The behavioral oracle `aspectRatio()` and the assertion $w = h$ both detect the invalid object [1 _ _ 0], but classify other objects differently.

assertion that satisfies all valid objects and the misclassified invalid object may still be suitable in the future.

Our caching mechanism only stores assertions that satisfy all valid objects. For example, after observing [1 1] we store the assertion *true* in the cache, but we do not store *false*.

Preventing Equivalent Assertions Our approach only adds assertions to distinguish invalid from valid objects, which prevents the generation of equivalent assertions. This strategy exploits *observational equivalence* [1,28], which creates equivalence partitions among assertions based on the values to which they evaluate. Because our approach only adds an assertion if the existing assertions cannot distinguish an invalid object from the valid objects, the added assertion is observationally inequivalent to any existing assertion. This property remains true because we only add (in)valid objects, thus refining this notion of equivalence. For example, *false* and *w=1* are considered to be equivalent with respect to [0 _ _ 0], but are inequivalent when also considering [1 1].

Observational equivalence cannot be used for approaches that only consider valid objects [8,27,34], because all suitable assertions are deemed equivalent. Instead, these approaches require static analysis to detect equivalent assertions.

Inexpressive Assertion Grammars If the assertion grammar for the example in Figure 4 would only be capable of generating the assertion $w = h$, then the invalid object [0 _ _ 0] cannot be integrated. This invalid object is said to be *indistinguishable* from the valid objects such as [1 1] with respect to the employed assertion grammar. Because our collected objects are proven (in)valid, indistinguishability can only be resolved by increasing the grammar's expressiveness. Instead, we continue learning but label the class invariant as *approximate*, which ensures that it is overly permissive and, thus, remains sound. Note that once the candidate class invariant becomes approximate, it remains so. However, an overly permissive invariant is still useful for program comprehension, because a subsequent manual invariant refinement only needs to add assertions.

Outperforming the Behavioral Oracle Our approach does not learn an invariant from a single complete oracle, utilizes two sources of sound information: behavioral oracles for invalid objects and random walks for valid objects. This can result in invariants that improve upon the accuracy of the underlying behavioral

oracle. For example, the oracle `aspectRatio()` in Figure 4 detects the invalid object [1 0], which can be integrated by adding the assertion $w = h$ to the candidate class invariant. Note that this assertion also detects the invalid object [1 -1] that is not detected by the oracle.

Qualities of Learned Class Invariants The quality of our learned class invariants depends on the expressiveness of the assertion grammar, the accuracy of the behavioral oracle, and the object state coverage achieved by the random walk for generating valid objects and the bounded-exhaustive object state generator for generating potential invalid objects. While an inexpressive assertion grammar may be detected during learning, an incomplete oracle or an insufficient object state coverage cannot be detected. Accordingly, no soundness/completeness guarantees can be given for a learned non-approximate class invariant except that it correctly classifies all collected (in)valid objects. Approximate class invariants classify some of the collected invalid objects as valid, which still aids comprehension in the presence of an inexpressive assertion grammar.

Learning a complete invariant that also correctly classifies so far unseen objects is only possible if the assertion grammar is sufficiently expressive, the behavioral oracle is complete, and the object state coverage is sufficient, e.g., exhaustive for finite object state spaces.

4 Evaluation

To evaluate our class invariant learning approach, we have implemented the prototype tool *Geminus* for Java. Our bounded-exhaustive object state generator uses the Java Reflection API to modify the internal object state and prevents the generation of symmetric object states in the style of [21]. Our grammar-based assertion generator performs an explicit top-down enumeration and generates strings representing native Java expressions, which allows for a simple grammar definition. We use the Java JShell to dynamically compile these strings into executable lambda expressions at runtime.

Our experiments focus on the following research questions:

RQ1 How do random walks and property-based tests compare to a ground-truth class invariant in terms of detecting invalid objects?

RQ2 What is the disparity between the class invariant learned by Geminus and the employed behavioral oracle?

RQ3 How does the accuracy of the class invariant(s) learned by Geminus, detected by Daikon, and documented as invariant validation methods differ?

4.1 Benchmark Composition

Our benchmark contains several dynamic data structures, whose implementations exhibit complex invariants. In addition, the corresponding classes are one of the few in the Java collections framework that contain state validation methods.

From the evaluation examples of Daikon [8], we pick `StackAr` and `QueueAr`, which were adapted from [37] and provide an array-based implementation of a stack and queue, respectively. The majority of our dynamic data structures originate from the Java collections framework `java.util`. Class `ArrayList` and legacy class `Vector` both provide a linear collection via an array-based implementation. In addition, class `LinkedList` provides `Deque`/`Queue` functionalities via a linkage-based implementation, while class `ArrayDeque` uses an array-based implementation. Class `PriorityQueue` handles comparable elements via an array-based priority heap, and class `BitSet` offers a memory-efficient bit vector.

For verification, a class invariant needs to be strong enough to prove an assertion. In our learning setting, we search for a class invariant that correctly classifies all reachable objects as valid and all unreachable objects as invalid. Depending on the verification task, the class invariant required for this may be weaker than the invariant we aim to learn. Accordingly, the manually specified ground-truth invariants for evaluating each benchmark item must be as strong as possible. Thus, the number of benchmark items is primarily limited by the cost of manually specifying these strong class invariants. Evaluating our approach on further data structures, including *Maps* and *Sets*, is left for future work.

To evaluate our approach, we have instantiated a random walk and bounded-exhaustive generator for each benchmark item and have written property-based tests using the provided documentation. We configure the assertion grammar to include binary operators among integers (`+`, `-`, `==`, `!=`, `>=`, `>`), object identity, range null checks in arrays, and the ternary operator (`c?b:true`) to encode implications. Extending the grammar with additional operators, such as multiplication or division among integers, is straightforward and may improve the expressiveness of the grammar. However, the increase of assertions expressible in the grammar may lead to timeouts during assertion synthesis. For our experiments, we limit assertion generation to a maximum of 75 000 assertions.

4.2 Evaluation Results

Our results in Table 3 show the number of valid (val.) and invalid (inv.) objects produced by the bounded-exhaustive generator for our ground-truth invariant, which contains A assertions. Because random walks (RW) and property-based tests (PBT) are sound, i.e., all objects classified as invalid are guaranteed to be invalid, we only report false-negatives (FN), i.e., the number of invalid objects that remain undetected. As a behavioral oracle, our random walks have a walk length and a walk count of 50. Increasing the walk length and count may improve detection accuracy, but at the cost of increased computation time.

Our evaluation results in Table 4 report on the accuracy of the class invariant learned by Geminus using random walks or property-based tests as oracle, the class invariant detected by Daikon in its default configuration, and the invariant validation method documented in the source code (Doc). Geminus and Daikon receive the same set of valid objects derived from deterministic random walks with both a walk length and a walk count of 500, respectively. Analogously to using random walks as oracles, increasing the walk length and count may

Table 3: Accuracy comparison in detecting invalid objects using manually written ground-truth class invariants, random walks, and property-based tests; best results are highlighted in bold.

Item	Ground-truth			RW	PBT
	val.	inv.	A	FN	FN
`SimpleSquare`	10	431	2	90	**0**
`StackAr`	4097	4095	3	**0**	**0**
`QueueAr`	322	10 678	13	3 078	**152**
`PriorityQueue`	1918	154 954	8	63 149	**36 708**
`BitSet`	2047	40 961	6	19 099	**18 434**
`ArrayList`	4083	38 925	4	**16 398**	**16 398**
`Vector`	4083	38 925	4	**16 398**	**16 398**
`LinkedList`	4	38 335	4	4	**0**
`ArrayDeque`	385	345 727	12	169 593	**0**

further improve the object state space coverage in terms of valid objects, but at the cost of increased computation time. In addition, Geminus derives invalid objects from the bounded-exhaustive object state generator using its respective oracle. We only report false-positives (FP) for Daikon, because the invariants learned by Geminus classify all valid object as valid in our experiments. We report the computation time (t) in seconds. All experiments were conducted on an Apple MacBook Air M2 with 16 GB RAM.

Regarding threats to validity, we manually examined the source code of the benchmark items to define the ground-truth class invariant. To mitigate the risk of specifying an overly restrictive invariant, we validated it against the objects visited by our random walk. To address threats to internal validity that may arise from random walks, we fixed the random number generator's seed to ensure that the same objects are generated during each walk. Furthermore, we excluded probabilistic data structures like *skip lists* [32] from the benchmark to ensure identical internal object states.

4.3 Oracle Accuracy Comparison

When used as a behavioral oracle, random walks detect numerous invalid object states in our experiments. They exhibit comparable accuracy to property-based tests for benchmark items `StackAr`, `ArrayList`, and `Vector`. Additionally, random walks identify a significant portion of invalid objects for `LinkedList`. The majority of unexpected exceptions arise from null dereferencing or accessing out-of-bounds indices in arrays. Random walks cannot assess whether the retrieved elements from a `PriorityQueue` are in the correct order. The documentation states that retrieving the first element from an `ArrayDeque` throws an exception

Table 4: Comparing the accuracy in detecting invalid objects using the class invariant learned by Geminus, detected by Daikon, and invariant validation methods documented in the code; best results are highlighted in bold.

Item	Geminus+RW					Geminus+PBT					Daikon				Doc	
	FN	t	A	O	$\bar{O}$	FN	t	A	O	$\bar{O}$	FN	FP	t	A	FN	A
`SimpleSquare`	**0**	3	2	2	2	**0**	4	2	1	2	1	0	7	2	–	–
`StackAr`	**0**	6	2	3	4	**0**	5	2	3	4	**0**	0	7	4	–	–
`QueueAr`	2 229	7	4	13	12	**542**	93	15	39	50	2 513	0	9	9	–	–
`PriorityQueue`	62 545	31	2	4	5	**9 277**	298	3	11	11	112 462	0	32	6	–	–
`BitSet`	18 434	11	2	3	4	18 434	9	2	3	4	55	2036	45	3	**0**	3
`ArrayList`	16 398	10	2	3	4	16 398	10	2	3	4	16 398	0	53	3	**7 181**	2
`Vector`	16 398	21	2	3	4	16 398	21	2	3	4	16 398	0	49	4	**7 181**	2
`LinkedList`	**0**	15	10	4	29	**0**	15	10	4	29	0	4	26	16	729	1
`LinkedList*`	**0**	10	2	3	2	**0**	10	2	3	2						
`ArrayDeque`	98 966	74	5	6	9	**0**	60	8	23	24	169 593	0	23	7	30 079	7

if the structure is empty, but random walks cannot detect cases where the queue is considered empty, yet a retrieval does not throw an exception.

The property-based tests fail to identify some invalid objects for five items. `BitSet`, `ArrayList`, and `Vector` implementations nullify unused array elements to aid garbage collection, which does not affect functional behavior. However, our tests, which focus on functional behavior, cannot detect objects violating this property. Random walks can also only uncover faults related to functional behavior. In the case of `StackAr`, where the ground-truth class invariant is limited to functional aspects only, both our tests and the random walks detect all invalid objects. For `PriorityQueue`, polling the first element involves a sift-down operation, partially repairing an invalid object state. In contrast, a `QueueAr` with a capacity of zero is considered both empty and full simultaneously, leading any method to return immediately, and concealing the remaining state. This is a known debugging scenario [38], where a bug can lead to an invalid object state without necessarily causing an observable error.

Regarding **RQ1**, our benchmark in Table 3 leads to the conclusion that property-based tests outperform random walks in terms of accuracy. Furthermore, we observed that the remaining undetected invalid objects either do not affect functional behavior or are partially repaired during method invocation, rendering their detection challenging.

4.4 Disparity between Learned Invariants and Leveraged Oracles

Using random walks as behavioral oracles, Geminus learns and often surpasses the accuracy of the oracles in our experiments. Although our random walks do

not detect all invalid objects for class `SimpleSquare` (see Table 2), Geminus still manages to learn the correct class invariant. The accuracy of the learned class invariant depends on the assertion grammar and the order in which candidate assertions are generated. For `SimpleSquare`, assertions $w = h$ and $w > 0$ are generated before assertions $w \geq 1$ and $h \geq 1$, which would also resolve all misclassified objects found by the random walk oracle.

Using property-based tests as the oracle, Geminus learns an approximate class invariant for class `PriorityQueue` and `ArrayDeque`. The current assertion grammar is not sufficiently expressive to generate a parametrized assertion such as `queue[(i-1)/2].compareTo(queue[i])<=0`, which is required for item `PriorityQueue`. Nevertheless, the learned invariant is more accurate than the underlying oracle. In contrast, Geminus learns a less accurate class invariant for `QueueAr`. While the assertion grammar is expressive enough to generate a suitable assertion with multiple conditions that resolves the indistinguishability, the current assertion limit is insufficient in this case.

Regarding **RQ2**, our benchmarks in Tables 3 and 4 demonstrate Geminus's ability to learn a class invariant that outperforms the oracle, resulting in a lower number of false-negatives. Both cases of approximate invariants are due to the inability of the assertion grammar to generate suitable assertions. To generate parametrized assertions, the assertion grammar needs to be extended with lambda expressions. To better support assertions with multiple conditions, which would pave the way for analyzing more complex Java projects, we plan to replace our conjunctive assertion model with a conjunctive normal form model for model training (cf. Section 6).

4.5 Comparing Geminus, Daikon, and Invariant Validation Methods

Daikon [8] generates assertions using templates and retains only those assertions that hold for valid objects. It performs equally well for simple data structures like `StackAr`, but it generates less accurate class invariants for other benchmark items. For `SimpleSquare`, it identifies the incorrect invariant $w = h \wedge w \geq 0$, which fails to detect $\lceil 0 \rceil \lceil 0 \rceil$. While [20] excludes unqualified calls, Daikon considers them, which may result in learning an overly permissive invariant. In contrast, Geminus considers qualified calls only and learns the correct invariant.

The invariants learned by Geminus may produce false-positives, but never did so in our experiments. The invariants documented in the state validation methods also produce no false-positives, as anticipated. However, Daikon does report false-positives for `BitSet` and `LinkedList`. For `BitSet`, this is due to the random walk configuration inadequately representing the object state space, which leads Daikon to retain the overly restrictive assertion `words[] elements >= 0`, encoding that all array elements are greater than or equal to zero. Because Geminus solely adds assertions to detect previously undetected invalid objects, it learns the correct invariant in this example. While this mechanism proves advantageous when dealing with unrepresentative valid objects, Geminus relies on a representative set of invalid objects.

The `LinkedList` class uses a doubly-linked list structure with `prev` and `next` attributes. Daikon detects assertions aiding program comprehension, but it lacks the necessary guards to avoid false-positives. While Daikon only considers valid objects and thus does not require an additional oracle to detect invalid objects, it may learn overly permissive invariants. For example, Daikon identifies the doubly-linked style through the `first == first.next.prev` assertion. However, it overlooks the need for a guard to prevent null dereferencing. Identifying necessary assertions containing guards is a challenging task when only valid objects are available. Considering invalid objects assists Geminus in finding the necessary assertions, like `first != last ? first == first.next.prev : true`. Despite its recursive structure, Geminus learns an invariant that accurately detects all invalid objects. This is possible because the bounded-exhaustive object state generator only covers object states for `LinkedList`, including up to three list nodes. Note that linkage-based classes exhibit large object state spaces even for a small number of linked elements, which is due to reference aliasing. While the documented validation method accurately characterizes the case of an empty list, it imposes an overly permissive constraint for non-empty lists, namely `first.prev == null && last.next == null`. The crucial constraint that the previous attribute of the next node is the current node is not documented.

The *linearization* [7] technique maps a linkage-based structure to an array representation. We can enrich our grammar with the *closure* abstraction to store the objects that are reachable from a given object, using a specific attribute in an array. While the linearization in [7] is used to reason about the *values* stored in a list, this closure abstraction allows one to characterize the double linkage *structure* by expressing that the closure from the `first` element via the `next` attribute is reverse to the closure from the `last` element via the `prev` attribute. In `LinkedList*`, Geminus uses this grammar to learn an invariant that generalizes to lists of arbitrary length.

The invariant validation methods for `BitSet`, `ArrayList`, and `Vector` require null elements at the *next* free array location, while our ground-truth checks all *remaining* locations. Both constraints do *not* affect the functional behavior and are thus not detectable by our oracles. In practice, invariants ensuring a functionally equivalent behavior typically suffice. Similarly, `ArrayDeque` requires elements in the queue to be different from null. It concludes from a null value when fetching the first/last element that the queue is empty. The documentation mentions that all non-live elements in the array are null, but this is only partially checked in their `checkInvariants` method, leading to numerous undetected invalid objects.

Regarding **RQ3**, our benchmark in Table 4 demonstrates that Geminus learns more accurate invariants when using the more accurate property-based tests as oracle, instead of the random walk oracle. Moreover, it often outperforms Daikon in terms of accuracy. Unlike Daikon, our tool identifies necessary guards for complex object states most of the time, avoiding overly permissive or incorrect invariants. Notably, Geminus achieves greater accuracy than the documented validation methods, especially for the complex object states of `LinkedList` or `ArrayDeque`.

5 Related Work

This section contrasts our dynamic class invariant learning to related dynamic assertion learning approaches.

Daikon [8] exhaustively instantiates its assertion templates and retains only those assertions that hold for all observed states at desired program locations. In contrast, Geminus uses the first assertion that suffices to detect a so far misclassified invalid object. Because Daikon considers valid objects only, it relies on static analysis to prune overly permissive, equivalent, or redundant assertions. In contrast, Geminus employs invalid objects to exclude such assertions, which allows us to consider a much larger set of candidate assertions.

PIE [27] learns preconditions and loop invariants from (in)valid objects and uses a feature grammar to construct assertions in conjunctive normal form on-the-fly; however, Valiant's algorithm [36] limits PIE to small formulas. While PIE requires a postcondition to correctly label the set of predefined program states during learning, Geminus uses behavioral oracles to detect invalid objects.

Alearner [30] derives preconditions and uses a test suite to detect invalid method inputs. While Geminus keeps the object graph of each (in)valid example, Alearner only stores an abstraction, which limits precondition expressiveness and hinders manual inspection of training data. Alearner uses program mutation to obtain potentially invalid object states, but does not validate this assumption.

OASIs [15] assesses soundness and completeness of an assertion located within the program. Similar to our random walks, OASIs generates execution scenarios to identify overly restrictive assertions. It uses mutation testing to deem an assertion overly permissive; however, this technique cannot be applied to class invariants, because they cannot be mapped to a *single* program location. *GAssert* [35] uses OASIs to evaluate the quality of an assertion and enhance it for soundness, completeness, and assertion size using an evolutionary learning algorithm. Its evolutionary technique can be an alternative to our grammar-based assertion enumeration, but necessitates defining evolutionary operators.

Proviso [2] addresses, like Geminus does, complex object states, but learns preconditions from observer methods. In contrast, Geminus learns class invariants from private attributes. While Proviso uses a test generator to obtain (in)valid argument values, invalid object states cannot be derived in this way. If no distinguishable feature can be constructed, Proviso relabels valid objects as invalid. Geminus' objects are guaranteed to be (in)valid.

Hanoi [22] and Geminus both learn invariants from (in)valid objects. While Hanoi's notion of constructible value bears similarity with random walks, their invalid objects are not proven invalid and must be recomputed after finding a new so far misclassified valid object. Hanoi learns representation invariants for types in a functional language and constructs a single definition that captures the recursive structure of the type. In contrast, Geminus iteratively refines a set of assertions to learn the invariant of a class in an object-oriented language.

EvoSpex [25] employs an evolutionary algorithm, but learns postconditions from (in)valid pre/post state pairs. Invalid pairs are obtained via state mutation, which does however not necessarily yield invalid states. Geminus solves this

problem for class invariants using behavioral oracles, and only considers thereby proven invalid states. While Geminus utilizes Java expressions, EvoSpex encodes assertions in the Alloy language [14]. The assertion enumeration component in Geminus is language agnostic and can be replaced with, e.g., Alloy.

SpecFuzzer [23] tackles the problem that inferred specifications often contain equivalent assertions. It uses Daikon to remove overly restrictive assertions and then applies program mutation to derive possibly invalid states in order to construct equivalence partitions among the remaining assertions. Geminus prevents the generation of equivalent assertions, similar to SpecFuzzer, via observational equivalence reduction [1,28]. While equivalence partitions can be constructed without knowing whether a state is valid or invalid, guaranteed to be invalid states allow us to assess whether an invariant is sufficient. Geminus generates new assertions until a suitable assertion that detects an invalid state is found.

6 Conclusions

To ensure that modifications to legacy software conform to existing assumptions, it is essential to make implicit guarantees explicit, e.g., in the form of method preconditions and class invariants. However, class invariants encoding object state assumptions are rarely documented and almost never checked automatically.

In this paper, we presented a dynamic analysis for class invariant learning that automatically derives (in)valid objects and distinguishes between them by grammar derived assertions. We leverage random walks in object state spaces to find valid objects and a combination of complex test input generators from bounded-exhaustive testing with behavioral oracles to find invalid objects. In this setting, random walks can even be reused as behavioral oracles. Our prototype tool *Geminus* improves upon related tools such as *Daikon* by learning invariants for complex classes, such as dynamic data structures included in the `java.util` package, resulting in a higher accuracy in detecting invalid objects. Considering invalid objects, too, allows Geminus to prevent the generation of equivalent assertions, thereby leading to concise invariants without the need for static assertion equivalence checks.

The capabilities of dynamic class invariant learning approaches primarily rely on finding so far misclassified (in)valid objects and training a suitable invariant model. While finding execution paths that result in a representative set of valid objects is well understood in the context of software testing, finding representative *invalid* objects is studied less and should be in the focus of future work. Sampling object states while executing a mutated program is likely a source for potentially invalid objects worth to be explored. Our conjunctive assertion model struggles to scale with respect to invariants containing multiple guards per assertion. Future work should focus on crafting heuristics for learning formulas in conjunctive normal form to model complex class invariants with multiple guards.

Data-Availability Statement The source code of Geminus, the benchmark items, the evaluation results and instructions for reproduction are available online via DOI **10.5281/zenodo.10514765**.

References

1. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification (CAV). LNCS, vol. 8044, pp. 934–950. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_67
2. Astorga, A., Madhusudan, P., Saha, S., Wang, S., Xie, T.: Learning stateful preconditions modulo a test generator. In: McKinley, K.S., Fisher, K. (eds.) Conference on Programming Language Design and Implementation (PLDI). pp. 775–787. ACM (2019). https://doi.org/10.1145/3314221.3314641
3. Boyapati, C., Khurshid, S., Marinov, D.: Korat: Automated testing based on Java predicates. In: Frankl, P.G. (ed.) International Symposium on Software Testing and Analysis (ISSTA). pp. 123–133. ACM (2002). https://doi.org/10.1145/566172.566191
4. Cheng, D.Y., Deutsch, J.T., Dutton, R.W.: "Defensive programming" in the rapid development of a parallel scientific program. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **9**(6), 665–669 (1990), https://doi.org/10.1109/43.55196
5. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (eds.) International Conference on Functional Programming (ICFP). pp. 268–279. ACM (2000). https://doi.org/10.1145/351240.351266
6. Coppit, D., Yang, J., Khurshid, S., Le, W., Sullivan, K.J.: Software assurance by bounded exhaustive testing. IEEE Trans. Software Eng. **31**(4), 328–339 (2005). https://doi.org/10.1109/TSE.2005.52
7. Ernst, M.D., Griswold, W.G., Kataoka, Y., Notkin, D.: Dynamically discovering program invariants involving collections. In: University of Washington Department of Computer Science and Engineering technical report UW-CSE-99-11-02, (Seattle, WA), November 16, 1999. Revised March 17, 2000. (2000)
8. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015
9. Fink, G., Bishop, M.: Property-based testing: A new approach to testing for assurance. ACM SIGSOFT Softw. Eng. Notes **22**(4), 74–80 (1997). https://doi.org/10.1145/263244.263267
10. Fraser, G., Arcuri, A.: Evosuite: Automatic test suite generation for object-oriented software. In: Gyimóthy, T., Zeller, A. (eds.) Symposium on the Foundations of Software Engineering and European Software Engineering Conference (FSE/ESEC). pp. 416–419. ACM (2011), https://doi.org/10.1145/2025113.2025179
11. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) International Conference on Software Engineering (ICSE). pp. 225–234. ACM (2010), https://doi.org/10.1145/1806799.1806835
12. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. Commun. ACM **21**(12), 1048–1064 (1978), https://doi.org/10.1145/359657.359666
13. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P.J., Lüttgen, G., Simons, A.J.H., Vilkomir, S.A., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2), 9:1–9:76 (2009), https://doi.org/10.1145/1459352.1459354

14. Jackson, D.: Alloy: A language and tool for exploring software designs. Commun. ACM **62**(9), 66–76 (2019), https://doi.org/10.1145/3338843
15. Jahangirova, G., Clark, D., Harman, M., Tonella, P.: Oasis: Oracle assessment and improvement tool. In: Tip, F., Bodden, E. (eds.) International Symposium on Software Testing and Analysis (ISSTA). pp. 368–371. ACM (2018), https://doi.org/10.1145/3213846.3229503
16. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006). https://doi.org/10.1145/1127878.1127884
17. Manès, V.J.M., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Trans. Software Eng. **47**(11), 2312–2331 (2021). https://doi.org/10.1109/TSE.2019.2946563
18. Marinov, D., Khurshid, S.: Testera: A novel framework for automated testing of Java programs. In: International Conference on Automated Software Engineering (ASE). p. 22. IEEE Computer Society (2001). https://doi.org/10.1109/ASE.2001.989787
19. Meyer, B.: Eiffel: A language and environment for software engineering. J. Syst. Softw. **8**(3), 199–246 (1988). https://doi.org/10.1016/0164-1212(88)90022-2
20. Meyer, B.: Class invariants: concepts, problems, solutions. CoRR **abs/1608.07637** (2016). https://doi.org/10.48550/arXiv.1608.07637
21. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: International Conference on Software Engineering (ICSE). pp. 771–774. IEEE Computer Society (2007). https://doi.org/10.1109/ICSE.2007.48
22. Miltner, A., Padhi, S., Millstein, T.D., Walker, D.: Data-driven inference of representation invariants. In: Donaldson, A.F., Torlak, E. (eds.) International Conference on Programming Language Design and Implementation (PLDI). pp. 1–15. ACM (2020). https://doi.org/10.1145/3385412.3385967
23. Molina, F., d'Amorim, M., Aguirre, N.: Fuzzing class specifications. In: International Conference on Software Engineering (ICSE). pp. 1008–1020. ACM (2022), https://doi.org/10.1145/3510003.3510120
24. Molina, F., Degiovanni, R., Ponzio, P., Regis, G., Aguirre, N., Frias, M.F.: Training binary classifiers as data structure invariants. In: Atlee, J.M., Bultan, T., Whittle, J. (eds.) International Conference on Software Engineering (ICSE). pp. 759–770. IEEE / ACM (2019), https://doi.org/10.1109/ICSE.2019.00084
25. Molina, F., Ponzio, P., Aguirre, N., Frias, M.F.: Evospex: An evolutionary algorithm for learning postconditions. In: International Conference on Software Engineering (ICSE). pp. 1223–1235. IEEE Computer Society (2021), https://doi.org/10.1109/ICSE43902.2021.00112
26. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: International Conference on Software Engineering (ICSE). pp. 75–84. IEEE Computer Society (2007), https://doi.org/10.1109/ICSE.2007.37
27. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Krintz, C., Berger, E.D. (eds.) Conference on Programming Language Design and Implementation (PLDI). pp. 42–56. ACM (2016). https://doi.org/10.1145/2908080.2908099
28. Peleg, H., Polikarpova, N.: Perfect is the enemy of good: Best-effort program synthesis. In: Hirschfeld, R., Pape, T. (eds.) European Conference on Object-Oriented Programming (ECOOP). LIPIcs, vol. 166, pp. 2:1–2:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.2

29. Pham, L.H., Sun, J., Le, Q.L.: Compositional verification of heap-manipulating programs through property-guided learning. In: Lin, A.W. (ed.) Asian Symposium on Programming Languages and Systems (APLAS). LNCS, vol. 11893, pp. 405–424. Springer (2019), https://doi.org/10.1007/978-3-030-34175-6_21
30. Pham, L.H., Thi, L.T., Sun, J.: Assertion generation through active learning. In: Duan, Z., Ong, L. (eds.) International Conference on Formal Engineering Methods (ICFEM). LNCS, vol. 10610, pp. 174–191. Springer (2017). https://doi.org/10.1007/978-3-319-68690-5_11
31. Ponzio, P., Bengolea, V.S., Brida, S.G., Scilingo, G., Aguirre, N., Frias, M.F.: On the effect of object redundancy elimination in randomly testing collection classes. In: Galeotti, J.P., Gorla, A. (eds.) International Workshop on Search-Based Software Testing (ICSE). pp. 67–70. ACM (2018), https://doi.org/10.1145/3194718.3194724
32. Pugh, W.W.: Skip lists: A probabilistic alternative to balanced trees. In: Dehne, F.K.H.A., Sack, J., Santoro, N. (eds.) Workshop on Algorithms and Data Structures (WADS). LNCS, vol. 382, pp. 437–449. Springer (1989), https://doi.org/10.1007/3-540-51542-9_36
33. Sankaranarayanan, S., Chaudhuri, S., Ivancic, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Ryder, B.G., Zeller, A. (eds.) International Symposium on Software Testing and Analysis (ISSTA). pp. 295–306. ACM (2008), https://doi.org/10.1145/1390630.1390666
34. Smith, C., Albarghouthi, A.: Program synthesis with equivalence reduction. In: Enea, C., Piskac, R. (eds.) International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI). LNCS, vol. 11388, pp. 24–47. Springer (2019), https://doi.org/10.1007/978-3-030-11245-5_2
35. Terragni, V., Jahangirova, G., Tonella, P., Pezzè, M.: Evolutionary improvement of assertion oracles. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 1178–1189. ACM (2020), https://doi.org/10.1145/3368089.3409758
36. Valiant, L.G.: A theory of the learnable. Commun. ACM **27**(11), 1134–1142 (1984), https://doi.org/10.1145/1968.1972
37. Weiss, M.A.: Data structures and algorithm analysis in Java, vol. 2. Addison-Wesley (2007)
38. Zeller, A.: Why programs fail - A guide to systematic debugging, 2nd ed. Academic Press (2009)