Training Binary Classifiers as Data Structure Invariants

Facundo Molina*†, Renzo Degiovanni§, Pablo Ponzio*†, Germán Regis*, Nazareno Aguirre*†, Marcelo Frias†‡
*Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Argentina
†Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
‡Dept. of Software Engineering, Buenos Aires Institute of Technology, Argentina
§SnT, University of Luxembourg, Luxembourg

***Abstract*—We present a technique that enables us to distinguish valid from invalid data structure objects. The technique is based on building an artificial neural network, more precisely a binary classifier, and training it to identify valid and invalid instances of a data structure. The obtained classifier can then be used in place of the data structure's invariant, in order to attempt to identify (in)correct behaviors in programs manipulating the structure. In order to produce the valid objects to train the network, an assumed-correct set of object building routines is randomly executed. Invalid instances are produced by generating values for object fields that "break" the collected valid values, i.e., that assign values to object fields that have not been observed as feasible in the assumed-correct program executions that led to the collected valid instances. We experimentally assess this approach, over a benchmark of data structures. We show that this learning technique produces classifiers that achieve significantly better accuracy in classifying valid/invalid objects compared to a technique for dynamic invariant detection, and leads to improved bug finding.**

I. Introduction

Given the current advances in automated program analysis, it is now possible to efficiently produce large sets of program inputs, as well as examining very large sets of program executions [8], [14], [29], [30], but effectively deciding whether the behavior of software is correct or not remains a problem in this context, that mainly depends on the provision of software specifications (i.e., *specified* oracles in the terminology of [6]). Various researchers have acknowledged this issue, and developed techniques that are able to *derive* specifications that are implicit in the software being assessed. Examples of techniques that derive specifications implicit in code are Daikon [13], JWalk [38], and related tools [10], [37]. Daikon produces a set of candidate properties from a program definition, and infers *likely* invariants by observing program executions, and checking which of the candidate properties were not "falsified" (violated) by any execution [13]. JWalk also infers properties from program executions, but it does so by interacting with the user to confirm "learned" observations, to incrementally produce a test oracle. While both tools are very powerful, they have limitations. Daikon is limited to relatively simple program properties, and complex structural constraints such as acyclicity are beyond the scope of the technique [13]. JWalk also shares this limitation, and the learned oracles are more "scenario-specific", i.e., closer to test assertions, than those produced by Daikon [38].

In this paper, we deal with the specification inference problem, in a way similar in motivation to techniques like Daikon and JWalk, but specifically targeting object validity classifiers for complex objects, like *class invariants* [25], [22] for data structures. Our technique differs from the mentioned ones in several respects. Firstly, it is based on the use of artificial neural networks [34] for *learning* classifiers from valid and invalid objects, obtained from program executions. This implies that learned classifiers are not formed by explicit constraints that the user can manually inspect, as opposed to traditional class invariants; but at the same time, our classifiers are able to recognize more complex data representation properties, in particular structural properties of heap-allocated linked data, that other techniques cannot handle. Secondly, we concentrate on object classifiers for bug detection, so our aim is to produce classifiers that tend to "over-approximate" data structure invariants, i.e., classifiers that identify invalid objects with a very high precision and recall (this poses the challenge of not falling into *naive* classifiers that simply identify most instances as "positive"). Thirdly, as opposed to other techniques that infer software properties from dynamic information, our approach requires inferring such properties from *positive* as well as *negative* cases (notice that both Daikon and JWalk only consider *positive* cases, since they explore executions of supposedly correct software to infer likely program properties). Positive cases are those that the classifying function we want to learn should satisfy, while negative ones are *invalid* instances, i.e., objects for which the classifier should return false. In order to produce positive cases, we assume correct a set of object builders, e.g., constructor and insertion routines, and use these to produce programs that build valid instances of the data structure of interest. This is a standard approach in various contexts, in particular in some program verification and test generation techniques [28], [24], [20] (Daikon and JWalk in essence also work under this assumption). On the other hand, negative inputs are produced as follows. As valid instances are generated, the observed *extensions* of class fields, composed of all observed values for each field, are collected; then, invalid instances are generated by exploiting these field extensions, by producing new instances in which a field is

This work was partially supported by ANPCyT PICT 2015-2341, 2015-0586, 2015-2088, 2017-1979; and by the INTER/ANR/18/12632675/SATOCROSS.

given a value that is either outside the corresponding field extension (and thus guaranteeing that is indeed a new object), or within the extension, but whose value has not been seen in combination with the rest of the instance being altered, within the valid ones.

We evaluate our learning approach in several ways. First, we assess the adequacy of our approach to generate invalid objects, analyzing how many of our assumed-invalid produced instances are indeed invalid (violate a provided invariant). Second, we take a benchmark of data structures for which class invariants and object builders (constructors and insertion routines) are provided; we generate object classifiers using the provided builders, and evaluate their precision and recall against the corresponding invariants on valid and invalid inputs, as is customary in the context of automated learning [34]. In this context, we compare our technique with Daikon [13], a tool for dynamic invariant discovery. Finally, we also compare our object classifiers with invariants produced by Daikon, in bug finding through test generation, for a number of case studies involving data structures, taken from the literature: `schedule` from the SIR repository [12], an implementation of n-ary trees in the ANTLR parser generator, a red-black tree implementation of integer sets introduced in [42], binary search trees and binomial heaps from the evaluation performed in [14], and fibonacci heaps from the graphmaker library [1]. Our experiments show that our mechanism for producing invalid inputs is effective, and that the learned classifiers achieve significantly better accuracy in identifying valid/invalid instances, compared to Daikon (high precision and recall both in negative and positive cases). Moreover, learned classifiers allow a test generation tool to catch bugs, if classifiers are used in place of invariants, that the same tool cannot detect if the invariants produced by Daikon are directly used instead, indicating that learned classifiers are not trivial.

II. Background

A. Class Invariants

One of the keys of object orientation is the emphasis that this programming paradigm puts into *data abstraction* [22]. Indeed, the concept of *class* is a useful, direct mechanism to define *new* datatypes, that extend our programming language's set of predefined types with custom ones that allow us to better capture or deal with concepts from a particular problem domain. A class defines the type of a set of objects, whose internal representation is given by the fields that are part of the class definition. This implementation of a new data abstraction in terms of provided data structures is often accompanied by a number of *assumptions* on how the data structure should be manipulated, that capture the intention of the developer in his chosen representation. These assumptions are often implicit, since they are not a necessary part of the definition of the data representation, in most programming languages [22]. Consider, as an example, a representation of sequences of integers, implemented using heap-allocated singly-linked lists. The classes involved in this data abstraction are shown in Figure 1. Clearly, solely from the class's fields one cannot infer the *intention* of the developer in the representation. Whether these lists are going to be arranged cyclicly, acyclicly, with or without sentinel node, with reserved nodes for some special information, etc., are all issues that are not an explicit part of the class's internal definition, although, of course, one may infer such information from how the internal representation is used by the methods of the class.

```java
public class SinglyLinkedList {
  private Node head;
  private int size;
  ...
}

public class Node {
  private int value;
  private Node next;
  ...
}
```

Fig. 1. Java classes for singly linked lists.

A *class invariant* [25], or *representation invariant* [22], is a predicate *inv* that, given an object o of class C, states whether o is a valid representation of the concept that C captures or not. Equivalently, *inv* can be described as a boolean classifying function that decides whether an object o satisfies the representation assumptions in the implementation of C. For instance, assume that the programmer's intention with singly linked lists is to represent sequences of integers using acyclic linked lists, with a dummy (sentinel) node [23], where `size` must hold the number of elements in the sequence, i.e., it must coincide with the number of non-dummy nodes in the list. Samples of valid lists, under this assumption, are shown in Figure 2. The *invariant* for `SinglyLinkedList` should then check precisely the above constraint, i.e., it must be satisfied by all instances in Fig. 2, and must not hold for, say, cyclic lists, lists where `head` is null, or where the dummy node has a `value` different from 0, or where the `size` field does not hold the number of non-dummy nodes in the list.

Class invariants can be captured using different languages. The Eiffel programming language [26], in particular, includes built-in support for expressing class invariants as assertions under a specific `invariant` clause. Other languages support design-by-contract [25] and assertions, invariants among them, via special languages and libraries, such as JML [9] for Java and Code Contracts [5] for .NET. Languages such as Alloy [17] have also been employed to express class invariants, as done, e.g., in [19]. Finally, various programming methodologies (e.g., [22]) and analysis tools (e.g., [29], [8]) can exploit class invariants expressed as Java predicates, i.e., via boolean methods that check internal object consistency. Figure 3 shows the class invariant for our singly linked list example, expressed as a Java predicate.

In this paper, we will be capturing close approximations of class invariants for data structures through artificial neural networks, that, as a consequence, will not be formed by

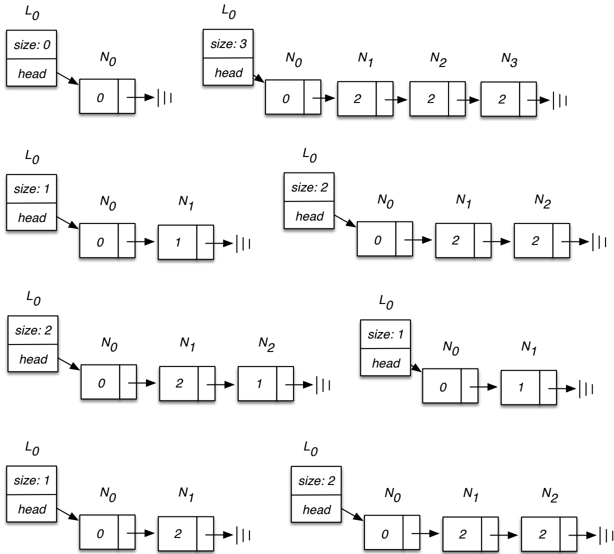

Fig. 2. Valid acyclic singly linked lists with dummy node.

```
public boolean repOK() {
  if (this.head==null) return false;
  if (this.head.value!=0) return false;
  int expectedSize = this.size+1;
  int currNode = this.head;
  while (expectedSize>0 && currNode!=null) {
    expectedSize--;
    currNode = currNode.next;
  }
  return (expectedSize==0 && currNode==null);
}
```

Fig. 3. Java invariant for acyclic singly linked lists.

explicit constraints, as in the example in Fig. 3.

B. Field Extensions

Various tools for program analysis that employ SAT solving as underlying technology adopt a *relational* program state semantics (e.g., [11], [14]). In this semantics, a field `f` at a given program state is interpreted as the set of pairs $\langle id, v\rangle$, relating object identifier *id* (representing a unique reference to an object o in the heap) with the value v of the field in the corresponding object at that state (i.e., $o.\mathtt{f} = v$ in the state). Then, each program state corresponds to a set of (functional) binary relations, one per field of the classes involved in the program. For example, fields `head` and `next` of a program state containing the singly linked list at the top right of Fig. 2 is represented by the following relations:

$$\begin{array}{lcl} \mathtt{head} & = & \{\langle L0, N0\rangle\} \\ \mathtt{next} & = & \{\langle N0, N1\rangle, \langle N1, N2\rangle, \langle N2, N3\rangle, \langle N3, null\rangle\} \end{array}$$

Notice that in the lists in Fig. 2, we have consistently identified the objects involved in each example. Although in this example it is not evident, due to the linear nature of the structure, we choose to identify each object by the order in which it is visited in a breadth-first traversal of the corresponding structure, using different identifier sets for different classes (*Li* for lists, *Ni* for nodes, etc.). Adopting this notion of object identifier allows us to have a canonical (isomorphism-free [18]) representation for each structure shape (a similar symmetry breaking approach is also present in other approaches, e.g., [8]).

The notion of *field extension* is associated with a *set of objects* or program states. It essentially corresponds to joining the above-described relational interpretation of fields, for various objects or program states. For instance, for the set of lists in Fig. 2, the extensions for fields `head` and `next` are the following:

$$\begin{array}{lcl} \mathtt{head} & = & \{\langle L0, N0\rangle\} \\ \mathtt{next} & = & \{\langle N0, N1\rangle, \langle N0, null\rangle, \langle N1, N2\rangle, \langle N1, null\rangle, \\ & & \langle N2, N3\rangle, \langle N2, null\rangle, \langle N3, null\rangle\} \end{array}$$

This notion of field extension is related to the concept of (upper) *bound* in KodKod [39], used with the purpose of optimizing the relational representation of fields in Alloy analyses. Technically, field extensions are *partial* bounds, in the KodKod sense.

When field extensions are built from *valid* objects, they capture the set of values for fields that have been identified as being *feasible*, in the sense that at least one observed structure admits each value in the corresponding extension. We will use these extensions to attempt to build *invalid* objects, e.g., considering pairs that are *not* in field extensions. This demands defining a *complement* for the field extensions, for which we have to consider domains and codomains for these relations. This is typically achieved in the context of bounded analysis by a notion of *scope*, in the sense of [17]. The *scope*, often simplified as a number k, defines a *maximum* number of objects for each class C_i, and finite ranges for basic datatypes (usually as a function of k). For a given scope k, the set of *all* possible structures or instances is composed of all possible assignments of values within the scope, for fields of the scope's objects, respecting the fields' types, and thus provides us with a notion of universe for the field extensions. For instance, if the scope for our analysis is 1 list, up to 5 nodes, size in the range 0..5 and values in the range 1..5, then pair $\langle N3, N4\rangle$ is in the complement of the extension of `next`, whereas if we instead consider up to 4 nodes, it is not.

C. Feed-Forward Artificial Neural Networks

Artificial Neural Networks (ANNs) are a state-of-the-art technique underlying many machine learning problems. These algorithms offer a number of advantages, including their remarkable ability to implicitly detect complex nonlinear relationships in data, that are otherwise very complex to be noticed. An ANN is composed of a group of different nodes, called *neurons*, connected by directed weighted links. Each neuron is a simple computational unit that computes a weighted sum of its inputs, and then applies an activation function g to produce an output, that will be an input of another neuron. Figure 4(a) depicts the structure of a neuron.

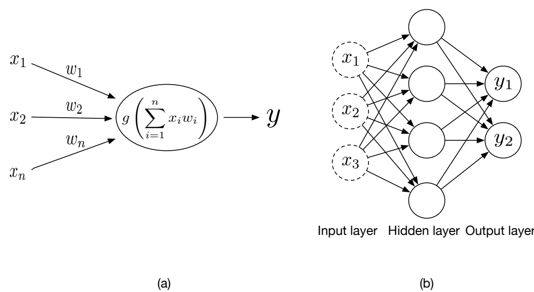

Fig. 4. An artificial neuron, and a feed-forward neural network with a single hidden layer.

Neurons can be disposed respecting certain network architectures. In particular, in a *feed-forward neural network*, neurons are typically organized in layers. Each neuron in a layer has a link to each neuron of the next layer, forming a directed acyclic graph. The first layer is the *input* layer, and its neurons receive a single value as an input, and simply replicate the received value through their multiple outputs, to the next layer. The final layer is the output layer and its neurons produce the output of the network computation. In addition to these two layers there can be any number of intermediate layers, called *hidden* layers. Often, neural networks will have one hidden layer, since one layer is enough to approximate many continuous functions [34]. The structure of a feed-forward neural network is depicted in Figure 4(b).

The behavior of a neural network can be dynamically altered, by changing the weights associated with the directed links in the network, or by modifying some of the neural network so-called *hyperparameters*, such as the number of neurons in the hidden layer. Assume that we want an artificial neural network to approximate a function f, and that we can characterize the inputs of f as a vector of values (to be fed in the input layer). Provided that one has a set of inputs for which the desired output is known (i.e., a set of known input-output pairs for f), one can *train* an artificial neural network to approximate function f, by analyzing the difference between the expected output and the output obtained from the network for a known input, and producing (slight) changes to the weights so that, if the network would be fed with the same input again, its output would be "closer" to the expected output [34] (a mechanism that is often employed for this task is *backpropagation*). This approach is known as *supervised learning*, and when the output has only two possible values, it is a *binary classification* problem. The problem we deal with in this paper, namely the approximation of a class invariant to classify valid vs. invalid data structure objects, clearly falls in the category of binary classification: we want to learn a function f that sends each valid instance to *true*, and each invalid instance to *false*. We will then need both valid and invalid instances, to appropriately train a neural network to learn a class invariant. Section IV describes the details of our technique.

III. An Illustrating Example

Let us provide an overview of our approach, through an illustrating example. Consider the Java implementation of sequences of integers, over singly linked lists, discussed earlier in this paper. We would like to check that this list implementation behaves as expected. This includes guaranteeing that all public constructors in `SinglyLinkedList` build objects that satisfy the previously stated *class invariant* [22], and public methods in the class (that may include various methods for element insertion, deletion and retrieval) preserve this invariant. That is, they all maintain acyclicity of the list, with its number of nodes from `head.next` coinciding with the value of `size`, etc. If we had this invariant formally specified, we may check that it is indeed preserved with the aid of some automated analysis tools, e.g., some run-time assertion checker as that accompanying the JML toolset [9], or a test generation tool like Randoop [29]. But getting these invariants right, and specifying them in some suitable language, even if the language is the same programming language of the program implementation, is difficult, and time consuming, and one does not always have such invariants available.

We would then like to *approximate* a class invariant *inv* : $C \rightarrow$ *Bool* using a neural network, from the implementation of C. In order to do so, we need to *train* the neural network with a sample for which we know the correct output. In other words, we need to train the neural network with a set of *valid* instances, i.e., objects that satisfy the invariant (or, equivalently, for which the invariant should return true), as well as a set of *invalid* instances, i.e., objects that do not satisfy the invariant (for which the invariant should return false). In order to do so, we will ask the user to provide a subset of the class's methods, that will be used as assumed-correct *builders*, i.e., as methods that allow us to build correct instances. For instance, in our example the user may trust the implementation of the constructor and the insertion routine, and thus all objects produced with these methods are assumed correct. Using these builders we can construct assumed-correct instances, by using, e.g., an automated test generation tool such as Randoop. A particular set of valid instances that we may obtain from this process could be the objects in Fig. 2.

Building *invalid* instances is more difficult. We may ask the user to manually provide such cases, but the number of objects necessary to appropriately train the network would be large, and thus this approach would seriously limit the efficacy of the approach. We may also ask the user to provide methods to build incorrect objects, but this would mean extra work (it is not something that the user has already at hand), and providing such methods is not, in principle, easy to do. Instead, our approach is based on the use of *field extensions*. We proceed as follows. We have already run some input generation tool using the builders for some reasonable amount of time, and have obtained a set of valid objects of class `SinglyLinkedList`. From these objects we can compute the field extensions for each field of the data structure. The extensions for `head` and `next` are shown in the previous section; the extensions for

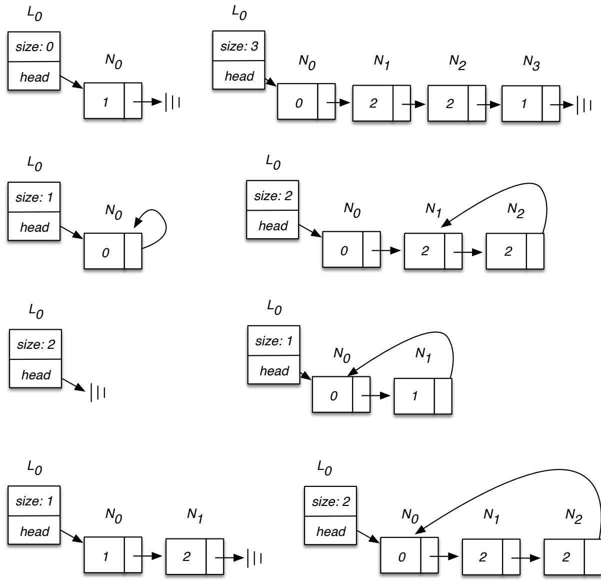

Fig. 5. Potentially invalid list structures, built by breaking field extensions.

`size` and `value` are the following:

$$\begin{array}{rcl} \texttt{size} & = & \{\langle L0,0\rangle,\langle L0,1\rangle,\langle L0,2\rangle,\langle L0,3\rangle\} \\ \texttt{value} & = & \{\langle N0,0\rangle,\langle N1,1\rangle,\langle N1,2\rangle,\langle N2,1\rangle,\langle N2,2\rangle,\langle N3,2\rangle\} \end{array}$$

We will build potentially invalid instances by changing field values in valid structures. We have two possibilities for a change in a given object field: we can go "outside" the extensions, i.e., assign a value to the field that has not been observed in any of the built valid structures (and thus guaranteeing that is a *new* object), or go "inside" the extensions, i.e., assign a value different from the original, but within the feasible "observed" ones for the field. In the former case, we need to define a *scope*, so that being "outside" the extension can be precisely defined. Assume, for the sake of simplicity, that we arbitrarily define the following scope: exactly one list object (*L0*), 4 nodes (*N0*, ..., *N3*), `size` in the range 0..3 and `value` in the range 0..2. Now, we can build (allegedly) invalid instances by changing, for each valid structure, a value of some reachable field to some different value, outside the corresponding extension, or within it but different from the original. In Fig. 5 we show a sample of potentially invalid instances obtained from those in Fig. 2 using this mechanism.

A few issues can be noticed from these examples. First, there is no guarantee that we actually build invalid instances with this process. The top right object in Fig. 5 is in fact a valid case. Artificial neural networks are however rather tolerant to "noise" during learning, so as long as the number of spurious invalid objects is low, learning may still work well. Second, what we are able to build as potentially invalid instances greatly depends on what we produced as valid ones, and what we define as the *scope*. Both issues are critical, and we discuss these further in the next section. Third, the specific mechanism for choosing a value within the corresponding field extension or outside it, is not specified. One may randomly choose on which direction to go, and with which proportion to go inside or outside the extensions. Intuitively, going "outside" the field extensions has better chances of producing invalid structures.

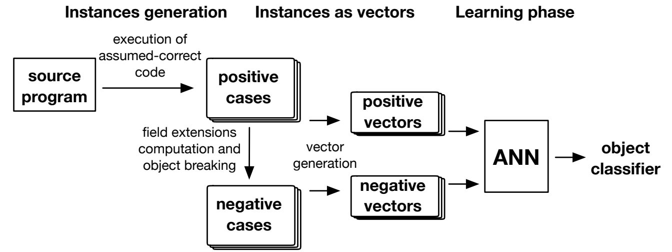

Fig. 6. An overview of the technique

IV. The Technique

Let us now present in more detail our approach to approximate class invariants using artificial neural networks. The technique, depicted in Fig. 6, has three main steps: *(i)* automatically generating valid and invalid data structure objects, *(ii)* representing objects of the class (both valid and invalid) as numerical vectors, to be able to feed these to a neural network, and *(iii)* building an artificial neural network, and training it with the produced valid and invalid objects, to learn to classify data structure instances as valid or invalid.

A. Generating Instances for Training

Assuming that we are given a class C for which we want to learn to classify valid vs. invalid data structure objects, our first step consists of generating instances that must and must not satisfy the *intended* invariant, respectively. As we mentioned before, our first assumption is that a set m of methods of C is identified as the assumed-correct *builders* of the class, i.e., a set of methods whose implementations are assumed correct, and that thus can be used to build valid instances. This assumption is in fact rather common in verification and test generation environments, that produce instances from a class's public interface (see, e.g., [20], [28], [24]). Our second assumption is that a notion of *scope* is provided (see previous section for an intuition of this requirement). The scope provides a defined domain for each field of each class involved in the analysis, and thus provides a domain where to search for field values when building potentially invalid objects from valid ones. The scope is not only relevant in defining a finite universe to compute the complement of field extensions (useful in building invalid instances); it also bounds the instances to be considered, allowing us to characterize them as fixed-size vectors (see next subsection).

In order to build valid instances, any input generation technique that can produce objects from a class interface, is suitable, including model checking based ones [20], [15], and random generation [29], [27], for instance (in our experiments

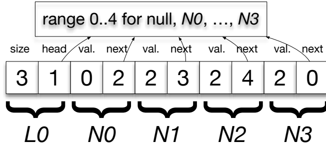

Fig. 7. Instance vector for a Singly Linked List example.

we will use random generation). Using the produced valid instances, we compute field extensions and generate potentially invalid instances by modifying valid ones, as follows: given a valid instance c, an object o reachable from c and a field $\mathtt{f}$ in o, we change the value of $o.\mathtt{f}$ to either a value within the extension of $\mathtt{f}$ with respect to o, or outside it but within the scope. The latter is favored since it exploits field extensions guaranteeing that a new object is constructed. From each valid structure, we produce as many invalid objects as object fields are reachable in the structure, using the above procedure to change a single object field in each case. Choosing to go within extensions or outside them can be done randomly. In our experiments, we change an object field going within the extensions with a probability of 0.2, and outside these with a probability of 0.8. The rationale for the selection is based on experimentation; we tried, for small scopes, different probabilities (and these were the ones that achieved better performance). When an object is changed, nodes are "relabeled" to preserve the breadth-first canonical ordering mentioned in section 2. Finally, we discard any potentially invalid object generated with the above procedure, that is also within the set of valid objects, so that there is no intersection between the valid and invalid objects used for training the network.

The generation technique for valid inputs is closely related to how the scope is chosen. Some techniques require the scope *a priori* for generation (e.g., [15]), while in others the scope can be derived from the generated instances, e.g., looking at the largest produced object, or the range of produced values. Intuitively, the scope should be at most "slightly loose" with respect to the field extensions corresponding to the generated valid instances, in the sense that, when building invalid objects, it should prevail to form field associations that are not part of any valid object, but that involve values that are part of valid objects (e.g., the association $N0.\mathtt{next} = N0$ in acyclic linked lists). In our experiments, we chose the scope a priori, and discarded any randomly generated object that lied outside the scope.

B. Representing Instances as Vectors

Artificial neural networks receive as inputs vectors of values, which are in general restricted to *numeric* types. While for some datatypes an encoding is direct (e.g., characters, enumerated types, tuples of basic types and strings of a maximum length), for objects of an arbitrary class C it is less straightforward. In order to encode object states as vectors, we adopt the candidate vector format of Korat [8]. Given a scope k, that defines ranges for numeric types and maximum number of instances for reference types, any instance o of class C within scope k can be represented by a vector containing a cell for each field $\mathtt{f}$ of each object o' reachable from o. The domain of each cell is a range of natural numbers, where each value uniquely identifies an object/value within the scope, for the corresponding field. For instance, given a scope of exactly one list object, 4 nodes, size in the range 0..3 and value in the range 0..2 for the singly linked list example, the top-right list of Fig. 2 is represented by the instance vector shown in Fig. 7.

C. Building and Training the Neural Net

The vectors representing the positive and negative instances form the *training set* that we feed the network with. The network that we build in order to learn to classify these instances as positive or negative is a *feed-forward artificial neural network*. Firstly, assuming that the size of the current vectors is n, the input layer will contain n input neurons, each receiving a position of the vector, and the output layer will always have 1 neuron since our classification problem involves two different classes. Only one hidden layer is used. The number of hidden units (i.e., number of neurons) in the hidden layer is a hyperparameter whose value can impact the network's effectiveness, and can be set automatically. Known algorithms to automatically select hyperparameters are *grid search* and *random search*; we use random search due to its ability to reduce the validation set error faster than grid search [7]. The parameter for the number of hidden layer units takes values in the range $[2, 100]$. Another hyperparameter that is usually considered, and we consider in our work, is the regularization term (a penalty coefficient that affects the neural network's learning process); values for this parameter were taken in the range $[-5, 3]$, evenly spaced in a logarithmic scale, as is customary in various domains. We launched 10 random combinations of hyperparameter values and then selected the combination with the best performance, to determine the final network architecture. As we show in the following experimental evaluation, the level of precision that we achieved did not demand further tuning of the neural network's hyperparameters.

We use the Multi-layer Perceptron neural network implementation of the python `scikit-learn` package [31].

V. Evaluation

The evaluation of our technique is driven by the following research questions:

RQ1 *Is the technique for building potentially invalid instances suitable for this task?*

RQ2 *How precise is the neural network in classifying valid/invalid objects?*

RQ3 *Do our learned object classifiers help in capturing relevant information on expected class behaviors, that can lead to improved bug finding?*

To evaluate RQ1, we need to assess every potentially invalid instance that we build with our technique based on field extensions, to check if it is indeed invalid. Our experiment

TABLE I
NUMBER OF SPURIOUS INVALID OBJECTS GENERATED BY EXPLOITING FIELD EXTENSIONS.

Scope	Instances			
	Total	Positive	Negative	False Negative
Singly Linked List				
6	15731	2486	13245	0 (0%)
7	41323	7867	33456	0 (0%)
8	81199	16416	64783	0 (0%)
Singly Sorted List				
6	10546	240	10306	239 (2%)
7	34779	830	33949	760 (2%)
8	85708	2577	83131	1966 (2%)
Doubly Linked List				
6	48190	5155	43035	0 (0%)
7	85955	9381	76574	0 (0%)
8	136893	14801	122092	0 (0%)
Binary Tree				
6	7255	193	7062	0 (0%)
7	21647	567	21080	0 (0%)
8	55953	1401	54552	0 (0%)
Binary Search Tree				
6	19137	692	18445	365 (2%)
7	53006	2198	50808	840 (2%)
8	112031	5368	106663	1700 (2%)
Red Black Tree				
6	8700	165	8535	131 (2%)
7	27358	464	26894	313 (1%)
8	73422	1323	72099	690 (1%)
Binomial Heap				
6	3243	407	2836	70 (2%)
7	2830	380	2450	58 (2%)
8	124563	12889	111674	5213 (5%)

proceeded as follows. We took all case studies accompanying the Korat distribution [8] (available in [2]), that involve various data structures of varying complexities for which class invariants are provided (notice that Korat requires class invariants expressed as Java predicates for test input generation [8]). We extended each class with a set of builders (e.g., constructor and insertion routines), and used Randoop to produce valid instances with these builders, disregarding the invariant provided with Korat, for different scopes. For each case we ran Randoop with 100 different seeds, 10 seconds each, and collected all produced objects (a total of over 16 minutes input generation time, per case study). We then used our technique based on exploiting field extensions to produce potentially invalid objects, and checked for their spuriousness using the corresponding invariant in the Korat distribution. The results are summarized in Table I. This Table shows, for each case study and various scopes, the total number of produced objects, distinguishing between valid and invalid objects, and for the latter the number of spurious cases (i.e., objects built by breaking objects using field extensions that actually satisfied the corresponding original `repOK` provided with Korat). Due to space reasons, we show a sample of the structures and scopes. Further details can be found in the experiments site [3].

To evaluate RQ2, we first performed the following experiment. We again took Korat case studies with our provided object builders, and ran, for each class C, our technique, obtaining a corresponding object classifier I'. In this step we used the object builders, and disregarded the provided class invariants. We then used for each class C its Korat invariant I to generate *all* valid objects within a given scope k (i.e., all valid objects of size at most k), using the Korat tool [8]. Moreover, we also collected the objects that Korat produced in the search for valid objects, that were deemed invalid (i.e., that did not satisfy the corresponding invariant). This collection of valid and invalid objects was used for measuring the precision and recall in object classification, as separate measures for valid and invalid objects. This experiment was performed for increasingly larger scopes, as long as the number of instances did not exceed 1 million. The results are summarized in Table II. For each case study and scope we report: the number of valid and invalid objects used for training, as well as the training time (notice that each training set was generated with Randoop and the object builders); the size of the sample used for measuring recall/precision, provided as total number of valid/invalid objects (notice that these were generated using Korat); the number of objects correctly and incorrectly classified (tp/tn for true positive/negative, fp/fn for false positive/negative), and the corresponding precision and recall, given as percentages. Notice that, since the training and evaluation sets are generated independently, some structures used in training may also appear in the evaluation set. We have indicated between parentheses the number of *new* positive and negative instances in the evaluation set, i.e., those structures that have been used for evaluation but that were not part of the corresponding training sets. Again, a sample of the structures and scopes is shown; more information can be found in the experiments site [3].

In order to have a reference of the accuracy of our approach, we compare our learned object classifiers with invariants generated using Daikon [13]. The process we followed to produce invariants with Daikon is the following. For each case study, we took the same tests used as a starting point for learning object classifiers with our approach, and ran Daikon using those. Daikon produced a list l of likely invariants, which in all cases included invalid properties (properties that were true of the provided tests but were not true in the general case for the corresponding structure). From l, we produced a list l', by manually filtering invalid properties (i.e., properties that do not hold for all structures). We measured the precision and recall of the obtained Daikon invariants, for the same objects used to measure precision/recall of our technique. The results are summarized in Table III.

RQ3 is the only research question that does not demand a provided class invariant for assessment. To evaluate it, we took buggy implementations of data structures from the literature: the `scheduler` implementation from the SIR repository [12], an implementation of n-ary trees that is part of the ANTLR parser generator, implementations of routines of a set of integers, over red black trees, with seeded bugs, presented in [42], binary search trees and binomial heaps used in the empirical evaluation in [14] containing one real bug each, and a fibonacci heap implementation taken from [1], containing a real bug. For each case study, we took a set of builders, and generated tests with Randoop from which we learned an

TABLE II
PRECISION AND RECALL OF CLASSIFYING TECHNIQUE ON COMPLEX DATA STRUCTURES.

Scope	Training Instances		Time	Testing Positive Instances					Testing Negative Instances				
	Positive	Negative	sec.	*total*	*tp*	*fp*	*precision(%)*	*recall(%)*	*total*	*tn*	*fn*	*precision(%)*	*recall(%)*
Data Structure: **Singly Linked List**													
6	2486	13245	3,96	3906 (1420)	3906	5	99,8	100,0	42778 (37988)	42773	0	100,0	99,9
7	7867	33456	26,65	55987 (48120)	55897	7	99,9	99,8	725973 (714149)	725966	0	100,0	99,9
8	16416	64783	75,66	960800 (944384)	960800	1	99,9	100,0	10000000 (9982384)	9999999	0	100,0	99,9
Data Structure: **Singly Sorted List**													
6	240	10306	24,51	252 (12)	246	5	98,0	97,6	96820 (94883)	96815	6	99,9	99,9
7	830	33949	133,02	924 (94)	873	2	99,7	94,4	1617109 (1610806)	1617109	51	99,9	100,0
8	2577	83131	453,14	3432 (855)	3142	57	98,2	91,5	10000000 (9988045)	9999943	290	99,9	99,9
Data Structure: **Doubly Linked List**													
6	5155	43035	18,38	55987 (50832)	55987	8	99,9	100,0	465917 (458009)	465909	0	100,0	99,9
7	9381	76574	240,68	960800 (951419)	960800	8	99,9	100,0	8914750 (8901434)	8914742	0	100,0	99,9
8	14801	122092	978,39	1000000 (999262)	999999	1	99,9	99,9	10000000 (9998920)	9999999	0	100,0	99,9
Data Structure: **Binary Tree**													
6	193	7062	12,81	197 (4)	197	0	100,0	100,0	4638 (3686)	4638	0	100,0	100,0
7	567	21080	162,47	626 (59)	524	3	99,4	83,7	17848 (15388)	17845	102	99,4	99,9
8	1401	54552	94,08	2056 (655)	2054	7	99,6	99,9	68810 (63224)	68803	2	99,9	99,9
Data Structure: **Binary Search Tree**													
6	692	18445	99,15	731 (39)	720	1621	30,7	98,4	61219 (59369)	59598	11	99,9	97,3
7	2198	50808	437,42	2950 (752)	2395	6105	28,1	81,1	468758 (466278)	462653	555	99,8	98,6
8	5368	106663	574,27	12235 (6867)	7185	109544	6,1	58,7	3613742 (3511344)	3504198	5050	99,8	96,9
Data Structure: **Red Black Tree**													
6	165	8535	29,31	327 (162)	231	52	81,6	70,6	25611 (25290)	25559	96	99,6	99,7
7	464	26894	118,20	911 (447)	679	267	71,7	74,5	111101 (110352)	110834	232	99,7	99,7
8	1323	72099	235,94	2489 (1166)	1709	2123	44,5	68,6	493546 (492139)	491423	699	99,8	99,5
Data Structure: **Binomial Heap**													
6	3013	31141	242,43	7602 (4589)	6864	31	99,5	90,2	35213 (35093)	35182	738	97,9	99,9
7	7973	70053	401,46	107416 (99443)	100301	456	99,5	93,3	154372 (154235)	153916	7115	95,5	99,7
8	12889	111674	289,46	603744 (590855)	562354	34756	94,1	93,1	719450 (719261)	684694	41390	94,2	95,1

object classifier with our technique, with a relatively small scope (5 for all cases), and produced likely invariants with Daikon, processed as for RQ2. We then compared Randoop with invariant checking disabled, and Randoop with invariant checking enabled (`@checkRep`) using: *(i)* the learned classifier, and *(ii)* the Daikon "filtered" invariant (only valid properties are kept), to check in each case the bug finding ability. Every Randoop execution for instance generation was run as for RQ2, while for bug finding a timeout of 10 minutes was set. The results are summarized in Table IV. In the case of ANTLR, Randoop is not able to catch the bug under any configuration. The reason is that, due to the mechanism that Randoop uses for incrementally building test cases, it cannot produce the aliasing situation that is necessary to catch the bug (basically, adding a node to itself as a child, a situation that the class interface allows for, but Randoop cannot produce). However, when manually building this scenario, the learned classifier detects the anomaly, whereas no anomaly is detected without invariant checking (i.e., no exception or other obvious error is observed) nor with the filtered Daikon invariant. We marked this case as "manual" to reflect this singularity.

All the experiments presented in this section can be reproduced following the instructions found in the site of the replication package of our approach [3].

A. Discussion

Let us briefly discuss the results of our evaluation. Regarding RQ1, our technique for producing invalid objects by exploiting field extensions worked reasonably well. In general, less than 5% of the presumably invalid objects were actually valid, with an effectiveness that increased for larger scopes. A closer look at these cases shows that most spurious invalid cases have to do with producing changes in data fields, that the neural network identifies as anomalous but the known invariant allows for. That is, when a change to a structure field is produced, it mostly leads to an invalid object. In other words, field extensions seem to accurately summarize field value feasibility. An issue that may affect this effectiveness is the budget set for generating valid instances (and collecting field extensions). The multiple Randoop runs performed with different seeds produced sufficiently large samples of valid structures, in our experiments, but this budget may be extended to obtain more precise field extensions in other case studies.

Regarding RQ2, our experimental results show that the artificial neural network produces object classifiers that closely approximate class invariants. Indeed, the technique learns classifiers that achieve a very high precision and recall for negative cases, and significantly better precision/recall for positive ones, compared with related techniques. In other words, misclassified cases are significantly more likely to be invalid inputs classified as valid, rather than the opposite. This is a positive fact for bug detection, since it confirms that classifiers tend to *over-approximate* class invariants (they will produce fewer false negatives). The case studies where we had less precision for positive cases were Binary Search Tree and Red Black Tree. These cases classify various invalid objects as valid. We confirmed that the reason for this observed learning limitation in these case studies has to do with the complexity of the invariants of these data structures regarding data objects, more precisely sortedness; indeed, all invalid cases that were misclassified as valid were correct from a structural point of view, but violated sortedness. Further experimentation with more complex network topologies (larger number of hidden neurons) may show a better performance in these cases. Still,

accuracy of our fully automated approach is significantly better than Daikon's manually filtered invariants.

Regarding RQ3, let us remark that our comparison is between our technique, that is fully automated, and a manually filtered instance of Daikon. In particular for this research question, the effort required to produce the filtered version of Daikon-produced invariants is significant, since in most of these cases we did not have reference invariants to compare to, and thus each likely invariant had to be carefully examined to decide whether it was valid, invalid, or valid for some cases.

For instance, for the binomial heap case study, the resulting post-processed Daikon invariant has 31 lines, and involved going through 26 likely invariants, difficult to reason about (it is a quite sophisticated data structure). Our results show that we achieve significantly better bug finding compared to "no invariant" and "filtered" Daikon invariant analyses, since our object classifiers catch 13 out of 17 bugs, while with no invariant only 3 bugs are detected, and the filtered Daikon invariant finds 6 out of 17. This is a very important result, taking into account the effort required for the engineer to produce the processed Daikon invariants, and the fact that our technique is fully automated.

The 3 bugs that in the case of `schedule` can be found with invariant checking disabled, throw exceptions, and thus do not need a specification to be caught. The remaining 5 bugs do not produce state changes, and thus cannot be caught by invariant checking. The additional bug that we found is in fact a bug that is not explicitly indicated in the repository. This 9th bug was discovered in the supposedly correct version. It is a bug in the `upgrade_process_prio` routine, that moves a process to a queue of a higher priority, but it does not update the process priority correctly. Indeed, a line in this routine that performs the following assignment:

```
proc->priority = prio;
```

should instead be as follows:

```
proc->priority = prio+1;
```

The SIR repository includes another scheduler implementation (`scheduler2`). We did not include this case study in our evaluation because all seeded bugs correspond to routines that do not change object states, and thus cannot be caught by just checking invariant preservation. The bugs seeded in the red-black tree implementation from [42] all correspond to the `insert` method. We trained the neural network using a correct version of this method, and then used it to attempt to catch the seeded bugs.

B. Threats to Validity

Our experimental evaluation is limited to data structures. These are good representatives of data characterized by complex invariants, which are beyond known invariant inference techniques such as Daikon. From the wide domain of data structures, we have selected a large set for which invariants were provided elsewhere, for answering RQ1 and RQ2, that required provided invariants. This benchmark includes data structures of varying complexities, including cyclic and acyclic topologies, balance, sortedness, etc. One may argue that restricting the analysis to these case studies might favor our results. While an exhaustive evaluation of classes with complex constraints is infeasible, we consider that invariant complexity (especially for invariants whose expression goes beyond simple constraints such as linear comparisons) is a crucial aspect we want our approach to target, and designed the experiments taking this issue into account. The evaluated structures correspond to a broad range of complexity, going from those with simple linear structures to other with tree-like, balanced shape. For RQ3, we did not need classes with provided invariants. We chose to analyze buggy data structures taken from the literature, as opposed to evaluating on our own seeded faults, to avoid unintentional bias.

The evaluation is largely based on implementations taken from the literature. Korat case studies had to be extended, however, with builders. Our implementations were carefully examined, as an attempt to make these respect the corresponding invariant, and to remove possible defects that would affect our experiments. We did not formally verify our implementations, but errors in these would have implied invalid objects being generated as valid, thus affecting the outcome of our whole learning approach. That is, errors in our implementations would have derived in less precision, i.e., they would hinder our results rather than favour them.

VI. Related Work

Many tools for automated program analysis can profit from invariants. Some tools use invariants for run time checking, notably the Eiffel programming language, that incorporates contracts as part of the programming language [26], the runtime assertion checker and static verifiers that use JML [9], Code Contracts for .NET [5], among others. Some techniques for automated test case generation also exploit these invariants for run time checking, converting the corresponding techniques into bug finding approaches. Some examples are Randoop [29] and AutoTest [27]. Our approach learns object classifiers that can be used in place of class invariants, but which are are *black box*, i.e., are not composed of explicit constraints that can be inspected. But since many of the above mentioned tools simply use invariants for object checking, without any dependency on the internal structure of the invariant, they are useful in these contexts. However, tools like Korat and Symbolic PathFinder, that require class invariants in the form of `repOK` routines (or partially symbolic versions of these) to be provided, cannot be used with our learned classifiers, since they exploit the program structure of the invariant to drive the search for valid inputs [8], [30].

The oracle problem has been studied by many researchers, and techniques to tackling it in different ways, have been proposed [6]. Our approach is more closely related to techniques for *oracle derivation* [6], more precisely, for *specification inference*. Within this category, tools that perform specification inference from executions, like ours, include Daikon [13] and JWalk [38]. Both these tools attempt to infer invariants from

TABLE III
PRECISION AND RECALL OF MANUALLY FILTERED DAIKON INVARIANTS ON COMPLEX DATA STRUCTURES.

Scope	Instances		Testing									
			Positive Instances					Negative Instances				
	Positive	Negative	*total*	*tp*	*fp*	*precision(%)*	*recall(%)*	*total*	*tn*	*fn*	*precision(%)*	*recall(%)*
Data Structure: **Singly Linked List**												
6	2486	13245	3906	3906	42770	8,3	100,0	42778	8	0	100,0	0,0
7	7867	33456	55987	55987	725964	7,1	100,0	725973	9	0	100,0	0,0
8	16416	64783	960800	960800	9999997	8,7	100,0	10000000	3	0	100,0	0,0
Data Structure: **Singly Sorted List**												
6	240	10306	252	252	713	26,1	100,0	911	198	0	100,0	21,7
7	830	33949	924	924	3421	21,2	100,0	3978	557	0	100,0	14,0
8	2577	83131	3432	3432	15944	17,7	100,0	17781	1837	0	100,0	10,3
Data Structure: **Doubly Linked List**												
6	5155	43035	55987	55987	345246	13,9	100,0	465917	120671	0	100,0	25,8
7	9381	76574	960800	960800	6862849	12,2	100,0	8914750	2051901	0	100,0	23,0
8	14801	122092	1000000	1000000	7930490	11,1	100,0	10000000	1069510	0	100,0	10,6
Data Structure: **Binary Tree**												
6	193	7062	197	197	4634	4,0	100,0	4638	4	0	100,0	0,0
7	567	21080	626	626	17844	3,3	100,0	17848	4	0	100,0	0,0
8	1401	54552	2056	2056	68806	2,9	100,0	68810	4	0	100,0	0,0
Data Structure: **Binary Search Tree**												
6	692	18445	731	731	40157	1,7	100,0	61219	21062	0	100,0	34,4
7	2198	50808	2950	2950	330636	0,8	100,0	468758	138122	0	100,0	29,4
8	5368	106663	12235	12235	2644744	0,4	100,0	3613742	968998	0	100,0	26,8
Data Structure: **Red Black Tree**												
6	165	8535	327	327	4122	7,3	100,0	25611	21489	0	100,0	83,9
7	464	26894	911	911	20796	4,1	100,0	111101	90305	0	100,0	81,2
8	1323	72099	2489	2489	94296	2,5	100,0	493546	399250	0	100,0	80,8
Data Structure: **Binomial Heap**												
6	3013	31141	7602	7602	13699	35,6	100,0	35213	21514	0	100,0	61,0
7	7973	70053	107416	107416	58791	64,6	100,0	154372	95580	0	100,0	61,9
8	12889	111674	603744	603744	297057	67,0	100,0	719450	422393	0	100,0	58,7

TABLE IV
EFFECTIVENESS OF LEARNED CLASSIFIERS IN BUG FINDING.

Case Study	#Bugs	#Found No Inv.	#Found Obj. Class.	# Found Filt. Daikon
Antlr	1	0 (man.)	1 (man.)	0 (man.)
Scheduler	8	3	4	3
IntTreeSet	5	0	5	3
BinTree	1	0	1	0
BinHeap	1	0	1	0
FibHeap	1	0	1	0

positive executions, as opposed to our case that also includes a mechanism to produce (potentially) invalid objects. We have compared in this paper with Daikon, since JWalk tends to infer properties that are more scenario-specific. The use of artificial neural networks for inferring specifications has been proposed before [36], [35]; these works, however, attempt to learn postcondition relations (I/O relations) from "golden versions" of programs, i.e., assumed correct programs. While this approach is useful, e.g., in regression testing or differential testing scenarios, using it in our case would mean to learn the I/O relation for a `repOK`, having the `repOK` in the first place, a simpler problem compared to what we are tackling here.

The notion of field extension as a compact representation of (the join of) a collection of generated structures was put forward in [33], and originates in the relational semantics of signature fields in Alloy [17], and more importantly, in the notions of upper and lower bounds introduced with the KodKod engine [39]. Our use of field extensions in this work, as a basis for the mechanism for "breaking" valid objects, is different from the purpose of bounds and partial bounds in the above cited works.

VII. CONCLUSIONS

Software specification plays a central role in various stages of software development, such as requirements engineering, software verification and maintenance. In the context of program analysis, there is an increasing availability of powerful techniques, including test generation [29], [27], [4], bug finding [14], [24], fault localization [43], [41] and program fixing [40], [21], [32], for which the need for program specifications becomes crucial. While many of these tools resort to tests as specifications, they would in general greatly benefit from the availability of stronger, more general specifications, such as those that class invariants provide. Invariants are becoming more common in program development, with methodologies that incorporate these [25], [22], and tools that can significantly exploit them when available for useful analyses.

We developed a technique, based on artificial neural networks, for inferring object classifiers to be used in place of class invariants. The technique is related to other, similarly motivated, approaches [13], [38], in the sense that it explores dynamic software behaviours for the inference, but it also incorporates a novel technique for producing *invalid* objects, enabling the training of a neural network. We have analyzed the use of neural networks for learning object classifiers, and showed that the learning process achieves very high accuracy compared to related approaches, that our mechanism to build supposedly invalid objects is effective, and that the learned

object classifiers improve bug detection, as evidenced by experiments on a benchmark of data structures of varying complexities.

This work opens several lines for future work. On one hand, our artificial neural network is built with rather standard parameters; adjusting variables such as number of hidden layers, activation function, etc., may be necessary, especially when scaling to larger domains. The performance of artificial neural networks can also be improved by *feature engineering* [16], a mechanism we have not yet explored. Our experiments were based on the use of random generation for producing valid objects, the initial stage of the technique. Using alternative generation approaches such as model checking and symbolic execution, may lead to different, possibly more precise, results.

REFERENCES

[1] Fibonacci heap implementation from the graphmaker library. https://github.com/nlfiedler/graphmaker. Version control revision of the bug: https://github.com/nlfiedler/graphmaker/commit/13d53e3c314d58cb48a6186437a36241842c98d7#diff-1c644baf14f6ab27ffa2691c9ff02cbd. Accessed: 2018-09-02.

[2] Home page of the korat test generation tool. http://korat.sourceforge.net. Accessed: 2017-07-01.

[3] Replication package of the object (in)validity learning approach. https://sites.google.com/site/learninginvariants.

[4] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. Improving test generation under rich contracts by tight bounds and incremental SAT solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 21–30. IEEE Computer Society, 2013.

[5] 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.

[6] 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.

[7] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, February 2012.

[8] 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.

[9] 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.

[10] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 281–290. ACM, 2008.

[11] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120. ACM, 2006.

[12] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[13] 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.

[14] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, pages 25–36. ACM, 2010.

[15] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 225–234. ACM, 2010.

[16] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

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

[18] Daniel Jackson, Somesh Jha, and Craig Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.*, 20(2):302–343, 1998.

[19] 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.

[20] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[21] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[22] Barbara Liskov and John V. Guttag. *Program Development in Java - Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.

[23] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[24] Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: bounded model checking of C and C++ programs using a compiler IR. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, volume 7152 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2012.

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

[26] 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.

[27] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In Jan van Leeuwen, Giuseppe F. Italiano, Wiebe van der Hoek, Christoph Meinel, Harald Sack, and Frantisek Plasil, editors, *SOFSEM 2007: Theory and Practice of Computer Science, 33rd Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 20-26, 2007, Proceedings*, volume 4362 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2007.

[28] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The yogiproject: Software property checking via static analysis and testing. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[29] 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.

[30] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[32] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Trans. Software Eng.*, 40(5):427–449, 2014.

[33] Pablo Ponzio, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Field-exhaustive testing. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 908–919. ACM, 2016.

[34] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[35] Seyed Reza Shahamiri, Wan M. N. Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. Artificial neural networks as multi-networks automated test oracle. *Autom. Softw. Eng.*, 19(3):303–334, 2012.

[36] 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.

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

[38] 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.

[39] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[40] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374. IEEE, 2009.

[41] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016.

[42] Razieh Nokhbeh Zaeem, Divya Gopinath, Sarfraz Khurshid, and Kathryn S. McKinley. History-aware data structure repair using SAT. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2012.

[43] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 272–281. ACM, 2006.