EvoSpex: An Evolutionary Algorithm for Learning Postconditions

Anonymous authors

***Abstract*—Software reliability is a primary concern in the construction of software, and thus a fundamental component in the definition of software quality. Analyzing software reliability requires a *specification* of the intended behavior of the software under analysis, and at the source code level, such specifications typically take the form of *assertions*. Unfortunately, software many times lacks such specifications, or only provides them for scenario-specific behaviors, as assertions accompanying tests. This issue seriously diminishes the analyzability of software with respect to its reliability.**

In this paper, we tackle this problem by proposing a technique that, given a Java method, automatically produces a specification of the method's behavior, in the form of postcondition assertions. This mechanism is based on generating executions of the method under analysis to obtain *valid* pre/post state pairs, mutating these pairs to obtain (allegedly) *invalid* ones, and then using a genetic algorithm to produce an assertion that is satisfied by the valid pre/post pairs, while leaving out the invalid ones. The technique, which targets in particular methods of reference-based class implementations, is assessed on a benchmark of open source Java projects, showing that our genetic algorithm is able to generate post-conditions that are stronger and more accurate, than those generated by related automated approaches, as evaluated by an automated oracle assessment tool. Moreover, our technique is also able to infer an important part of manually written rich postconditions in verified classes, and reproduce contracts for methods whose class implementations were automatically synthesized from specifications.

I. Introduction

The quality of software systems is typically defined around various dimensions, such as reliability, usability, efficiency, etc. Among these, reliability is in general considered a fundamental attribute of software quality, and a primary concern in software development [12], [16]. Analyzing software reliability is strongly related to finding software defects, i.e., actual software behaviors that diverge from the expected behavior. Discovering such defects requires one to state *what* the expected behavior is, in other words, a *specification* of the software. Many times such specifications are either implicit, or stated informally, diminishing the possibility of exploiting specifications for (automated) reliability analysis.

Software specifications can appear in different forms. At the level of source code, when present, they generally manifest either as *comments*, i.e., informal descriptions of what the software is supposed to do, or more formally as *program assertions*, i.e., (usually executable) statements that assert properties that the software must satisfy at certain points during program executions. The former are more common, but cannot be straightforwardly used for automated reliability analysis. The latter, on the other hand, are readily usable for program analysis, especially when stated as *contracts* [21], but they are seldom found accompanying source code. Moreover, many times program assertions state scenario-specific properties, e.g., statements that only express the expected software behavior for a test case, as opposed to the more general, and also significantly more useful, assertions associated with contract elements such as invariants and pre/post-conditions.

Due to the above described situation regarding specifications at the level of source code, the *specification inference problem* (a special case of the well known *oracle problem* [3]), i.e., taking a program *without* a corresponding specification and attempting to automatically produce one that captures the current program's behavior, is receiving increasing attention by the software engineering community. Automatically inferring specifications from source code is a relevant topic, as it enables a number of applications, including program comprehension, software evolution and maintenance, bug finding [31], and specification improvement [31], [15], among others.

In this paper, we tackle this problem by proposing a technique that, given a Java method, automatically produces a specification of the method's current behavior, in the form of postcondition assertions. This mechanism is based on generating *valid* and *invalid* pre/post state pairs (i.e., state pairs that represent, and do not represent, the method's current behavior, respectively), which guide a genetic algorithm to produce a JML-like assertion characterizing the valid pre/post pairs, while leaving out the invalid ones. The generation of valid pre/post pairs is based on executing the method on a *bounded exhaustive* test set, generated by exercising the method inputs' APIs using user-defined ranges for basic datatypes, and bounding their execution sequences. The invalid pre/post pairs, on the other hand, are obtained by *mutating* valid pairs, i.e., arbitrarily modifying the post-states so that the resulting pair does not belong to the set of valid pairs. This mutation-based approach is unsound in the sense that it may lead to *valid* pairs, an issue that the design of our genetic algorithm takes into account. The approach is particularly well-suited for reference-based class implementations, such as heap-allocated structural objects, and custom types, with (implicit) strong representation invariants.

We assess our technique on a benchmark of open source Java projects taken from [11], featuring complex implementations of reference-based classes. In these case studies, our genetic algorithm is able to generate post-conditions that are stronger and more accurate, than those generated by related specification-inference approaches, as evaluated by an automated oracle assessment tool [15]. Moreover, our technique is also able to infer an important part of manually written rich postconditions (strong contracts used for verification) present in verified classes [36], and reproduce contracts for methods whose class implementations were automatically synthesized from specifications [19].

II. Background

A. Assertions as Program Specifications

The use of assertions as program specifications dates back to the works of Hoare [13] and Floyd [9], in the context of program verification and associated with the concept of program correctness. Technically, an assertion is a statement predicating on program states, that can be used to capture *assumed properties*, as in the case of preconditions, or *intended properties*, as in the case of postconditions. A program P accompanied by a precondition *pre* and postcondition *post* is said to be (partially) correct with respect to this specification, if every execution of P starting in a state that satisfies *pre*, if it terminates, it does so in a state that satisfies *post* [13]. That is, every valid terminating execution of P, i.e., every execution satisfying the requirements stated in the precondition, must lead to a state satisfying the postcondition.

While program assertions originated in the context of program verification, they soon permeated into programming languages constructs and (informal) programming methodologies. More recently, they have been central to the definition of methodologies for software design, notably *design by contract* [22]. Most modern imperative and object-oriented programming languages support assertions, either as built-in constructions [23] or through mature libraries such as Code Contracts [2] and JML [5]. Moreover, libraries for unit testing make extensive use of assertions to automate checking the expected results of running a test case.

Preconditions are more commonly seen in source code, e.g., within methods in the form of state and argument checks, throwing appropriate exceptions when these are found invalid, preventing normal execution. Postconditions, on the other hand, are less common. Post-execution checks are commonly seen as part of test cases, although they rarely capture postconditions, in the sense of general properties that every execution must satisfy on termination; post-execution checks in tests generally state properties that should be satisfied for the specific test where they are stated.

B. Quality of Assertions

As described above, program assertions are a way of capturing the expected software behavior via expressions that convey intended properties of program states in specific parts of a program. Such expected behavior can be captured with different degrees of precision, leading to assertions of different quality. The most typical issue with program assertions is the misclassification of invalid program states as valid. This is essentially the effect of having *weak* assertions, that are able to detect some, but not all, faulty situations. It is rarely considered a *defect* in the assertion, but an inherent issue associated with a balance between expressiveness and economy/efficiency in the definition of assertions. Indeed, it is even considered methodologically correct to express weak (and efficiently checkable) assertions [22]. Following the terminology put forward in [15], a real program execution leading to an invalid program state that a corresponding assertion is unable to detect is called a *false negative*.

A second issue with program assertions is the dual of the previous, i.e., the misclassification of valid program states as invalid. This issue indicates that the assertion is *wrong*, as it does not properly specify the intended behavior of the software. Such issues are typically considered to be specification defects. This situation can also often arise as a consequence of software evolution, when required changes in program behavior are (correctly) implemented, but the accompanying assertions are not kept in synchrony with the evolved behavior [6]. A real program execution leading to a valid program state, that a corresponding assertion classifies as an assertion violation, is a *false positive*, according to the terminology put forward in [15].

Assessing the quality of assertions accompanying a program is a very challenging problem, that is typically performed manually. A way of measuring the quality of assertions is by attempting to determine the number of false positives and false negatives that a given assertion has. This idea has been exploited in [15], where an automated mechanism for evaluating the quality of assertions, based on evolutionary computation, is proposed. The approach presented therein executes an evolutionary test generation tool (the well-known tool EvoSuite [10]) that tries to find false positives and false negatives, and when found, produces witnessing test cases, that can be used to (manually) improve the corresponding assertions. It is worth remarking that, for contracts specified in standard assertion languages, it is hardly expected for a contract to *fully* capture the behavior of a program. As explained in [27], precisely capturing a program's intended semantics requires additional mechanisms, such as the use of model classes, that imply the manual definition of abstractions of the state space of the program being specified. In terms of the above-mentioned issues with program assertions, it means that, technically, one can very often come up with false negatives, i.e., finding states that satisfy a given assertion but correspond to incorrect program behavior.

III. An Illustrating Example

As an illustrating example, let us consider a Java class implementing lists, partially shown in Figure 1[1]. This class implements list operations over balanced trees, supporting insertion and deletion from the list in $O(\log n)$, as opposed to the classic array-based and linked-list based list implementations. Let us focus on method add, that inserts an element in the list. Notice how the precondition of the method is captured in the source code, checking the validity of the index for insertion and that the tree has not reached its maximum size. The method postcondition, on the other hand, is not

[1]This implementation is taken from https://www.nayuki.io/page/avl-tree-list

```
import java.util.AbstractList;

public final class AvlTreeList<E> extends AbstractList<E> {

   private Node<E> root;

   public void add(int index, E val) {
      if (index < 0 || index > size())
         throw new IndexOutOfBoundsException();
      if (size() == Integer.MAX_VALUE)
         throw new IllegalStateException("Max size reached");
      root = root.insertAt(index, val);
   }

   private static final class Node<E> {

      private E value;
      private int height;
      private int size;
      private Node<E> left;
      private Node<E> right;

      public Node<E> insertAt(int index, E obj) {
         assert 0 <= index && index <= size;
         if (this == EMPTY_LEAF)
            return new Node<>(obj);
         int leftSize = left.size;
         if (index <= leftSize)
            left = left.insertAt(index, obj);
         else
            right = right.insertAt(index-leftSize-1, obj);
         recalculate();
         return balance();
      }

  }

}
```

Fig. 1. Add method of class AvlTreeList

present in this implementation. Having the postcondition has multiple applications, in particular as assertions for testing future improvements of this method, and as a declarative description of what this method does (how it operates on the data structure), among many others. Writing the specification is, however, nontrivial, and thus coming up with the right expression for the postcondition is an important problem.

A well-known tool to assist the developer in this situation is Daikon [7]. Daikon performs run-time invariant detection, it runs the program on a set of test cases, and observes which properties hold during these runs at particular program points, such as after method invokations. It then suggests as likely invariants those properties that were not falsified by any execution, or equivalently, that held true for all observed executions. The quality of the obtained invariants strongly depends on the program executions considered by Daikon (i.e., the set of tests that the user provides), and the set of candidate expressions to be considered. In particular for method `add` in Figure 1, Daikon produces the postcondition shown in Figure 2, when fed with *all* valid tree lists of size up to 4. The shown postcondition is actually that produced by Daikon, but manually filtering out invalid expressions (inducing false positives) that could not be falsified by the suite. Still, as it can be seen, the postcondition generated in this case is relatively weak: we would expect to have some further information about

```
// height
this.root.height >= old_this.root.height &&
this.root.height >= old_this.root.left.height &&
this.root.height >= old_this.root.right.height &&
// size
this.root.size > old_this.root.height &&
this.root.size > old_this.root.left.height &&
this.root.size > old_this.root.right.height &&
// left height
this.root.left.height <= old_this.root.height &&
this.root.left.height >= old_this.root.left.height &&
this.root.left.height >= old_this.root.right.height &&
// right height
this.root.right.height <= old_this.root.height &&
this.root.right.height >= old_this.root.left.height &&
this.root.right.height >= old_this.root.right.height
```

Fig. 2. Postcondition generated by Daikon for AvlTreeList.add(int, E)

how node attributes get manipulated in this implementation of lists over trees. The reason why Daikon produces this very simple postcondition in this case has to do with the set of candidate expressions that Daikon considers, which are produced from the definition of the program, and are restricted to relatively simple program properties (e.g., structural constraints, membership checking, etc., are not considered) [7].

Our aim is to provide stronger postconditions in cases such as the above. Our approach is, in essence, similar to Daikon's: the method under analysis is run for different inputs, and from information extracted from these runs we propose a postcondition for the method. There are, however, multiple differences. Firstly, our approach is based on generating runs for the method under analysis *bounded exhaustively*, as opposed to Daikon, which requires tests to be provided (in the above example, the suite we provided Daikon with was the one that our technique produces). Our technique for generating the bounded exhaustive test suite is based on exercising the API of the inputs of the program under analysis, contrary to related approaches that require a specification [4], [17]. Secondly, we consider both *valid* and *invalid* program states (although, as we explain later on, the approach to generate invalid states may unsoundly generate valid ones) in attempting to determine a method's postcondition, instead of only *valid* executions, as is the case with Daikon. Third, our approach is based on evolving specifications, instead of considering non-falsified candidate properties. The details of our technique are described in the next section. Let us just mention that, for method `add` of class `AvlTreeList`, our obtained postcondition is the one shown in Figure 3. Notice how the size update (referring to the relation between the pre and post states) and the membership of the inserted element are captured, as well as some structural properties of the representation.

IV. EvoSpex

We now present the details of our technique for inferring method postconditions. An overview is shown in Fig. 4. The technique is composed of two main phases: state generation and learning. During state generation, we produce pre/post program state pairs which are later on used in the learning

```
// root
this.root != null &&
this.root.left != null &&
// height
all n : this.root.*(left+right) : (
  n.left != null => n.height > n.left.height &&
  n.right != null => n.height > n.right.height
) &&
// size
old_this.root.size < this.root.size &&
this.root.size == #(this.root.*(left+right - null)) - 1 &&
all n : this.root.*(left+right) : (
  n.left != null => n.size > n.left.size &&
  n.right != null => n.size > n.right.size
) &&
// arguments
index != this.root.size &&
val in this.root.*(left+right).value &&
// structural
all n : this.root.*(left+right) : n !in n.^(left + right)
```

Fig. 3. Postcondition generated by our tool for AvlTreeList.add(int, E)

phase to guide the search for suitable postcondition assertions. Two kinds of state pairs are generated: *valid* ones, which capture actual method behaviors that candidate assertions should satisfy; and *invalid* ones, which *attempt* to capture incorrect behaviors (pre/post pairs that do not correspond to the current method behavior), that candidate assertions should not satisfy. Valid pre/post pairs are generated by executing the target method, using a test generation technique; clearly, these pairs correspond to the behavior of the method, as they were generated from its execution. Invalid pre/post pairs, on the other hand, are generated by *mutating* valid pairs, going out of the set of valid pairs; contrary to the case of valid pairs, it is not guaranteed that our invalid pairs are indeed incorrect method behaviors, i.e., that they represent behaviors that are *not* exhibited by the method. Two important issues must be remarked about our mechanism for generated allegedly invalid state pairs. Firstly, its effectiveness clearly depends on the exhaustiveness of the set of valid pairs: the more exhaustive the set of valid pairs, the greater the chances that mutating out of this set leads to a truly incorrect method behavior. Secondly, as the soundness of the mechanism for invalid state pair generation cannot be guaranteed, one may risk discarding useful assertions based on wrong invalid state assumptions. The former motivates the use of a bounded-exhaustive test generation approach for valid state pairs. The latter drives an asymmetric treatment of valid and invalid state pairs in the fitness function, that gives the reliable information provided by valid pairs a greater relevance. We further describe in the section how we handle these issues, as well as all the main components of our genetic algorithm.

A. Generation of Valid/Invalid Method Executions

The learning phase of our algorithm depends on a set of valid/invalid method executions, which guide the search for postcondition assertions. This is an important part of our algorithm. The overall process starts by generating runs of the target method m, collecting the pre/post states $\langle s, s' \rangle$ of each execution; these are the *valid* execution pairs V. In order to generate *invalid* execution pairs I, valid pairs are *mutated*: for a valid pair $\langle s, s' \rangle$, we mutate s' into s'', and check that $\langle s, s'' \rangle$ does not belong to V, to consider it part of I. Of course, the mutated pre/post pair may actually correspond to a valid execution of m that we had not generated in V. The effectiveness of this latter approach depends on how thorough V is (although we may still generate "unseen" valid execution pairs via mutation), motivating a bounded exhaustive approach for generating valid execution pairs.

The mechanism for generating valid execution pairs works as follows. Let $C, C_1, \ldots, C_n$ be classes, and m, the target method, a method in C with parameters of types $C_1, \ldots, C_n$. The initial states for the execution of m will be tuples $\langle o_C, o_{C_1}, \ldots, o_{C_n} \rangle$ of objects of types $C, C_1, \ldots, C_n$, respectively. We build the objects to form these tuples, for each class, bounded exhaustively, in the following way. Let C_i be a class, and methods $b_1, \ldots, b_l$ a set of *builders* for C_i, i.e., a set of manually identified methods that can be used to create objects of class C_i. For instance, for a set collection, builders would include constructors and insertion routines. Given a bound k (maximum length for method sequences), we build a set of objects of class C_i using a variant of Randoop [26] (which randomly generates sequences of methods of C_i's API, of increasing length), with two main modifications:

- The random selection of a method to extend a previously produced trace t (test case), implemented in [26], is replaced by a mechanism to systematically select *all* methods in $b_1, \ldots, b_l$, leading to l different extensions of t. This is applied until the bound k is reached.
- A state matching mechanism is implemented, to reduce the number of method combinations: when a newly produced trace leads to an object that matches a previously collected one, the trace (and the object) are discarded. The state matching approach borrows the canonical object representation put forward in [29].

Besides the bound k on trace length, the state matching mechanism also requires a maximum number of objects per type, and a range for primitive types (e.g., 0..k-1 for integers). This is a k-based scope, as defined in finitization procedures in [4] (a standard issue in bounded exhaustive generation). Using the above mechanism, we build the tuples of initial states, to execute m. We execute m in each of these tuples, and collect the corresponding post-states, building in this way the set V of *valid* pre states and corresponding post states for m.

The mutations applied to produce the "invalid" pre/post states, take a valid execution pair $\langle s, s' \rangle$, and create $\langle s, s'' \rangle$, where s'' mutates s' by selecting a random field in the receiving object or return value (the constituents of s'), and replacing the value by a randomly generated value of the corresponding type within the above mentioned scope.

B. Chromosomes representing Candidate Postconditions

Our representation of candidate assertions is based on the encoding used in [24], where chromosomes represent conjunctions of assertions (each gene in a chromosome represents

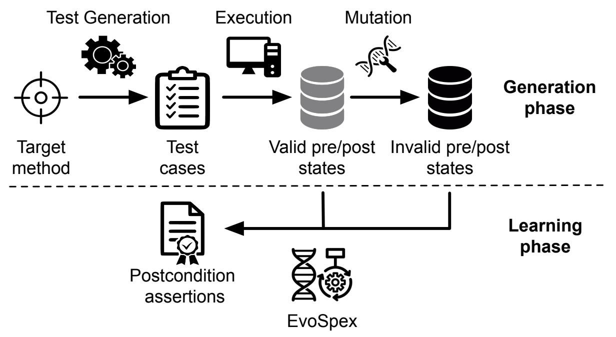

Fig. 4. An overview of the proposed approach

an assertion). That is, given a chromosome c, the candidate postcondition φ_c represented by c is defined as follows:

$$c = \langle g_1, g_2, \ldots, g_n \rangle \Rightarrow \varphi_c = g_1 \wedge g_2 \wedge \ldots \wedge g_n$$

As opposed to what is most common in genetic algorithms, chromosomes have varying lengths in this representation (up to a maximum chromosome length), and gene positions are disregarded by the genetic operators (see below), due to the associativity and commutativity of the conjunction. Genes need to encode complex assertions. Below we describe how genes are built, as well as how they are mutated and combined to produce new ones.

C. Initial Population

Let us describe how we build the initial population, to start our genetic algorithm. In order to create individuals representing "meaningful" postconditions, i.e., assertions stating properties of objects that are reachable at the end of the method executions, we take into account typing information, as in [24]. We consider a *type graph* built automatically from the class under analysis: nodes represent types, and each field f of type B in class A will produce an arc in the graph going from the node representing A to the node representing B. For example, if we consider the `AvlTreeList` class in Fig. 1, the corresponding type graph would be the one shown in Fig. 5. It is straightforward to see that by traversing the graph, typed expressions can be built, using the fields of the object from which the method was executed. Some examples are `this.root`, `this.root.left`, `this.root.size`, `this.root.value`, and so on. Moreover, from loops in the graph, expressions denoting sets, such as `this.root.*left` (the set of nodes reachable from `this.root` via `left` traversals only), `this.root.*right` and `this.root.*(left+right)`, can be created (we are using * for reflexive-transitive closure here, as in [14]). Size one chromosomes are created using expressions denoting a single value, evaluating these on a randomly selected subset of the valid (resp. invalid) method executions, in the following way: if the result of evaluating an expression `expr` in a valid (resp. invalid) tuple t returns a value v, then we create the individual ⟨`expr == v`⟩ (resp. ⟨`expr != v`⟩).

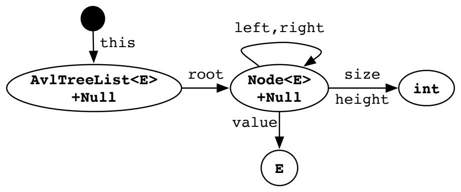

Fig. 5. Type graph for AvlTreeList example.

In addition to these basic individuals we also create chromosomes containing comparisons of random expressions of the same type (e.g., `this.root == this.root.right`), chromosomes with quantified formulas considering expressions denoting sets (e.g., `all n: this.root.*(left+right) - null : n == n.right`), and individuals comparing integer expressions with the cardinality of expressions denoting sets (e.g., `this.root.height == #(this.root.*right)`). Finally, since the method under analysis may have a return value or a set of arguments, we also include, in the set of initial candidates, expressions comparing them against expressions of the same type (e.g., `result < this.f`). The expressions used to compare with the result variable or the arguments, as well as the operators, are randomly chosen.

Notice that *all* our initial chromosomes are size 1 chromosomes. The main reason for this design choice is to allow the genetic algorithm to progressively produce complex candidate postconditions by means of the genetic operators, that we define later on in this section. While this size-one chromosomes for the initial population is non-standard in genetic algorithms, in our case it helps the algorithm to more quickly converge to better fitted individuals. The replication package site contains the results of comparing the effectiveness of our size-one chromosomes in the initial population, with standard size-N chromosomes (we do not include the comparison here due to space restrictions).

D. Fitness Function

Our fitness function assesses how good a candidate postcondition is, by distinguishing between the set V of valid executions and the set I of invalid executions. To do so, before computing the fitness value of a given candidate c, we obtain the postcondition φ_c that c represents, and then compute the sets P and N of positive and negative counterexamples, respectively. These sets are defined as follows:

$$P = \{v \in V | \neg\varphi_c(v)\} \quad N = \{i \in I | \varphi_c(v)\}$$

where φ_c is the postcondition represented by the candidate c. Basically, the sets P and N contain those executions for which the postcondition φ_c does not behave correctly. Recall that, as opposed to the case of V which reliably represents actual execution information of m, the set I may contain mutated executions that are considered "invalid", but correspond to actual executions of m. This motivates a definition of our fitness function that does not treat P and N symmetrically. The fitness function $f(c)$ is computed as follows:

$$\begin{cases} \#P > 0 \rightarrow (\mathsf{MAX} - \#P - \#I) + \left(\frac{1}{l_c + comp_c}\right) + \frac{mca_c}{l_c} \\ \#P = 0 \rightarrow (\mathsf{MAX} - \#N) + \left(\frac{1}{l_c + comp_c}\right) + \frac{mca_c}{l_c} \end{cases}$$

This case-based definition aims at considering the negative counterexamples only when no positive counterexamples are obtained. In fact, for arbitrary candidates c_1 and c_2, if c_1 has no positive counterexample and c_2 has some positive counterexamples, then f is guaranteed to produce worse fitness values for c_2, no matter how many negative counterexamples these candidates have. The rationale here is to make the reliable positive-counterexample information more relevant.

The definition of the fitness function has three parts. The first term reflects the most important aspect: to minimize the number of counterexamples. The fact that when the candidate postcondition φ_c has positive counterexamples, i.e., it is falsified by a correct method execution, the whole set I of invalid executions is considered as counterexamples too, is what guarantees our above observation regarding the prioritization of candidates with no positive counterexamples. More precisely, the first term of the function subtracts $\#I$ when $\#P > 0$, to ensure that the fitness value of such individual is lower that the fitness value of any other individual that only has negative counterexamples. The second term of the fitness function acts as a penalty regarding two aspects: the candidate length l_c and the candidate "complexity" $comp_c$. The candidate length is simply the number of conjuncts in the assertion, and it is considered in order to guide the algorithm towards producing smaller assertions. The candidate complexity is the sum of each conjunct complexity. Intuitively, the complexity of an equality between two integer fields is lower than the complexity of an equality between an integer field with a set cardinality, and both of these are lower than the complexity of a quantified formula, and so on. The last term of the function acts as a reward favoring those candidates with a greater number of "method component assertions" mca_c, i.e., with a high number of conjuncts of the candidate postcondition that represent properties regarding the parameters, the result, or a relation between initial and final object states. As described, the penalty related to the candidate length and complexity as well as the reward prioritizing the method component assertions just contribute a fraction to the fitness value, since we want the algorithm to focus on individuals whose number of counterexamples is approaching zero.

E. Genetic Operators

During evolution, the genetic operators allow the algorithm to explore the search space of candidate solutions, by performing certain operations that produce individuals with new characteristics as well as combinations of existing ones. In particular, our algorithm implements two well known genetic operators, namely the *mutation* and *crossover* operators. Some of these genetic operators were inspired by similar ones introduced in [24], while others are novel. Also, a custom *selection* operator was implemented, to keep in the population those candidates that are more suitable to be part of the real post condition.

Each chromosome gene is selected for mutation with a probability of 0.3, and the operation can perform a variety of modifications depending on the shape of the selected gene expression. From a general point of view, the set of considered mutations are the following:

Gene deletion: it can be applied to any gene and simply removes the gene expression from the chromosome.

Negation: it negates the gene expression and is applied to any gene except quantified assertions.

Numeric addition/subtraction: it is only applied to genes that compare two expressions evaluating to a number, and it adds/subtracts a randomly selected numeric expression to the right-hand side of the comparison.

Expression replacement: it applies to any gene, and it replaces some part of the gene with a randomly selected expression of the same type.

Expression extension: it can be applied to any gene that involves a navigational expression, and it extends this expression with a new field, for example replacing `this.root` by `this.root.left`.

Operator replacement: it replaces an operator by an alternative one. The operators vary depending on the current gene expression. For instance, for relational equalities, the possible operators are $\{==, !=\}$; for numeric comparisons, the operators are $\{==, !=, <, >, <=, >=\}$; and for quantified expressions, the quantifiers are $\{all, some\}$.

To produce combinations of individuals, we use a crossover rate of 0.35. Given two randomly selected chromosomes c_1 and c_2, our *crossover* operator simply produces a new individual that contains the union of the genes of c_1 and c_2, and thus representing the candidate postcondition $\varphi_{c1} \wedge \varphi_{c2}$. An important detail in our *crossover* operator is that before selecting individuals for combination, we filter the population, keeping individuals which only have negative counterexamples, i.e., that represent formulas that are consistent with all valid method executions. The main reason for this policy is that we want the algorithm to join chromosomes that are already consistent with the valid method executions.

Finally, to keep in the population the *best* candidates of each generation, our *selection* operator is defined as follows: given a number n to be used as constant population size, our operator first sorts all the candidate postconditions in decreasing order, and then the candidates to be moved to the next generation are the first $n/2$ individuals plus the best $n/2$ unary non-valid individuals, i.e., size 1 chromosomes whose only gene is a formula that still has positive counterexamples. Additionally, our operator keeps all the unary valid candidates, that is, those that only have negative counterexamples. This last policy in our selection operator allows us to keep in the population all the discovered valid properties that the algorithm can use in future crossover operations.

TABLE I
POSTCONDITIONS INFERRED BY EVOSPEX AND DAIKON AFTER REMOVING FALSE POSITIVES

Method	EvoSpex	Daikon
jiprof - com.mentorgen.tools.profile.runtime.ClassAllocation		
`getAllocCount(): int`	`result == this._count`	`this._count == result && result == old(this._count)`
`incAllocCount(): void`	`this._count == 1 + old(this_count)`	`this._count >= 1 && this._count - old(this_count)- 1 == 0`
jmca - com.soops.CEN4010.JMCA.JParser.SimpleNode		

`jjtSetParent(Node n): void`	`n == this.parent`	`this.parent == old(n)&&`
		`this.children == old(this.children)&&`
		`this.id == old(this.id)&&`
		`this.parser == old(this.parser)&&`
		`this.identifiers == old(this.identifiers)`

bpmail - ch.bluepenguin.email.client.service.impl.EmailFacadeState		
`setState(Integer ID, boolean dirtyFlag): void`	`ID in this.states.keySet()`	`this.states == old(this.states)`
byuic - com.yahoo.platform.yui.compressor.JavaScriptIdentifier		

`preventMunging(): void`	`this.mungedValue == old(this.mungedValue)&&`
	`this.refCount == old(this.refcount)&&`
	`all n : this.declaredScope.*parentScope: n !in n.^parentScope`
		`this.mungedValue == old(this.mungedValue)&&`
		`this.recCount == old(this.refcount)&&`
		`this.declaredScope == old(this.declaredScope)&&`
		`this.markedForMunging == false`

dom4j - org.dom4j.tree.LazyList		

`add(E element): boolean`	`old(this.size)== this.size - 1 &&`
	`result == true &&`
	`element in this.header.*next.element`
		`this.header == old(this.header)&&`
		`this.size >= 1 &&`
		`result == true &&`
		`this.size - old(this.size)- 1 == 0`

V. EVALUATION

To evaluate our technique, we performed experiments focused on the following research questions:

RQ1 *Do the oracles learned by EvoSpex have any deficiency compared to oracles produced by related tools?*

RQ2 *Are the assertions produced by the algorithm close to manually written contracts?*

To evaluate RQ1, we need to consider programs (in our case, Java programs) for which to infer method specifications. As mentioned earlier in the paper, and as it is clear from our candidate assertion state space and evolution operators, we target classes and methods with reference-based implementations, in particular classes where the corresponding internal representation has strong (implicit) invariants. As a source for our benchmark, we considered SF110[2] (originally used in [11]), a collection of 110 Java projects (100 random projects, plus the 10 most popular ones according to SourceForge), that covers a wide variety of software, representative of open source development. Our process of assessing postcondition assertions makes use of the OASIs tool [15], essentially, to evaluate the quality of a postcondition assertion in terms of its associated number of false positives and false negatives. The process of computing this number requires a manual process (as described in [15], to compute the false negatives one first needs to get rid of the false positives, which implies having to manually refine the produced postconditions every time OASIs reports the presence of a false positive). Therefore, we are unable to consider the whole 110 projects in the benchmark. We randomly selected 16 projects, skipping cases in our selection that have a clear dependency on the environment (our technique involves automated test generation, and environment dependencies seriously affect these tools). The 16 projects can be found in Table II. For each case study, we selected various methods with different behaviors for analysis, manually defined a set of builders, and then generated the corresponding valid and invalid method executions with a relatively small scope (3 for all cases). Then, we executed our tool in the following way: for each method m selected for analysis, we executed the genetic algorithm to produce a postcondition for m until it reached 30 generations or a 10 minutes timeout was fulfilled. We repeated this execution 10 times, and then selected the postcondition assertion that repeated the most number of times, from the 10 produced by the algorithm. Additionally, in order to compare our tool with related approaches, we executed Daikon to infer post conditions for each method m. It is important to remark that the test suites that we fed Daikon with to produce postconditions for the methods under analysis, were exactly the same test suites that were used to generate the valid method executions in our technique (our valid bounded exhaustive suites). Both our tool and Daikon can produce assertions leading to false positives (see Section 2 for a comment on this issue), as well as redundant assertions.

The results of this experiment are shown in Tables I and II. Table I presents the postconditions generated by the tools, after removing the false positives and the redundant assertions, with the aim of giving a clear glance of the complexity of the assertions that the techniques are able to generate. We considered these assertions, as the ones produced by the two techniques. We then measured the quality of the corresponding assertions by automatically computing false positives as well as false negatives, using the OASIs [15] tool. Table II shows the results of this quality assessment. Specifically, for each case study, we report in Table II: *(i)* lines of code (LoC) of the evaluated project; *(ii)* number of analyzed methods from the corresponding project; *(iii)* number of assertions produced as part of the postconditions; *(iv)* amount and percentage of false positives present in all generated assertions; and *(v)* number of methods for which false negatives were detected. Notice that, as proposed in [15], false negatives detection is performed once all the false positives have been removed from

[2]https://www.evosuite.org/experimental-data/sf110/

the postcondition (hence the manual task that made us consider a subset of SF110). For both false positives detection and false negatives detection, we executed OASIs with a timeout of one minute. Problems with the OASIs tool prevented us from reporting the number of false negatives for each method and case study; more precisely, when the tool reported the existence of false negatives, it was unable to produce the witnessing counterexamples (test cases), preventing us from measuring the number of false negatives identified by the tool. This issue was confirmed by the developers of the tool. We therefore report the number of methods for which OASIs reported the existence of false negatives. For instance, for project `imsmart`, out of the 3 methods analyzed, OASIs found one of the corresponding assertions discovered by Daikon to have false negatives, and one of the assertions discovered by EvoSpex to have a false negative, too.

The evaluation of RQ2 requires having classes with methods featuring manually written contracts. Moreover, as discussed in Section 2, assertions for run-time checking are typically weak, efficiently checkable assertions, that weakly capture the semantics of the corresponding classes and methods [22], [27]. In order to compare with strong contracts, we took:

- A set of case studies with contracts written for the verification of object oriented programs. More precisely, these programs are written in Eiffel [23], and the accompanying contracts were used for verification using the AutoProof tool [36], a verifier for object-oriented programs written in the Eiffel programming language, for Eiffel programs.
- A set of case studies for which a data representation and method implementations are automatically synthesized from a higher-level specification. More precisely, the synthesized implementations are taken from [19], are generated by the Cozy tool, and are guaranteed to be correct with respect to higher level specifications, which serve as method contracts.

From [36], we specifically considered various methods and their corresponding postconditions, from the following cases: *(i)* **Composite**: A tree with a consistency constraint between parent and children nodes. Each node stores a collection of its children and a reference to its parent; the client is allowed to modify any intermediate node. A value in each node should be the maximum of all children's values. *(ii)* **DoublyLinkedListNode**: Node in a (circular) doubly-linked list with a structural invariant that its left and right links are consistent with its neighbors. *(iii)* **Map<K,V>**: Generic Map ADT implementation, based on two lists that contain the keys and values, and with operations that perform linear searches on the lists. *(iv)* **RingBugger<G>**: Bounded queue implemented over a circular array. Since our tool is for Java, and these implementations are in Eiffel, we had to manually translate the whole classes into Java, for analysis with our tool (this also prevented us from considering more sophisticated case studies in this evaluation).

From [19], we considered several high-level specifications and their corresponding synthesized Java implementations: *(i)* **Polyupdate**, a bag of elements that keeps track of the sum of its positive elements. *(ii)* **Structure**, a simple class encapsulating a function and caching a parameter. *(iii)* **ListComp02**, a structure composed of two collections of different elements, and operations that combine elements of the collections. *(iv)* **MinFinder**, a bag of elements with a min operation. *(v)* **MaxBag**, a set of elements, with a max operation.

In order to infer postconditions for methods in these classes, we first generated valid and invalid method executions, as described earlier in this paper, for each of the target methods using a scope of 4. Then, we executed our algorithm using the same configuration described for RQ1 (30 generations with a 10-minute timeout). Again, we repeated the execution 10 times and selected the most frequently obtained postcondition. Notice that our approach is not using the real contracts already accompanying the target methods. We fully ignore these in the inference approach, and only consider the methods source code, both for the generation of valid/invalid method executions, and for the actual evolutionary inference. A similar procedure was followed for the Cozy case studies. We computed postconditions for the Java implementations, and contrasted them with those in the original high-level specifications, from which the Java implementations were derived.

The results of this experiment are shown in Tables III and IV. In Table III, for each of the target methods, the column Eiffel Contracts lists the properties that are present in the original postcondition (expressed as text, for easier reference). In Table IV, the original postcondition is described in column High-level spec in terms of the abstract state declared in the specification. In both tables, the column EvoSpex indicates which of the corresponding assertions in the original contract, our evolutionary algorithm was able to infer.

Our tool, all the case studies, and a description of how to reproduce the experiments presented in this section can be found in the site of the replication package of our approach [1]. All the experiments were run on an Intel Core i7 3.2Ghz, with 16Gb of RAM, running GNU/Linux (Ubuntu 16.04).

A. Assessment

Let us briefly discuss the results of our evaluation. Regarding RQ1, the results show that our approach is able to generate postconditions containing more complex assertions than the ones produced by Daikon. This is mainly due to the fact that our technique focuses on evolving assertions targeting reference-based conditions in reference-based implementations, as opposed to Daikon whose expressions are comparatively simpler properties, that do not include complex structural constraints, membership checking, etc (with the exception of arrays and implementations of `java.util.List`, for which Daikon generates interesting structural properties). Furthermore, as Table II shows, a significant number of the assertions inferred by our technique are *true positives*, i.e., assertions that hold for all valid post-states of the corresponding methods, for any scope. Of course, this check for true positives is in the end manual (we carefully analyzed how each of the evaluated methods operates, and inspected the obtained assertions after filtering out assertion conjuncts as per

TABLE II
Measuring the quality of postconditions inferred by Daikon and EvoSpex, using OASIs.

Project	LOCs	Methods	Technique	#Assertions	FPs		FNs
					Total	%	
imsmart	1407	3	Daikon	21	2	9.52	1
			EvoSpex	4	1	25	1
beanbin	4784	5	Daikon	35	5	14.29	0
			EvoSpex	7	0	0	0
byuic	7699	7	Daikon	165	21	12.73	4
			EvoSpex	36	4	11.11	2
geo-google	20974	7	Daikon	93	30	32.26	0
			EvoSpex	10	3	30	4
templateit	3315	7	Daikon	37	4	10.81	3
			EvoSpex	20	0	0	2
water-simulator	9931	9	Daikon	39	3	7.69	9
			EvoSpex	18	3	16.67	9
dsachat	5546	9	Daikon	138	15	10.87	3
			EvoSpex	18	2	11.11	2
jmca	16891	9	Daikon	205	26	12.68	0
			EvoSpex	25	1	4	3
jni-inchi	3100	10	Daikon	122	12	9.84	2
			EvoSpex	50	1	2	4
bpmail	2765	11	Daikon	46	6	13.04	8
			EvoSpex	17	0	0	7
dom4j	42198	18	Daikon	166	27	16.27	7
			EvoSpex	25	2	8	10
jdbacl	28618	19	Daikon	115	17	14.78	10
			EvoSpex	80	3	3.75	8
jiprof	26296	20	Daikon	352	81	23.01	20
			EvoSpex	35	4	11.43	19
summa	119963	21	Daikon	273	67	24.54	6
			EvoSpex	62	5	8.06	5
corina	78144	22	Daikon	155	13	8.39	17
			EvoSpex	55	1	1.82	17
a4j	6618	23	Daikon	257	59	22.96	9
			EvoSpex	60	5	8.33	5
TOTAL		200	Daikon	2219	388	17.49	99
			EvoSpex	522	35	**6.70**	**98**

TABLE III
Comparing manually written contracts (in Eiffel verified classes) with postconditions inferred by EvoSpex.

Method	Eiffel Contracts	EvoSpex
Composite		
add_child(Composite c) : void	child added	✓
	c value unchanged	
	c children unchanged	
	ancestors unchanged	✓
DoublyLinkedListNode		
insert_right(DoublyLinkedListNode n) : void	n left set	✓
	n right set	
remove() : void	singleton	✓
	neighbors connected	
Map<K,V>		
count() : int	result is size	✓
extend(K k,V v) : int	key set	✓
	data set	✓
	other keys unchanged	
	other data unchanged	
	result is index	
remove(K k) : int	key removed	✓
	other keys unchanged	
	other data unchanged	
	result is index	
RingBuffer<G>		
count() : int	result is size	
extend(G a_value)	value added	✓
item() : G	result is first elem	
remove() : void	first removed	
wipe_out() : void	is empty	✓

TABLE IV
Inferring postconditions of synthesized collections.

High-level state	Method	High-level spec	EvoSpex
Polyupdate			
x : Bag<Int>	a(Integer y) : void	y added to x	✓
s : Int		y added to s if positive	
	sm() : Integer	result is s + sum of x	✓
Structure			
x : Int	foo() : Integer	result is x + 1	✓
	setX(Integer y)	x = y	✓
ListComp02			
Rs : Bag<R>	insert_r(R r) : void	r added to Rs	✓
Ss : Bag<S>	insert_s(S s) : void	s added to Ss	✓
	q() : Integer	result is the sum of Rs $\otimes$ Ss	
MinFinder			
xs : Bag<T>	findmin() : T	result is min of xs	✓
	chval(T x, int nv) : void	inner value of T is x	
MaxBag			
l : Set<Int>	get_max() : Integer	result is max of l	✓
	add(Integer x) : void	x added to l	✓
	remove(Integer x) : void	x removed from l	

OASIs assessment); the oracle deficiency analysis performed by OASIs is inherently incomplete, we cannot guarantee the truth of the remaining assertions.

As shown in Table II, in most of the case studies (13 out of 16), the percentage of false positives that our tool generates, when considering the total amount of produced assertions, is less than that produced by Daikon. Thus, comparing it with Daikon, and solely based on false positives, our assertions are significantly more precise. In fact, in a total of 200 methods analyzed, our technique had a 6.7% false positives, compared to the 17.49% of Daikon (an order of magnitude improvement). Moreover, the relationship between the number of produced assertions (in total, 522 of EvoSpex vs. the 2219 produced by Daikon) and the identified presence of false negatives, shows that our technique produces overall assertions of similar strength, with significantly fewer constraints. Daikon seems to make a more heavy used of specifically observed values in the assertions, leading to assertions that, while true within the provided test suite cases, are violated when considering larger scopes. Our algorithm is guided both by valid and invalid pre/post method states, giving it an advantage over Daikon, and explore a state space of candidate assertions that are less affected by specific values observed in executions. Regarding false negatives, both Daikon and our technique lead to assertions for which OASIs is able to identify false negatives (with our technique having a small margin of advantage in this respect). The conclusion is clear: the assertions obtained with both tools are weaker than the "strongest" postcondition, thus letting "pass" undetected some mutations of the analyzed methods (cf. how OASIs identifies false negatives [15]). Unfortunately, as discussed earlier, a problem with OASIs did not allow us to perform a more detailed comparison, based on the *number* of identified false negatives in each case. Nevertheless, by inspecting the obtained postconditions, as shown in Table I, it is apparent that our technique produces stronger assertions.

Regarding RQ2, the assertions that our technique can produce are close to those that may be defined by developers when manually specifying rich contracts. As Table III shows, our algorithm generated at least one of the exact properties defined in the original assertions for the Eiffel methods in 8 out of 11 cases. Similarly, as Table IV indicates, in 9 out of 12 methods we correctly identified at least one property of the corresponding postcondition. This confirms that our technique is capable of generating assertions that are actually true positives and are scope-independent. In the case of the remaining properties that the algorithm was not able to infer, these are specific assertions regarding the arguments, complex properties over sets that are beyond the assertions that the algorithm may produce, such as the "other keys unchanged" in the Map.extend routine, or are relatively complex arithmetic constraints such as the ones present in Cozy's ListComp02 and the RingBuffer methods (notice that our assertions concentrate on properties of reference-based fields rather than sophisticated arithmetic assertions). All the assertions produced for each one of the targeted methods in this evaluation can be found in the replication package site [1].

VI. Related Work

Assertions can be exploited for a wide variety of activities in software development, notably program verification [5], [8] and bug finding [35], [18], but also including program comprehension, software evolution and maintenance [30], and specification improvement [15], [31], among others. Thus, the problem of automatically inferring specifications from source code, and in general the problem of producing software oracles, has received increasing attention in the last few years [3]. Techniques for inferring specifications from source code, i.e., for *deriving* oracles, include approaches based on program executions, such as those reported in [7], [34], as well as some recent techniques based on machine learning [32], [33], [25]. Compared to the execution based approaches, our technique is guided both by valid and invalid executions (actually, pre/post method states); compared to the machine learning approaches, our technique concentrates on method postconditions, and produces interpretable assertion in standard assertion languages, as opposed to assertions encoded into artificial neural networks and other machine learning models. A closely related technique is that proposed in [7], with which we compare in this paper. Tools for automated test generation, notably EvoSuite [10], can produce assertions accompanying the generated tests. However, these assertions are scenario-specific, i.e., they capture properties particular to the generated tests, as opposed to our postconditions that attempt to characterize general method behaviors.

Our technique embeds a mechanism for test input generation, that follows a bounded exhaustive testing approach. As opposed to the previous mechanisms for generating bounded exhaustive suites, e.g., via tools like Korat [4] or TestEra [17], our technique generates bounded exhaustive suites from the program's API, rather than from an invariant specification. In this sense, our technique is more closely related to Randoop [26], replacing the random method selection in building test traces, with a systematic generation of *all* bounded method traces. The state matching mechanism we used in this paper is crucial in making this approach effective, but its discussion is beyond the scope of the paper. Besides producing valid method executions in the search of assertions, our technique also produces invalid program executions. The approach is based on mutating state. It is somehow related to the oracle assessment approach (for false negatives) implemented in the OASIs tool [15], although therein the authors mutate *programs* (source code), as opposed to mutating *state*. The idea of mutating state is used elsewhere, e.g., in [20], [25].

VII. Conclusion

The oracle problem has become a very important problem in software engineering, and within this context, oracle derivation or inference is particularly challenging [3]. In this paper, we have proposed an evolutionary algorithm for oracle inference, in particular for inferring method assertions in the form of *postconditions*. Our technique features various novel characteristics, including a mechanism for generating test inputs bounded exhaustively, from a component's API, and the definition of a genetic algorithm whose state space of candidate assertions includes rich constraints involving method parameters, return values, internal object states, and the relationship between pre and post method execution states. Our experimental evaluation shows that our tool is able to produce more accurate assertions (stronger contracts in the sense of [28], with the associated benefits described therein), with a total of 6.70% of false positives, compared to the 17.49% of false positives of related techniques, for a set of randomly selected methods from a benchmark of open source Java projects. Furthermore, our evaluation shows that our technique is able to infer an important part of rich program assertions, taken from a set of case studies involving contracts for program verification and synthesis.

This work also opens several lines for future work. On one hand, our genetic algorithm uses a finite set of genetic operators, in particular the ones used for mutation; extending the set of operators and exploring new ones may be necessary to increase the scope of properties that the algorithm may produce, especially when dealing with more sophisticated programs. Fitness functions in genetic algoritms play a crucial role in the quality of the solutions; adapting the fitness function of our algorithm in order to prioritize general aspects of method postconditions may considerably improve our results. Our experiments were based on the use of a variant of random generation for the production of bounded exhaustive test suites. Using alternative test suite generation approaches such as fully random generation may allow us to produce different postconditions. The existence of false negatives for our produced postcondition assertions also opens lines of improvement for our inference mechanism.

References

[1] Evospex site. https://sites.google.com/view/evospex, 2020.

[2] Mike Barnett. Code contracts for .net: Runtime verification and so much more. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 16–17. Springer, 2010.

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In Phyllis G. Frankl, editor, *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, pages 123–133. ACM, 2002.

[5] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and esc/java2. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363. Springer, 2005.

[6] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. Reassert: Suggesting repairs for broken unit tests. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 433–444. IEEE Computer Society, 2009.

[7] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[8] Manuel Fähndrich. Static verification for code contracts. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2010.

[9] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

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

[11] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, 2014.

[12] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[13] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[14] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[15] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 247–258. ACM, 2016.

[16] Pankaj Jalote. *An Integrated Approach to Software Engineering, Third Edition*. Texts in Computer Science. Springer, 2005.

[17] Shadi Abdul Khalek, Guowei Yang, Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Testera: A tool for testing java programs using alloy specifications. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Lawrence, KS, USA, November 6-10, 2011*, pages 608–611. IEEE Computer Society, 2011.

[18] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development = test driven development - writing test cases. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 425–434. ACM, 2007.

[19] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 958–968, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Muhammad Zubair Malik, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 190–199. IEEE Computer Society, 2011.

[21] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[22] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[23] Bertrand Meyer. Design by contract: The eiffel method. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, page 446. IEEE Computer Society, 1998.

[24] Facundo Molina, César Cornejo, Renzo Degiovanni, Germán Regis, Pablo F. Castro, Nazareno Aguirre, and Marcelo F. Frias. An evolutionary approach to translate operational specifications into declarative specifications. In Leila Ribeiro and Thierry Lecomte, editors, *Formal Methods: Foundations and Applications - 19th Brazilian Symposium, SBMF 2016, Natal, Brazil, November 23-25, 2016, Proceedings*, volume 10090 of *Lecture Notes in Computer Science*, pages 145–160, 2016.

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

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007.

[27] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. Specifying reusable components. In Gary T. Leavens, Peter W. O'Hearn, and Sriram K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings*, volume 6217 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2010.

[28] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 262–271. IEEE Computer Society, 2013.

[29] Pablo Ponzio, Valeria S. Bengolea, Simón Gutiérrez Brida, Gastón Scilingo, Nazareno Aguirre, and Marcelo F. Frias. On the effect of object redundancy elimination in randomly testing collection classes. In Juan Pablo Galeotti and Alessandra Gorla, editors, *Proceedings of the 11th International Workshop on Search-Based Software Testing, ICSE 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 67–70. ACM, 2018.

[30] Manoranjan Satpathy, Nils T. Siebel, and Daniel Rodríguez. Assertions in object oriented software maintenance: Analysis and a case study. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 124–135. IEEE Computer Society, 2004.

[31] Todd W. Schiller and Michael D. Ernst. Reducing the barriers to writing verified specifications. In Gary T. Leavens and Matthew B. Dwyer, editors, *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*, pages 95–112. ACM, 2012.

[32] Seyed Reza Shahamiri, Wan Mohd Nasir Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information & Software Technology*, 53(7):774–788, 2011.

[33] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design*, 48(3):235–256, 2016.

[34] Anthony J. H. Simons. Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Autom. Softw. Eng.*, 14(4):369–418, 2007.

[35] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In Bernhard Beckert and Reiner Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

[36] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Nadia Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015, London, UK, April 11-18, 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 566–580. Springer, 2015.