Learning to Prune Infeasible Paths in Generalized Symbolic Execution

Facundo Molina*, Pablo Ponzio*, Nazareno Aguirre*, Marcelo Frias†
*University of Río Cuarto and CONICET, Argentina
†Buenos Aires Institute of Technology and CONICET, Argentina
{fmolina, pponzio, naguirre}@dc.exa.unrc.edu.ar, mfrias@itba.edu.ar

***Abstract*—Symbolic execution allows one to systematically explore program paths by executing programs on symbolic inputs, and constructing path conditions that can be analyzed using constraint solving. When programs handle heap-allocated structures, and executions are assumed to begin in states satisfying a property like a precondition or invariant, symbolic execution not only needs to maintain path conditions, but also partially concrete heaps. Partially concrete heaps are increasingly concretized as symbolic execution progresses, and their feasibility (i.e., deciding whether they can be extended to fully concrete structures that satisfy the precondition) needs to be determined, to deem a path realizable and continue execution. This latter task generally requires the manual provision of routines to check the feasibility of partially concrete structures, which are often imprecise (e.g., do not detect all infeasible structures), and increase the cost of symbolic execution.**

In this paper, we improve the above situation by proposing an automated machine learning technique for determining whether a partially symbolic structure can be extended into a concrete structure satisfying a given invariant. Our approach does not require additional, manually provided routines for checking structure feasibility. It is based on recognizing feasible/infeasible partially symbolic structures by using a neural network, which is trained with automatically generated partially symbolic structures. These structures can be obtained by either symbolically executing the assumed invariant, or by generating and mutating structures using assumed-correct building routines. Our experiments, based on a benchmark of heap-allocated data structures of varying complexities, show that by incorporating our learned symbolic invariants as a pruning mechanism within Symbolic PathFinder, path infeasibility detection is greatly improved, as well as symbolic execution running times.

***Index Terms*—Symbolic execution, lazy initialization, neural networks**

I. Introduction

Symbolic execution (SE) [16] is a well-established program analysis technique, that allows one to systematically explore paths of a program under analysis, and exploit these explorations for a number of tasks, such as test generation [15], [13], [7], program verification [15], [22], and estimating worst case running times [6], [18], among others. As opposed to concrete program executions, which run on concrete program inputs, SE operates by executing programs on *symbolic* inputs, and constructing *path conditions* that correspond to the conditions that have to hold on program inputs, so that the currently explored path is executed [16]. Analysis via SE constitutes a significant advantage over concrete executions due to, essentially, two main characteristics: many concrete program executions are collapsed into single symbolic executions, thus leading to a significantly smaller explosion of executions, and collected path conditions can be checked for satisfiability using constraint solvers, so that unsatisfiable paths are pruned early in the symbolic analysis. Constraint solving is actually an important part of SE, since it is the driving technology used, e.g., to solve paths and generate program inputs for testing, among other applications.

SE was originally devised as a technique for programs handling basic datatypes, or simple structured types. As a result, path conditions typically involve arithmetic and logical constraints, that in many cases can be solved resorting to constraint solvers, such as SAT and SMT solvers [20]. When programs handle complex heap-allocated data structures, and their SE is assumed to begin in a state satisfying a certain property, e.g., a precondition or invariant, SE starts on a symbolic representation of the heap [15], [5]. As SE progresses, path conditions are collected and heaps become partially "concretized", according to the behavior of the program under analysis, and its manipulation of the heap. In order to decide whether a given (partially) symbolic program state is feasible or not, so that the path can be continued or pruned, one then needs to determine whether a partially symbolic heap can lead to a fully concrete structure that satisfies the precondition or not. In general, this cannot be straightforwardly decided by a constraint solver, since structure feasibility depends on some *ad hoc* precondition, that in many cases is captured as code in the programming language, e.g., as an operational representation invariant or repOK [17]. Thus, it is in principle beyond theories that SMT solvers support. Existing approaches to determining whether a partially symbolic heap satisfies a given precondition or not are based on manually or semi-automatically producing "hybrid" repOK procedures, that are typically weak approximations of repOK routines for partially symbolic structures (i.e., they tend to accept many infeasible partially symbolic structures) [15], [23], or require further assistance from users, e.g., by requiring an additional manually provided "hybrid" repOK, or a declarative version of this constraint, expressed in a logical formalism [25]. Besides the manual effort required in providing these "hybrid" repOKs, there is also a cost in calling these routines whenever the partially symbolic heap is further concretized, diminishing the overall efficiency of the SE process.

In this paper, we deal with the described situation via an

automated machine learning technique that obtains a partially symbolic structure classifier, i.e., a model that can detect whether a partially symbolic structure can be extended into a fully concrete structure satisfying a given invariant. Our approach is based on training a neural network with automatically produced valid/invalid partially symbolic structures. We employ two alternative mechanisms to generate the training set. The first consists of using SE on the corresponding invariant, and from each valid (resp. invalid) structure obtained, automatically generating a family of valid (resp. invalid) structures through an abstraction (resp. concretization) process. This approach is worthwhile because symbolically executing repOK generally scales better than symbolically executing other more sophisticated methods, especially "destructive" methods, i.e., those that modify the heap (repOK is intrinsically side-effect free, making it simpler to symbolically execute than other methods [25]). The second approach consists of using run-time structure generation: a set of assumed-correct building routines is used to produce valid structures, which are in turn mutated to build invalid ones; then, the same abstraction/concretization mechanism described above is employed, to generate families of partially symbolic structures from the collected concrete ones.

Notice that our technique, whatever the procedure used to generate the training set, is unsound, in the sense that using a neural network may wrongly classify some valid partially symbolic structures, as invalid. While this issue invalidates the use of our technique in SE contexts that need to guarantee exhaustive path exploration, the technique is still worthwhile for other SE execution contexts, e.g., for automated test generation, or worst-case efficiency estimation. Also, its applicability is better suited for programs dealing with heap-allocated structures involving complex representation constraints: in SE on subjects with weak constraints, almost every path leads to feasible structures, thus leaving almost no room for pruning, and making our approach to constitute a time overhead. Our experiments, based on a benchmark of heap-allocated data structures of varying complexities, show that by incorporating our trained neural networks as a pruning mechanism into Symbolic PathFinder [23], we preserve high levels of test suite quality metrics (for test suites generated with Symbolic PathFinder), while greatly improving path infeasibility detection, as well as SE running times.

II. Background

A. Symbolic Execution

Symbolic execution [16] collapses families of executions by substituting concrete values for program variables with *symbolic* ones. The use of symbolic values instead of concrete ones implies that whenever branching conditions involving symbolic values are visited while symbolically executing a program, the satisfaction of these conditions cannot be directly decided and instead constraints are collected to reflect the decisions that must be taken to visit specific paths. To systematically explore (bounded) program paths, this process is exhaustively followed through backtracking. The constraints gathered in the traversal of a specific path, that typically include some *precondition* (an initial assumed condition on program inputs), are called its *path condition*, since it represents the conditions that variables must meet to traverse the path. Such conditions can be checked for feasibility using constraint solvers (typically SMT solvers [9], [24]); for final paths (paths that cannot be further extended), solving the conditions produces inputs that exercise the corresponding path, and can be used for automated test generation and verification (in this last case, conjoining the path condition with the negation of a postcondition, for instance) [23], [28]; for non-final paths, path conditions deemed infeasible can prune the systematic path traversal of a symbolic executor, forcing it to backtrack and explore other paths. Thus, path condition checking is a very important part of symbolic execution. In general, for programs involving iteration or recursion, the number of program paths is either very large or infinite. Thus, symbolic execution typically requires some sort of *bound*, e.g., in the maximum length of paths to be considered.

B. Lazy Initialization

When programs manipulate variables of basic datatypes or simple structured types, solving path conditions can be resolved resorting to standard constraint solving technology such as SMT solving, thanks to the availability of theories that support these datatypes. But when programs involve user-defined heap-allocated datatypes, SMT-solvers cannot straightforwardly handle constraints on these datatypes, and thus some complementary approach is in order. A successful mechanism to deal with this situation is the concept of *lazy initialization* (LI) [15]. LI proposes to initially set every reference-based variable to a symbolic "object", that will be partially concretized when first accessed. The partial concretization will have a number of alternative paths, that will need to be exhaustively explored. A symbolic object can be concretized: *(i)* as `null`; *(ii)* as a reference to a previously observed object of the corresponding type, i.e., a compatible object already in the partially symbolic heap (notice that this alternative actually corresponds to as many cases as compatible objects are already in the heap); or *(iii)* as a reference to a *new* object (one not previously observed), whose fields are all symbolic. Notice that in this generalized context, symbolic execution will maintain a path condition as well as a partially symbolic heap, and one still needs to establish feasibility, now of the path condition in conjunction with the partially symbolic heap, over which one will also typically have some assumed precondition (e.g., an assumed representation invariant on a heap-allocated data structure).

As an example, let us consider a simple implementation of heap-allocated binary trees of integers, and a search method, that given a node of a tree and a value to search for, returns true iff the value belongs to the subtree having the node as root. The method is shown in Figure 1(a), and features a precondition, that assumes the node is the root of a binary tree (acyclic structure, with each reachable node except the root having exactly one parent). Structure (1) in Figure 1(b)

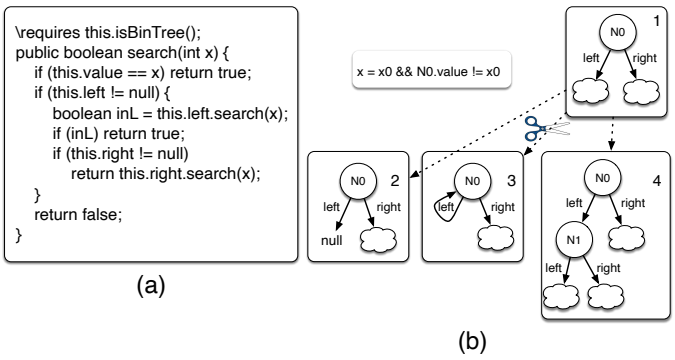

Fig. 1. A binary tree method, and its symbolic execution using LI.

shows a sample partially symbolic heap, along with its path condition, that would correspond to evaluating the if statement in the first line of method `search`, on its "else" branch. Now, when the condition in the second if statement is symbolically evaluated, three branches are generated, according to LI: one for the case in which `this.left` is null (structure (2)); one in which `this.left` points to the only previously seen node (structure (3)); and one in which `this.left` points to a new node, with all its fields symbolic (field `value` is not shown for simplicity). Notice how the second branch should be pruned, since it violates the precondition, stating that `this` points to a valid tree (more details on how this is implemented are presented later on).

In the same way as paths have to be bounded for an exhaustive exploration, heap size must be limited too. A maximum number of objects, for instance, is a typical approach, that when reached will limit the alternatives for LI to the first two cases in the above-described alternatives (i.e., instantiating a symbolic object to null or a previously observed object). We will sometimes refer to this bound as the *scope* of the analysis. For further details, we refer the reader to [15].

C. Feed-forward Neural Networks

Feed-forward neural networks are one of the most traditional and successful machine learning models. These learning mechanisms offer some key advantages, including their remarkable ability to learn and model non-linear and complex relationships in data, that are otherwise very complex to be captured. Moreover, unlike many other prediction techniques, neural networks do not impose any restriction on the input data, such as how it should be distributed.

The main components of neural networks are its computational units called *neurons*. Each neuron receives inputs, multiplies these inputs by the corresponding weights in its incoming links, and then applies a mathematical function g, called the *activation function*, that calculates the output o of the neuron. A neural network is then simply a group of neurons, connected by directed weighted links and arranged according to a certain topology. In particular, in a *feed-forward neural network* neurons are organized as a directed acyclic graph, and thus information only travels forward in the network, first through the input nodes (*input* layer), then through the so-called hidden nodes (*hidden* layers), and finally through the output nodes in the *output* layer. Often, neural networks will have one hidden layer, since one layer is enough to approximate many continuous functions [26], but there may be many of them.

Feed-forward neural networks are primarily used for *supervised* learning problems [14], i.e., problems for which one has a set of input-output pairs, and wants to approximate some function f such that for every input of fixed size x, $f(x) \approx y$, with y being the output of the network. To approximate such a function f, one first defines the architecture of the network. Then, the network can be fed with the available inputs, and the obtained outputs are compared with the expected ones, slightly adjusting the weights of the neurons according to the difference obtained between the expected and actual results. Performing this task over and over for a sufficiently large number of available inputs will result in the network approximating function f. A mechanism that is often employed to alter the weights of all neurons in a network, including those in hidden layers, is *backpropagation* [14].

Supervised learning problems where the function to be approximated has a boolean output (only two possible outcomes) are called *binary classification problems*. The problem we deal with in this paper is the classification of partially symbolic structures as feasible (i.e., as a structure that can be concretized to one satisfying a given precondition) or infeasible (i.e., a structure for which any possible concretization would violate a given precondition), and clearly falls in the category of binary classification problems.

III. A Motivating Example

Consider again the binary tree example, depicted in Figure 1, in the previous section. Let us continue with this example, and see in more detail the precondition of method `search`. A typical implementation of such a precondition is in an *operational* manner, with source code that checks for acyclicity, as in `isBinTree()` in Figure 2. Clearly this code is designed to perform on fully concrete structures, and thus cannot be directly run on structures such as those of Figure 1(b), since these have symbolic references as well as symbolic values for basic datatype fields (recall that we did not include in the figure the `value` field). One may, however, use this operational specification on partially symbolic structures, by attempting to run the code on such a structure, and if it is sufficiently concrete so that the code executes without running into a symbolic value/reference, return the corresponding value; if not, simply return true.

Using this simple approach, the partially symbolic structure (3) in Fig. 1(b) can be deemed infeasible; but its mirror (left subtree symbolic, right subtree with a loop) would be considered valid. The following, more general mechanism, resolves this issue: start by attempting to run the code on a partially symbolic structure; if it is sufficiently concrete so that the code reaches a final state without running into a symbolic value/reference, return true or false as appropriate;

```
public boolean isBinTree() {
   Set visited = new HashSet();
   visited.add(this);
   LinkedList workList = new LinkedList();
   workList.add(this);
   while (!workList.isEmpty()) {
      Node curr = workList.removeFirst();
      if (curr.left != null) {
         if (!visited.add(curr.left)) return false;
         workList.add(curr.left);
      }
      if (curr.right != null) {
         if (!visited.add(curr.right)) return false;
         workList.add(curr.right);
      }
   }
   return true;
}


public boolean hybridIsBinTree() {
   Set visited = new HashSet();
   visited.add(this);
   LinkedList workList = new LinkedList();
   workList.add(this);
   while (!workList.isEmpty()) {
      Node curr = workList.removeFirst();
      if (curr.left != null && curr.left != symb) {
         if (!visited.add(curr.left)) return false;
         workList.add(curr.left);
      }
      if (curr.right != null && curr.right != symb) {
         if (!visited.add(curr.right)) return false;
         workList.add(curr.right);
      }
   }
   return true;
}
```

Fig. 2. An implementation of `isBinTree` and its hybridized version.

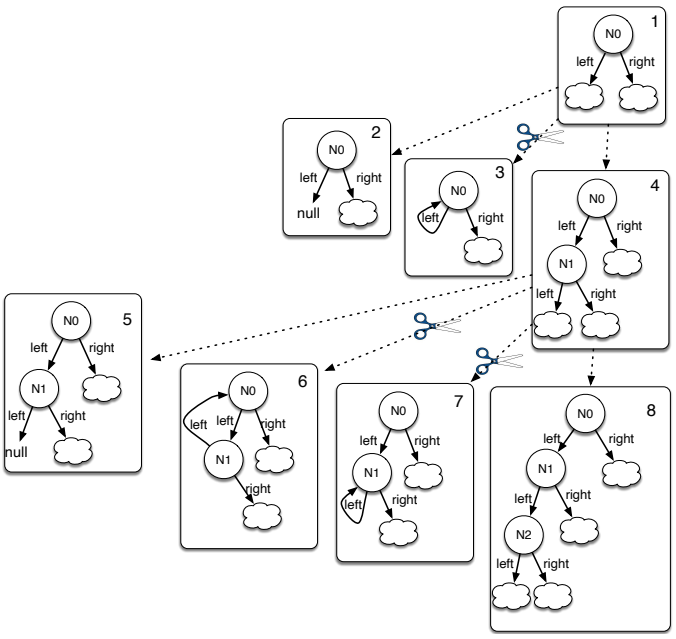

Fig. 3. Symbolic execution using LI from a balanced binary tree.

if a symbolic value is found instead, backtrack and continue execution. As opposed to the previous case, this case does not necessarily overapproximate the precondition, since it will depend on how the precondition code is structured, and what its default return value is. But it has as an advantage that it can be systematically produced from the code of the precondition, i.e., we do not need to ask the developer to provide a "hybrid" precondition (able to operate both on fully concrete structures and partially symbolic ones), we obtain it from the provided one. The "hybridized" version `hybridIsBinTree()` of `isBinTree()` is also shown in Fig. 2.

The above described situation is the best case scenario for the hybridization approach for preconditions, since the obtained function perfectly characterizes partially symbolic structures that can be extended into fully concrete structures satisfying the original precondition; that is, the hybrid precondition returns true if and only if the structure is fully concrete and satisfies the precondition, or is partially symbolic and can be extended into a fully concrete one satisfying the precondition. But this best-case scenario is rarely the case. Consider, continuing with our example, the slight modification in which the binary tree is also assumed to be *balanced*. Assume further that the method that implements the precondition checks first that the structure is indeed a binary tree, and then that it is balanced, using the usual recursive procedure of checking balancedness of the two subtrees, and then comparing their heights. Moreover, consider the structures in Figure 3, that extend the symbolic execution in Figure 1. Structures 6 and 7 can be pruned using the hybrid precondition. But the feasibility of structure 8, on the other hand, depends on various factors, the bound on the size of the heap, or equivalently the bound on the maximum length to consider on bounded paths, in particular. In fact, notice that if the maximum number of nodes is 3, then the structure should be considered infeasible, since there is no way of concretizing the symbolic nodes such that the tree becomes balanced, with the available resources.

Being able to prune as many as possible, and as early as possible, of the infeasible paths, is crucial for the efficiency and effectiveness of symbolic execution. This is due to the fact that continuing infeasible paths will not only slow down the symbolic execution process, but it will also lead to producing spurious results in analysis, such as spurious violations to post-conditions or invalid inputs for testing (inputs not satisfying the corresponding precondition).

The aim of this paper is to contribute in exactly those cases in which the hybrid precondition is not good enough for infeasibility detection. Basically, we would like to build some kind of "oracle", able to determine whether a partially symbolic structure can be concretized in a way that satisfies a given precondition. The approach will use *invariants* of the structure, rather than preconditions, since these are a common core of all preconditions of methods of a given structure, and thus the cost of generating these oracles can be amortized across the analysis of all these methods. Our "oracles" for discerning valid from invalid partially symbolic structures will be generated using *neural networks*. This implies that we will need to tolerate some imprecision in these oracles. The

imprecision that corresponds to accepting invalid structures (i.e., false positives, considering infeasible program paths as feasible) is already an issue in symbolic execution, e.g., due to limitations in constraint solving technology (usually, when solving a constraint is beyond the capabilities of the employed constraint solver or times out, one may go on considering the path as feasible). The imprecision that corresponds to wrongly pruning feasible paths (false negatives), on the other hand, can be considered more serious, since it will prevent symbolic execution from being a *conservative* (bounded) analysis: an analysis technique that may report infeasible violations, but that, when no violations are reported, guarantees the absence of violations (at least within the bounds for analysis). Notice that this is, however, only important for symbolic execution as a means for *verification*, but other applications of symbolic execution, including test input generation and the estimation of worst-case running times, do not require conservativeness.

The key is then on how (im)precise the resulting pruning mechanism is: if it is too imprecise (letting many invalid cases pass and rejecting many valid cases), then the approach is worthless; but if the neural network achieves good precision, then it has the potential of increasing the scalability of symbolic execution (making it more efficient, allowing it to scale to larger scopes, etc.), while losing a relatively small number of valid executions and thus mildly affecting analyses such as test generation and running time estimation. As we will show later on in this paper, our neural networks achieve good precision for a number of sophisticated heap-allocated data structure implementations. Scaling to larger scopes is highly relevant, since certain software defects only arise with sufficiently large scopes. Estimations of software behaviors are in some cases precise enough if run for sufficiently large scopes, too. For instance, in self-balancing structures, rebalancing mechanisms are only triggered with sufficiently large number of elements. In a well studied implementation of binomial heaps, that was thought to be correct, a bug arose when analysis was able to reach scope 13 (binomial heaps with 13 nodes) [10]. In the context of worst-case running time estimation as proposed in [18], how far the required "exhaustive" symbolic execution up to a given scope can go affects the computed policy used for larger scopes, and the overall precision in the estimation; for instance, scope 8 leads to wrongly estimating a quadratic running time for `TreeSet.add`, and scope 9 results in out-of-memory errors (see our replication package).

To train a neural network we need a mechanism to produce valid and invalid partially symbolic structures. We will provide two alternatives for the generation of the training set. The first consists of producing valid/invalid partially symbolic structures by symbolically executing, using LI, the invariant of the corresponding datatype. The second uses run-time generation: it executes assumed correct building routines to build valid structures, and mutates valid structures to build invalid ones. For both alternatives, an abstraction/concretization mechanism is applied: it builds additional valid partially symbolic structures from generated valid ones, making symbolic some concrete parts in the structures; and builds additional invalid

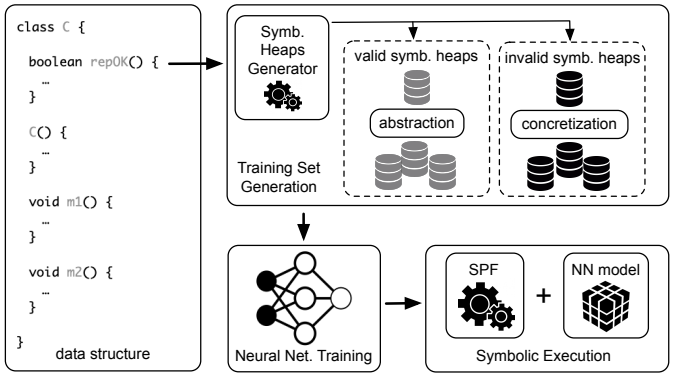

Fig. 4. An overview of the technique.

structures by concretizing parts of generated partially symbolic structures. Using these techniques, we generate sufficiently large training sets for our approach. The details are described in the next section.

IV. Learning a Classifier for Partially Symbolic Structures

In this section we present the details of our approach based on defining and training a feed-forward neural network, to learn to distinguish *feasible* partially symbolic structures (i.e., those that can be extended to fully concrete structures satisfying the given invariant) from *infeasible*, and then using the neural network as a pruning mechanism during symbolic execution. The workflow of the technique is shown in Figure 4. As described earlier in this paper, in order to use a neural network we will need to define a mechanism to automatically generate the partially symbolic structures that will then be used as the *training set*, propose an approach to represent these structures as numerical vectors (so that these can be fed to the network), and determine the architecture of the network as well as the values for *hyperparameters*.

A. Generating the Training Set

Let us assume that we want to perform symbolic execution on methods of a class *C*, for which a repOK routine, i.e, a representation invariant [17], is already provided. In order to train the neural network, sets of *feasible* and *infeasible* partially symbolic structures of class *C* are generated, using two alternative procedures. Intuitively, in both cases, *feasible* structures are those for which there exists an assignment of concrete values to each field holding a symbolic value, such that the final structure satisfies the given repOK; *infeasible* structures, on the other hand, are those for which there is no such assignment. To do so, we implemented a symbolic heaps generator with two different mechanisms. As we explain below, the first mechanism uses the provided repOK to generate the structures, while the second one is based on the use of assumed-correct building routines.

RepOK-based mechanism. Our repOK-based mechanism to generate partially symbolic structures for training is shown in

Algorithm 1. It starts by performing symbolic execution over a given repOK. Every time a final state is reached, it is either because the current structure s satisfies the repOK or not. In the case that s satisfies the repOK, then it is added to the set S_f of feasible structures, since it is clearly feasible. At this point, s probably has most of its fields with concrete values (at least the ones that are required to determine the satisfaction of the repOK), so additional *feasible* partially symbolic structures are automatically created from s, by randomly selecting concrete values v in s, and replacing them by symbolic values. Notice that this abstraction process is guaranteed to produce feasible structures: s itself is the witness of feasibility of each of the "less concrete" structures produced from s. Similarly, when the final state of the symbolic execution finishes with a structure s that does not satisfy the repOK, the structure is added to the set S_i of *infeasible* structures. In this situation, the structure s will in general have some of its fields with symbolic values, since often the repOK routine does not need a fully concrete structure to determine that it is invalid. When symbolic fields are present in s, additional infeasible partially symbolic structures are created, by replacing some symbolic fields in s by concrete values of the corresponding type. Basically, this approach is sound because we maintain fixed the part of s that led repOK to determine infeasibility; no matter how we concretize the remaining symbolic values in s, the structure will continue to be infeasible.

Algorithm 1: RepOK-based training set generation

```
Input: repOK of class C.
Output: the sets S_f and S_i.
Function GEN-TRAINING-SET(repOK):
    S_f ← ∅;
    S_i ← ∅;
    Paths ← SYMB-EXEC(repOK);
    for p in Paths do
        s ← GET-SYMBOLIC-HEAP(p);
        b ← GET-OUTPUT(p);
        if b then
            S_f ← S_f ∪ {s} ∪ ABSTRACT(s);
        else
            S_i ← S_i ∪ {s} ∪ CONCRETIZE(s);
    return ⟨S_f, S_i⟩
```

API-based mechanism. Algorithm 2 implements our second alternative for generating partially symbolic structures. This API-based mechanism for structure generation works under the assumption that a set of assumed-correct building routines, e.g., constructors, insertion and deletion routines, is provided. Since these methods are assumed to have correct implementations, they can be used to build valid, fully concrete, structures, employing any run-time test input generation mechanism, e.g., random generation as implemented in tools like Randoop [21]; this is in fact the mechanism that we use to automatically generate valid fully concrete structures in our evaluation. Once a set of valid concrete structures is built, a set of *valid* partially symbolic structures is created using the same abstraction process as in the previous mechanism: replacing fields with concrete values in valid structures, by symbolic ones. In order to generate *invalid* partially symbolic structures, we first need to create a set of invalid concrete structures. We do so from the valid structures, as follows: given a valid structure s, an object o reachable from s and a field f in o, the value of $o.f$ is mutated by randomly changing it to null or to a previously seen value of the same type, and then verifying if the resulting structure s' is actually invalid; we do so by using the provided repOK. Finally, from each invalid structure, a set of *infeasible* partially symbolic structures is generated, by maintaining the concrete part that caused the original structure to be invalid (the "accessed fields", in the terminology of [3], traversed when running the failing repOK), but with some of the remaining fields pointing to a symbolic value.

Algorithm 2: API-based training set generation

```
Input: a set B of assumed-correct building routines of class
       C
Output: the sets S_f and S_i.
Function GEN-TRAINING-SET(B):
    T_s ← GEN-TEST-SUITE(B);
    C_f ← EXECUTE-TESTS(T_s);
    C_i ← ∅;
    for c in C_f do
        m ← MUTATE(c);
        if ¬ SATISFIES(repOK, m) then
            C_i ← C_i ∪ {m};
    S_f ← ∅;
    for s in C_f do
        S_f ← S_f ∪ {s} ∪ ABSTRACT(s);
    S_i ← ∅;
    for s in C_i do
        S_i ← S_i ∪ {s} ∪ ABSTRACT-INF(s);
    return ⟨S_f, S_i⟩
```

B. Partially Symbolic Structures as Vectors

Most implementations of neural networks require as inputs vectors of values, which are in general restricted to *numeric* types. In order to encode symbolic structures as vectors, we adopt the candidate vector format of Korat [3]. We extend the original representation, that only considers concrete values, to support symbolic values too. Given a scope k, that defines ranges for numeric types and maximum number of instances for reference types, any instance of a class C within scope k is represented in Korat by a vector containing a cell for each object $o' : C'$ within the scope and field `f` of C'. The domain of each cell is a range of natural numbers, where each value uniquely identifies an object/value within the scope. We add a special value -1 to represent a symbolic value of the corresponding type (determined by the index in the vector) and a special value -2 to represent unexplored fields, i.e., fields that were not traversed, either because of a small size of the structure or due to the presence of a symbolic value. Consider, as an example, our binary tree example, and assume that our analysis takes a scope of exactly one tree object, 3

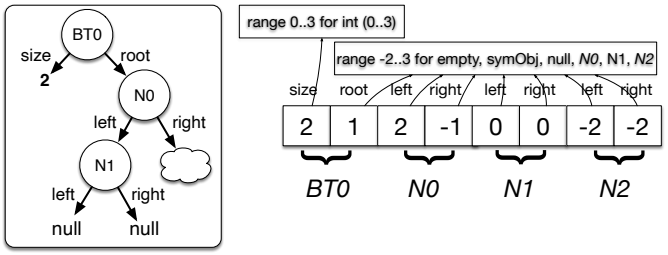

Fig. 5. A partially symbolic binary tree and its vector representation.

nodes, and size in the range 0..3. Input vectors will have, for this scope, 8 cells, as shown in Figure 5 (we ignore the cell for the receiver object, which will be in this case always initialized to 1, representing the sole concrete object of type `BinTree`). In this Figure, we also show a particular assignment for the vector, and the partially symbolic structure it represents.

C. Neural Network Architecture and Training

Our approach is based on the use of a *feed-forward artificial neural network*, whose input depends on the size of the vectors encoding the partially symbolic structures. Assuming that the size of the vectors is n, the input layer of our network contains n input neurons, each one receiving a position of the vector. The output layer will always have 1 neuron because our classification problem involves 2 classes, the class of *feasible* structures, and the class of *infeasible* ones. Only one hidden layer is used. The remaining parameters of the network such as the number of hidden units in the hidden layer and the activation function of the neurons, are determined using an optimization process known as *random search* [2]. More precisely, we used this approach by launching 10 random combinations of hyperparameter values, where, from all the partially symbolic structures generated as described in Subsection IV-A, 75% were used for training, and the remaining 25% were used to assess the performance of the neural network. Then, we selected the combination that exhibited the best performance. The level of precision that we achieved in our experiments did not demand further tuning of the neural network's parameters (see the evaluation section).

Implementation. Our approach is implemented as an extension of Symbolic PathFinder [23]. Given a target program with a (concrete) repOK and it's "hybridized" version for the corresponding data structure, standard Symbolic PathFinder uses LI with the hybridized repOK for pruning infeasible structures. Our neural network based approach (LI+NN) *complements* LI, as follows: after calling the hybridized repOK on a partially symbolic structure, if LI finds it feasible (i.e., LI lets the structure "pass"), we call the neural network to check for further pruning (we let the structure "pass" if the neural net accepts it). Our implementation uses the Python `Keras` library [8] to build and train the neural networks, and the `Deeplearning4j` open source Java library [12] to *wrap* the keras models, in order to make them available in Java.

V. Evaluation

The evaluation of our approach is organized around the following research questions:

RQ1 *How precise are the neural networks in detecting infeasible symbolic structures?*

RQ2 *Is generalized symbolic execution improved when incorporating the neural networks for further pruning?*

RQ3 *Does our approach improve symbolic execution based test generation?*

RQ1 analyzes the performance of the neural networks in detecting infeasible symbolic structures. RQ2 evaluates the impact of incorporating neural networks for further pruning during LI (in terms of efficiency/effectiveness for generating valid test data, and in comparison with standard LI). Finally, RQ3 analyzes our approach in an application scenario, namely automated test generation based on symbolic execution.

A. Evaluation Subjects

Our evaluation subjects are Java classes taken from the generalized symbolic execution literature, whose methods handle heap-allocated data that involve complex constraints. In order to appropriately address RQ1 and RQ2, we need subject methods for which *perfect* pruning information, i.e., a precise ground truth on which are the valid and invalid partially symbolic objects generated during symbolic execution, is available. Thus, we took the subjects used as case studies in [25], and considered as ground truth the pruning performed by the BLISS technique. We considered the following implementations featuring the fundamental methods of data structures of varying complexities: **BinTree**, the binary tree implementation discussed in our motivating example; **TreeSet**, a red-black tree implementation of the Set abstract datatype, from *java.util*; **AVLTree**, an AVL tree based implementation of the Map abstract datatype; and **BinomialHeap**, a Binomial Heap implementation of the Heap abstract datatype.

To address RQ3, we included case studies handling heap-allocated data with complex conditions, most of which contain either real or seeded bugs: **Caching**, a doubly linked list implementation that caches node objects to improve efficiency, from the Apache package *commons.collections* (used as case study in [10], [4]); **IntTreeSet**, an integer Set implementation over red black trees from KodKod [29], used for bug finding in [19]; **Schedule**, a scheduler implementation from the SIR repository, based on the use of doubly linked lists, also used for bug finding in [19]; and **Tsafe**, the class TrajectorySynthesizer of the TSAFE prototype, computing plane trajectories based on position and light plan, used in [4].

B. Evaluation Setup

All our experiments were run on an octa-core Intel Core i7 CPU, running GNU/Linux 4.4.0 at 3.4Ghz, with 6Mb of cache and 16Gb of RAM. 4 GB of heap memory were allocated for the Java virtual machine. Further details and case studies, as well as instructions to reproduce the experiments, can be found in the replication package site of our technique [1].

TABLE I
PERFORMANCE ON SYMBOLIC STRUCTURES CLASSIFICATION

Subject	S	Precision (%)			Recall (%)		
		LI	NN_{rb}	NN_{ab}	LI	NN_{rb}	NN_{ab}
BinTree	4	100	100	52.63	100	100	100
	5	100	100	61.66	100	100	100
	6	100	100	45.79	100	100	100
	7	100	100	80.42	100	100	100
	8	100	100	44.77	100	100	100
	9	100	100	29.6	100	100	100
TreeSet	4	100	69.23	64	77.77	100	88.88
	5	100	70.58	50	88.88	100	88.88
	6	100	82.31	69.46	76.85	100	95.86
	7	100	85.92	62.80	85.05	100	96.55
	8	100	89.27	56.25	82.21	100	100
	9	100	89.31	57.11	81.94	100	100
AvlTree	4	100	100	74.19	72.41	100	79.31
	5	100	100	96.36	86.88	100	86.88
	6	100	100	88.46	89.04	94.52	94.52
	7	100	98.75	70.08	68.20	99.58	82.84
	8	100	100	58.31	84.68	99.64	94.62
	9	100	100	80.94	86.50	99.78	96.20
BinomialHeap	4	100	95.23	60.52	86.95	86.95	100
	5	100	100	52.63	92.00	92.00	100
	6	100	100	65.07	96.32	96.32	100
	7	100	100	62.98	97.93	97.93	100
	8	100	100	69.75	98.63	98.63	100
	9	100	100	71.84	98.79	98.79	100
AVG		100	95.02	63.56	89.62	98.50	96.02

C. Performance of the Neural Networks (RQ1)

To evaluate the performance of the neural networks in detecting infeasible symbolic execution paths, we proceeded as follows. First, for each subject/scope, we created the training sets of valid/invalid symbolic instances using the procedures described in Subsection IV-A. Then, we determined the network architecture by using a random search process as described in Subsection IV-C. After this, we obtained a neural network NN_{rb}, trained with the dataset generated using the repOK-based mechanism, and a neural network NN_{ab}, trained with the dataset generated using the API-based mechanism. Finally, to assess the performance of these networks, we measured the standard precision and recall metrics [27] using as *validation set* the ground truth of valid/invalid symbolic instances, obtained from previous work [25]. Precision represents the proportion of invalid instances that were correctly detected by the network, with respect to the total number of instances classified as invalid. Recall represents the proportion of invalid, correctly classified, instances, with respect to the total number of invalid instances (this includes the valid symbolic structures incorrectly classified as invalid).

Table I summarizes the results of this first analysis. Notice how LI performs on these subjects: it achieves perfect precision, since it only prunes actually invalid structures (determined through the hybrid repOK); regarding recall, LI is incapable of detecting many invalid symbolic structures for the majority if the subjects (except for the BinTree subject, a case in which the hybridized repOK is able to precisely identify feasible/infeasible partially symbolic structures). Regarding the performance of the neural networks, on the one hand, both the NN_{rb} and the NN_{ab} outperform LI in terms of recall. That is, the neural networks are able to detect more invalid symbolic structures, an average of 98.50% for the NN_{rb} and 96.02% for the NN_{ab}, compared to the 89.62% of LI. On the other hand, the neural networks are less precise than LI, meaning that they disregard (i.e., prune) a number of valid inputs. However, the precision achieved by the NN_{rb} technique is significantly better than the precision of NN_{ab}.

Our experiment shows that the neural networks are very effective in correctly identifying feasible/infeasible partially symbolic structures. This is in keeping with the observations in [30], that learning models are in general very good at detecting structural properties. The technique can, however, disregard (i.e., prune) a number of valid inputs, due to false negatives. That is, the use of our approach might result in part of the state space not being explored. This is acceptable, as long as the wrongly pruned part is not significant and if symbolic execution is used for non-exhaustive analyses, e.g., testing, since such analyses are inherently incomplete; but may be unacceptable for verification. Our approach might also let invalid inputs pass in some cases (due to false positives). This is an issue that also affects LI, and that implies that these techniques may generate spurious symbolic paths, i.e., it may produce spurious test cases or report spurious bugs.

D. Impact on Generalized Symbolic Execution (RQ2)

To assess the impact on generalized symbolic execution, we took the fundamental methods of the analyzed data structures, and performed symbolic execution (for increasingly larger scopes) with LI and LI+NN, to explore *all* supposedly feasible bounded paths, and collect the corresponding structure instances. Table II summarizes the results of these experiments. For each method and scope (S), we report the exact number of feasible structures (#Feas.), as reported in [25] (the exact number of feasible structures is known as these are computed in [25] from a declarative invariant using SAT solving, preventing false positives or false negatives). Then, for each technique, scope and method, we report the running times (mm:ss) (using - to indicate that the timeout of 1 hour was exceeded) and the corresponding number of reported structures. Regarding running times, notice that in the case of binary trees, LI+NN is less efficient; this is reasonable: as we mentioned earlier in this paper, binary tree is a case where there is no room for improvement with respect to LI (thus, our technique constitutes an overhead for this case study). In the rest of the cases, and as the scope is increased, our technique outperforms LI, with a remarkable margin in some cases, AVL in particular. With respect to the reported structures, thanks to the further pruning, LI+NN is able to report far fewer structures compared to LI.

Evaluating the impact of LI+NN in terms of running times and reported structures can be misleading: our technique may

TABLE II
LI AND LI+NN ON GENERALIZED SYMBOLIC EXECUTION

Method	S	#Feas.	Time		Reported Structs		Precision of LI+NN	
			LI	LI+NN	LI	LI+NN	Feas.(%)	Spur.↓(%)
BinTree								
bfs	6	196	00:01	00:02	196	196	100	0
	7	625	00:02	00:04	625	625	100	0
	8	2055	00:04	00:07	2055	2055	100	0
	9	6917	00:13	00:19	6917	6917	100	0
	10	23713	00:48	00:50	23713	14463	60.99	0
	11	82499	03:13	02:11	82499	36259	43.95	0
dfs	6	196	00:01	00:02	196	196	100	0
	7	625	00:01	00:04	625	625	100	0
	8	2055	00:04	00:07	2055	2055	100	0
	9	6917	00:13	00:18	6917	6917	100	0
	10	23713	00:52	00:44	23713	14926	62.94	0
	11	82499	03:23	01:42	82499	29776	36.09	0
TreeSet								
bfs	6	26	00:01	00:03	196	185	100	6.47
	7	55	00:04	00:06	625	560	100	11.4
	8	95	00:13	00:06	2055	385	94.73	84.94
	9	141	00:45	00:07	6917	619	61.7	92.14
	10	217	02:40	02:47	23713	20875	98.61	12.06
	11	407	10:28	11:09	82499	75684	96.06	8.28
	12	863	38:50	35:18	290511	230512	84.47	20.66
	13	1767	-	43:13	-	254537	66.77	∞
add	6	26	00:44	00:14	26	24	92.3	0
	7	55	03:57	00:41	55	52	94.54	0
	8	95	23:47	01:30	95	78	82.1	0
remove	6	493	00:44	00:39	1542	1060	87.22	39.94
	7	2229	02:51	02:36	5367	3966	86.72	35.21
	8	8933	10:51	02:42	17957	3755	34.88	92.91
	9	20242	40:14	05:40	58542	7983	28.08	93.99
AvlTree								
contains	6	7	00:04	00:01	63	7	100	100
	7	15	00:10	00:07	127	78	100	43.75
	8	31	00:24	00:07	255	81	100	77.67
	9	31	00:58	00:10	511	107	100	84.16
	10	31	02:16	00:13	1023	139	100	89.11
	11	31	05:21	00:12	2047	132	100	94.99
	12	63	12:11	01:20	4095	634	100	85.83
	13	127	29:44	11:16	8191	3287	95.27	60.73
	14	255	-	15:04	-	4330	99.6	∞
add	6	29	02:54	00:06	485	40	100	97.58
	7	121	15:21	01:18	1383	204	72.72	90.8
	8	441	14:52	08:37	3883	697	52.38	86.46
	9	477	-	48:01	-	1500	80.29	∞
remove	6	51	11:00	00:35	427	63	100	96,8
	7	294	-	08:15	-	333	71.76	∞
BinomialHeap								
bfs	6	7	00:01	00:02	41	14	100	79.41
	7	8	00:02	00:03	72	9	87.5	96.87
	8	9	00:05	00:04	130	10	66.66	96.69
	9	10	00:16	00:07	232	12	80	98.19
	10	11	00:55	00:13	416	8	45.45	99.25
	11	12	03:47	01:18	742	36	91.66	96.57
	12	13	12:02	07:10	1328	241	100	82.66
	13	14	52:09	04:50	2372	12	42.85	99.74
insert	6	49	05:57	06:11	51	51	100	0
	7	105	34:24	32:13	107	82	76.19	0
AVG							84.12	46.98

be wrongly pruning a significant part of the search space, thus exhibiting more efficiency but at the cost of less thoroughness in the analysis. Although Table I suggests this should not be the case, we also report the precision of LI+NN showing the percentage of feasible structures (Feas.(%)) produced and the reduction of spurious structures achieved (Spur.↓(%)) compared with LI. Notice that LI can only prune infeasible structures. Thus, it always produces 100% of the actual feasible structures, but the number of spurious structures can be high for the most complex cases. LI+NN, on the other hand, although it incorrectly prune some cases, as this table confirms, it still produces a high percentage of the feasible structures (84.12% on average) while achieving a significant reduction of the spurious structures (46.98% on average).

TABLE III
SYMBOLIC EXECUTION BASED TEST GENERATION

Method	S	Test cases		Time	
		RepOK	LI+NN	RepOK	LI+NN
Caching					
removeIndex	4	6	54	00:00	00:01
	5	12	64	00:00	00:01
	6	20	66	00:00	00:01
IntTreeSet					
add(err1)	4	456	377	00:16	00:11
	5	916	1032	01:07	00:32
	6	2344	3376	04:58	02:08
remove(err1)	4	352	257	00:14	00:07
	5	876	695	00:57	00:19
	6	2147	2148	04:19	01:12
add(err7)	4	356	387	00:16	00:10
	5	888	992	01:03	00:27
	6	2176	3060	04:42	01:43
remove(err7)	4	241	259	00:11	00:06
	5	589	673	00:46	00:17
	6	1414	2056	03:43	01:04
Schedule					
upgrade	4	1202	145	02:45	00:06
	5	2244	191	05:15	00:06
	6	3876	257	09:17	00:06
TSafe					
getRouteTrajectory*	3	1	1	20:52	21:07
TOTAL		20116	16090	1:00:41	29:47

∗ The execution was performed until the bug was revealed.

E. Symbolic Execution based Test Generation (RQ3)

RQ3 evaluates our technique in a particular application scenario, symbolic execution based test generation. For the evaluation, we took a different set of case studies. These subjects only have a repOK without a corresponding hybridized version. Thus, we compare two alternative ways of generating test inputs. Firstly, for each target method m and scope (S), we first perform symbolic execution using as driver the following code snippet: `if (repOK()) { m(); }`, and then we count all the generated inputs reaching the end of the method. Secondly, we perform test generation by symbolically executing only the target method (without considering the repOK) using LI+NN. In this second approach, we use the neural network to complement the LI pruning.

Table III shows the results of this experiment. For each method and scope, we report the obtained test cases and the symbolic execution running times, using the above code-snippet (if-repOK) and our approach (LI+NN). Our technique

generated tests more efficiently, taking 29:37 (min:sec) compared to the 1:00:41 that the if-repOK approach required. Both approaches report similar numbers of tests cases (16090 of LI+NN, compared to the 20116 produced by the if-repOK approach). In summary, our technique enables an alternative way of performing symbolic execution based generation, that is more efficient in a number of subjects (notably IntTreeSet and Schedule). The main reason for the difference in performance has to do with if-repOK needing to eagerly concretize the heap objects, while LI+NN lazily concretizes heap object are these are accessed by the target method.

F. Limitations

Our LI+NN technique has a limitation: even when the neural network based pruning would allow us to scale symbolic execution to larger scopes, we are constrained by the largest scope we are able to symbolically execute repOK on (since we need to do so to obtain the training set). This is where alternative mechanisms to generate the training set become necessary. Our second mechanism to generate the training set is based on executing building routines and mutating valid structures, and can scale better than symbolic execution on repOK. This pays, however, a cost in precision, since the obtained training sets are less thorough. In Table II we have identified with * the cases for which the used network was trained with the building routines approach. For instance, in the case of method bfs of TreeSet, we trained with repOK-based sets up to scope 9, and from that point on, we used training sets obtained with building routines and structure mutation.

VI. Related Work

Symbolic execution, as a technique for automated analysis, is gaining increasing attention, and it is essential for the technique to effectively handle heap-allocated datatypes. The problem of symbolically executing code handling heap-allocated datatypes has been investigated by various researchers, and generalized symbolic execution via lazy initialization, as introduced in [15], is the main technique for doing so in the presence of operational invariants/preconditions. Improvements to this approach explore different directions: substituting operational specifications by specialized logical languages that can be subject to constraint solving [5]; introducing precomputed bounds to reduce the number of nondeterministic choices in lazy initialization [11], and complementing operational specifications with equivalent declarative specifications to exploit SAT Solving [25], are some known enhancements to generalized symbolic execution. These works differ from our current approach in the fact that they either require an additional specification, as well as the (costly) computation of infeasible cases from these descriptions [11], [25], or they replace the use of operational specifications altogether, introducing new specialized specification languages, that have to be mastered to exploit the associated techniques [5]. The main requirement for the application of our technique is the provision of an operational (repOK) specification, and/or the provision of assumed correct building routines, to generate the training sets. Our technique does pay a price in losing soundness, compared to the cited approaches.

Learning techniques have been applied to discovering specifications of complex data structures, in particular in relation to shape predicates [31], and in the binary classification of valid/invalid structures [19]. The learning problem is however different from ours, since in those works specifications are discovered from program behaviors, on fully concrete structures. In our case we do count with a specification for fully concrete structures, and need to learn a classifier for partially symbolic ones. Other applications of non-symbolic artificial intelligence techniques in combination with symbolic execution, include the use of evolutionary computation to build tests, composed of method sequences, guided by path conditions collected through symbolic execution [4].

VII. Conclusion

As symbolic execution gains relevance as a program analysis technique, and finds increasing adoption both in academic and industrial settings, the need for effective approaches that broaden the technique's applicability becomes apparent. In particular, the possibility of applying symbolic execution on custom heap allocated data abstractions is highly relevant, and at the same time technically very challenging. We have introduced an approach to improve lazy initialization, one of the main techniques for symbolically executing code handling heap-allocated data. The technique is based on learning a classifier for partially symbolic data, that can be used to identify infeasible (invalid) partially symbolic heaps and prune symbolic execution's search space. As opposed to related techniques, this approach does not demand further effort from the engineer: no extra specifications besides the usual operational specification of the data being handled, is required. This standard specification is exploited to produce training data, that is fed to a neural network, then used for identifying invalid partially symbolic heaps and opportunities for pruning symbolic paths. Alternatively, a set of assumed-correct building routines can be used to generate the training set. However, the introduced technique, whichever the training set approach, is *unsound*, in the sense that that it may lead to pruning feasible paths (this is in contrast to most known enhancements to symbolic execution, which only prune infeasible cases).

We have compared our technique with lazy initialization, on a number of data structures of varying complexities. Our results show that our learning approach can very precisely distinguish feasible from infeasible partially symbolic heaps, helps in improving the detection of infeasible symbolic paths, and reduces symbolic execution running times.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful feedback. This work was partially supported by ANPCyT PICT 2017-2622, 2019-2050, 2020-2896, and by an Amazon Research Award. Facundo Molina's work is also supported by Microsoft Research, through a Latin America PhD Award.

REFERENCES

[1] Replication package site for *Learning to Prune Infeasible Paths in Generalized Symbolic Execution*. https://sites.google.com/view/learning-symbolic-invariants.

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

[3] 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.

[4] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. SUSHI: a test generator for programs with complex structured inputs. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 21–24. ACM, 2018.

[5] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. JBSE: a symbolic executor for java programs with complex heap inputs. 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 1018–1022. ACM, 2016.

[6] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 463–473. IEEE, 2009.

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[8] François Chollet et al. Keras. https://keras.io, 2015.

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[10] 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.

[11] Jaco Geldenhuys, Nazareno Aguirre, Marcelo F. Frias, and Willem Visser. Bounded lazy initialization. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[12] Adam Gibson, Chris Nicholson, Josh Patterson, Melanie Warrick, Alex D. Black, Vyacheslav Kokorin, Samuel Audet, and Susan Eraly. Deeplearning4j: Distributed, open-source deep learning for java and scala on hadoop and spark, May 2016.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[15] 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.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

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

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

[18] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Symbolic complexity analysis using context-preserving histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 58–68. IEEE Computer Society, 2017.

[19] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. Training binary classifiers as data structure invariants. In *Proceedings of the 41th International Conference on Software Engineering: ICSE 2019, Montreal, Canada*, 2019.

[20] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[21] 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.

[22] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[23] 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.

[24] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[25] Nicolás Rosner, Jaco Geldenhuys, Nazareno Aguirre, Willem Visser, and Marcelo F. Frias. BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Software Eng.*, 41(7):639–660, 2015.

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

[27] Alaa Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 2018.

[28] 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.

[29] 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.

[30] Muhammad Usman, Wenxi Wang, Kaiyuan Wang, Cagdas Yelen, Nima Dini, and Sarfraz Khurshid. A study of learning data structure invariants using off-the-shelf tools. In Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay, editors, *Model Checking Software - 26th International Symposium, SPIN 2019, Beijing, China, July 15-16, 2019, Proceedings*, volume 11636 of *Lecture Notes in Computer Science*, pages 226–243. Springer, 2019.

[31] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 491–507. ACM, 2016.