Model Counting for Complex Data Structures

Antonio Filieri[1], Marcelo F. Frias[2], Corina S. Păsăreanu[3], and Willem Visser[4]

[1] University of Stuttgart, Stuttgart, Germany
[2] Instituto Tecnológico de Buenos Aires and CONICET, Buenos Aires, Argentina
[3] Carnegie Mellon Silicon Valley, NASA Ames, Moffet Field, CA, USA
[4] Stellenbosch University, Stellenbosch, South Africa

Abstract We extend recent approaches for calculating the probability of program behaviors, to allow model counting for complex data structures with numeric fields. We use symbolic execution with lazy initialization to compute the input structures leading to the occurrence of a target event, while keeping a symbolic representation of the constraints on the numeric data. Off-the-shelf model counting tools are used to count the solutions for numerical constraints and field bounds encoding data structure invariants are used to reduce the search space. The technique is implemented in the Symbolic PathFinder tool and evaluated on several complex data structures. Results show that the technique is much faster than an enumeration-based method that uses the Korat tool and also highlight the benefits of using the field bounds to speed up the analysis.

Keywords: Model counting, probabilistic software analysis, symbolic execution

1 Introduction

Model counting is the problem of computing the number of solutions (models) that satisfy a set of constraints. Model counting has found applications in worst case execution time estimation [27], increasing parallelism [38], quantitative information flow analysis [34], and many others.

We focus here on another important application, namely Probabilistic Software Analysis (PSA) [36,15,7,21]. PSA is an emerging technique to quantify the probability of reaching program events of interest assuming that program inputs follow given probabilistic distributions [15]. The input distributions allow data from real world observations to be incorporated in the analysis of programs that interact with their environment, as well as to encode uncertainty in design assumptions about the usage profile of a program, including the interactions with third-party components and systems. PSA is useful in many domains including debugging, cryptographic protocols, cyber-physical systems, biology, and reliability analysis [21].

Recent PSA techniques [18,15,16,28] use symbolic execution of the program to collect symbolic constraints on the inputs that lead to the occurrence of target program events. The number of satisfying assignments for these constraints are then calculated using *model counting procedures*. This gives a measure of how likely it is for an input distributed according to a given probabilistic distribution to satisfy the constraints.

Most work on probabilistic symbolic execution has focused on integers and used off-the-shelf model counting tools for computing the number of integer values within the volume of a convex polytope, e.g. LattE [2], to compute probabilities [18,15,16,28]. Recent techniques have been introduced to estimate the (approximate) number of solutions for floating point constraints [7]. However these techniques can not directly be applied to complex data structures, such as lists and trees. Analysis of programs that manipulate complex data is well studied with many approaches available, see e.g. shape analysis [40], specification-based testing [9] and constraint solving [24], among others. However model counting for data structures has not been addressed so far.

In this paper we propose a model counting procedure for a combination of heap and numeric constraints collected along a symbolic execution of a program. A simple approach is to enumerate all the possible data structures up to a given size and then to check their validity against the given constraints. However this becomes quickly intractable for large solution sets. We instead propose an approach based on symbolic execution and *lazy initialization* [25] to generate and thus count data structures that satisfy mixed heap and numeric constraints; we further use off-the-shelf model counting procedures [2] for the numeric constraints.

Lazy initialization extends symbolic execution with the ability of handling input data structures: it constructs the heap as the program paths are explored, and defers concretization of symbolic heap objects as much as possible. It produces symbolic heaps that are pairwise non-isomorphic while guaranteeing that no relevant states are missed. It can thus be used as a powerful procedure for generating and *counting* all the structures (up to a given bound). We further use relational field bounds [35] to reduce the search space for the solutions. Intuitively, field bounds restrict the number of choices that lazy initialization needs to consider when it concretizes a part of the heap.

We have implemented the model counting procedure in the Symbolic PathFinder tool-set [32] and have evaluated it on several complex data structure subjects from the literature. The experiments show that our proposed approach scales much better than an optimized enumeration-based method that uses the Korat tool [9]. The experiments also show the benefits of relational bounds on the overall cost of model counting.

2 Background

2.1 Symbolic Execution

Symbolic Execution [26,12] is a program analysis technique that executes programs on unspecified inputs, by using symbolic inputs instead of concrete data. The state of a symbolically executed program is defined by the (symbolic) values of the program variables, a *path condition* (PC), and a program counter. The path condition is a (quantifier-free) boolean formula over the symbolic inputs; it accumulates constraints on the inputs to follow that path. The program counter defines the next statement to be executed.

A *symbolic execution tree* characterizes the execution paths followed during symbolic execution. The tree nodes represent program states and the arcs the transitions between states due to the execution of program instructions. Typical applications of symbolic execution include test case generation and error detection, with many tools available [32,20,37,10]. Symbolic execution of looping programs may result in an infinite symbolic execution tree. For this reason, symbolic execution is typically run with a

(user-specified) bound on the search depth. Our work on probabilistic software analysis uses the symbolic execution tool Symbolic PathFinder (SPF) [32].
Lazy Initialization SPF uses lazy initialization [25] to handle dynamic input data structures (e.g., lists and trees). The components of the program inputs are initialized on an "as-needed" basis. The intuition is as follows. To symbolically execute method `m` of class `C`, SPF creates a new object o of class `C`, leaving all its fields uninitialized. When a reference field `f` of type `T` is accessed in `m` for the first time, SPF non-deterministically sets `f` to `null`, to a new object of type `T` with uninitialized fields, or to an alias to a previously initialized object of type `T`. This enables the systematic exploration of different heap configurations during symbolic execution. Here we will also consider an optimized form of lazy initialization called Bounded Lazy Initialization (BLISS) [35] that uses relational bounds and SAT solving to reduce the number of possible structures to consider. BLISS reduces the time and memory requirements over lazy initialization and therefore makes the techniques for counting discussed here tractable.

2.2 Probabilistic Software Analysis

We build on our previous work from [18,15,7], that uses symbolic execution for PSA. The goal of the analysis is: (1) to identify the symbolic constraints characterizing the inputs that make the execution satisfy a given property, and then (2) to quantify the probability of satisfying the constraints. For simplicity, we assume the satisfaction of the target property to be characterized by the occurrence of a target event (e.g. successful termination or failure), but our work extends to bounded LTL [41] as well.

The analysis works with a limited budget of symbolic paths, obtained with a bounded symbolic execution of the program. Some of these paths lead to failure and some of them to success (termination without failure). These path conditions are classified in two disjoint sets: $PC^s = \{PC^s_1, PC^s_2, \ldots, PC^s_m\}$ and $PC^f = \{PC^f_1, PC^f_2, \ldots, PC^f_p\}$. The path conditions may not cover the full input domain due to inherent incompleteness in the analysis, e.g. due to non-terminating loops or non-exhaustive path exploration. These remaining paths are called *grey* paths and are used in [15] to quantify the confidence one can put in the bounded symbolic analysis.

Probabilistic Usage Profiles The constraints generated with symbolic execution are analyzed to quantify the likelihood of an input to satisfy them, where the inputs are distributed according to given *usage profiles* [15]. A usage profile is a probabilistic characterization of the software interactions with the external world, e.g. the users or the physical execution environment. It assigns to each valid combination of inputs its probability to occur during execution. Usage profiles can be specified based on physical phenomena, known sensor parameters or other domain specific knowledge about the program and its deployment context. They can also be built automatically based on observed data from past usages of the program [19,5].

In [15], we assumed that all the input variables range over finite discrete domains, whose joining is generically indicated as D. We relaxed this assumption in more recent work [7]. We profile the expected usage for the program through a profile *UP*, which is a set of pairs $\langle c_i, p_i \rangle$ where c_i is a *usage scenario* defined as a (constraint representing

a) subset of D and p_i ($p_i \geq 0$) is the probability that a user input belongs to c_i. We further require, for simplicity, $\{c_i\}$ to be a complete partition of D, and thus $\sum_i p_i = 1$. Intuitively, *UP* is the distribution over the input space. Notice that c_i could contain even a single element of D, allowing for the finest grained specifications of *UP*.

Given the output of symbolic execution, the probability of success can be defined as the probability of executing the program (P) with an input satisfying any of the successful path conditions, given the profile *UP*. This definition can be formalized as $Pr^s(P) = \sum_i Pr(PC_i^s \mid UP)$. An analogous definition is provided for the probability of failure, $Pr^f(P)$. The probability of grey paths is $1-(Pr^s(P)+Pr^f(P))$ and it quantifies the ratio of elements of the input domain for which neither success nor failure have been revealed for the current analysis. This information is a measure of the confidence we can put on the probability estimation, under the current exploration bound.

Computing Probabilities with Model Counting To compute the probabilities of path conditions, we use a quantification procedure for the generated constraints. In [15] we used model counting techniques, i.e. LattE [14], to calculate the exact number of points of a bounded (possibly very large) discrete domain that satisfy linear constraints. Recently [7], we developed quantification procedures for the analysis of programs that have mixed integer and floating point constraints of arbitrary complexity.

To compute the probability of a path (described by PC) we use the fact that *UP* defines a partition of the input domain and then, from the law of total probability [33]:

$$Pr(PC \mid UP) = \sum_i Pr(PC \mid c_i) \cdot p_i$$

Furthermore, from the definition of conditional probability [33]: $Pr(PC \mid c_i) = Pr(PC \wedge c_i)/Pr(c_i)$.

To use model-counting techniques for the computation of the conditional probabilities, let us define for a constraint c the function $\sharp(c)$ that returns the number of elements of D satisfying c. $\sharp(\cdot)$ is always a finite non negative integer because we assumed D finite and countable. Under this same assumption, $Pr(c)$ is, by definition [33], $\sharp(c)/\sharp(D)$ (where $\sharp(D)$ is the size of the non-empty input domain). Thus, one can express the probability of success as:

$$Pr^s(P) = \sum_i Pr(PC_i^s \mid UP) = \sum_i \sum_j Pr(PC_i^s \mid c_j) \cdot p_j = \sum_i \sum_j \frac{\sharp(PC_i^s \wedge c_j)}{\sharp(c_j)} \cdot p_j$$

3 Approach

We describe here how the probabilistic software analysis is extended to handle programs that take as input structured data types, e.g. lists or trees.

3.1 Usage Profiles

Usage profiles (UP) for data structures are defined with the help of Java predicates (i.e., boolean methods) that define data structure properties that *partition* the input state space. To each element of this partition a probability value is assigned, with the sum of those values being equal to 1. For example, for a program with an input list, the UP may specify that the input list is non-null 90% of the time (and null 10%). Alternatively, the UP may specify that the list is acyclic say 95% (and cyclic 5%), or that the list is "small" (number of nodes less than 10) most of the time (90%) and "large" (number of nodes greater than 10) rest of the time (10%) etc.

As before, we restrict ourselves to finite input domains, which for data structures also lead to a limited number of possible heap nodes for the input. It is the responsibility of the user to ensure that the predicates in the UP define a partition of the input domain (i.e. a division of the domain as the union of non-overlapping non-empty subdomains).

3.2 Symbolic Constraints

SPF can analyze programs with unbounded data structures as inputs, using lazy initialization [25]. The result of symbolic execution is a set of paths, each characterized by a path condition that encodes both numeric and heap constraints.

The numeric constraints are generated whenever a branching condition on primitive (numeric) fields is evaluated. The heap constraints are generated during the lazy initialization of instructions that perform a first access to an uninitialized field (i.e., bytecodes `aload`, `getfield`, and `getstatic`).

The heap constraints can have the following forms:

– $ref = null$. Reference *ref* points to *null*.
– $ref \neq null$. Reference *ref* is non *null*.
– $ref_1 = ref_2$. References ref_1 and ref_2 are aliased (point to the same object).
– $ref_1 \neq ref_2$. References ref_1 and ref_2 are not aliased.

Example Consider the Java code in Listing 1.1 [25] that declares a class `Node` for a linked lists. Fields `elem` and `next` represent the node's integer value and a reference to the next node in the list, respectively. Method `swapNode` destructively updates its input list, referenced by the implicit parameter `this`, according to a numeric condition on the first two nodes.

Listing 1.1: List example.

```java
class Node {
  int elem;
  Node next;

  Node swapNode() {
    if(elem > next.elem) {
      Node t = next;
      next = t.next;
      t.next = this;
      return t;
    }
    return this;
  }
}
```

Symbolic execution with lazy initialization results in seven symbolic paths (see Figure 1), due to the `if` condition and the different aliasing possibilities in the input

PC_1: $in.next = null \wedge in \neq null$
PC_2: $in.next = in \wedge in \neq null$
PC_3: $in.next \neq in \wedge in.next \neq null \wedge in \neq null \wedge in.elem \leq in.next.elem$
PC_4: $in.next.next = null \wedge in.next \neq in \wedge in.next \neq null \wedge in \neq null \wedge in.elem > in.next.elem$
PC_5: $in.next.next = in \wedge in.next \neq in \wedge in.next \neq null \wedge in \neq null \wedge in.elem > in.next.elem$
PC_6: $in.next.next = in.next \wedge in.next \neq in \wedge in.next \neq null \wedge in \neq null \wedge in.elem > in.next.elem$
PC_7: $in.next.next \neq in \wedge in.next.next \neq in.next \wedge in.next.next$
$\neq null \wedge in.next \neq in \wedge in.next \neq null \wedge in \neq null \wedge in.elem > in.next.elem$

Figure 1: Symbolic paths from method swapNode.

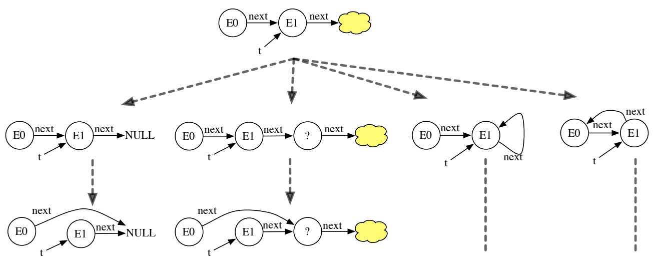

Figure 2: Lazy Initialization.

These symbolic execution paths together represent all possible actual executions of `swapNode`. The PCs represent an isomorphism partition of the input space, e.g., PC_7 describes all (cyclic or acyclic) input lists with at least three nodes such that the first element is greater than the second element. The analysis reports a *failure* for PC_1 – the method raises an unhandled `NullPointerException`. There are no grey paths (since there are no loops).

As an illustration of lazy initialization consider the symbolic execution of `next = t.next;` for the symbolic heap configuration depicted in the root of the tree in Figure 2. In the figure a "blob" indicates field `next` pointing to it is uninitialized (it has not been accessed yet by the symbolic execution along this path). "E0" and "E1" represent some fresh symbolic values for the numeric field `elem`; a "?" means that a field of numeric (or other primitive) type has not been initialized yet. Dashed arrows depict a branching of nondeterministic choices, describing all the possible instantiations of the symbolic structure. Since `t.next` is uninitialized, SPF uses "lazy initialization" to assign it either `null`, a new symbolic object with uninitialized fields, or an object created during a previous initialization (resulting for our example in two instances of circular lists). Intuitively, this means that SPF makes four different assumptions about the shape of the input list according to different aliasing possibilities and it explores all of them systematically. The PCs are updated according to these choices. Once `t.next` has been initialized the execution proceeds according to the Java semantics.

3.3 Model Counting For Data Structures

Though the counting-based probabilistic analysis method from Section 2 can be applied on any finite input domain, we need an efficient procedure for counting data structures.

In the worst case a complete (and expensive) enumeration of all the possible input instances (up to a pre-specified bound) might be performed. A less expensive alternative, that we proposed in [15] is to use Korat [9] for the data structure enumeration. Korat is a tool that performs constraint-based generation of structurally complex test inputs for Java programs. Korat's goal is to systematically generate all complex test data structures (within prescribed bounds) for the purpose of testing. Although Korat was not designed for model counting we can use it to compute all input data structures that satisfy a complex predicate within pre-defined bounds. The predicate is written as a boolean method often called `repOk`, whose body can embed any arbitrarily complex computation. The *finitization* of the input domain is accomplished by specific Korat methods to specify bounds on the size of input data structures as well as on the domain of primitive fields.

Thus we can encode the constraints provided by symbolic execution together with the constraints from the usage profile as a `repOK` predicate and run Korat to count the data structures that satisfy the constraints for the given finitization. Experiments with this approach (see Section 4) show that it often scales poorly when the path constraints contain a combination of heap and numeric constraints, and the numeric domains are very large. This is due to the enumeration of the valid values for integer fields performed by Korat. In the next section we propose an efficient alternative method.

3.4 Model Counting using Lazy Initialization

We propose to use symbolic execution with lazy initialization to efficiently generate and count the data structures that satisfy given constraints. The core insight is to use lazy initialization to enumerate the structures, but to keep the constraints on the numeric fields of the structures *symbolic*; the valid assignments for these symbolic fields can then be counted with an off-the-shelf model counting procedure, such as LattE [2]. LattE implements Barvinok's algorithm [4], which constructs a generating function suitable for determining the number of points within a convex polytope without enumeration.

To count the number of solutions to a set of mixed heap and numeric constraints, we apply symbolic execution with lazy initialization to a boolean method encoding the constraints (similar to the `repOk` method in Korat). The model counting procedure requires thus two inputs:

Method `repOk`: Java boolean method encoding the constraints; returns true if the structure satisfies the constraints (e.g. the list is acyclic).

Finitization: Domain bounds for both reference and numeric data (e.g., a list may have up to 5 nodes, whose elements are between 1 and 10).

For example, if we want to count all the acyclic lists having at most 6 nodes, whose elements are between 1 and 10, we would use the code reported in Listing 1.2.

Listing 1.2: Counting acyclic lists.

```
class List{
  Node head;
  boolean repOkacyclic(){
    Set<Node> nodes = new HashSet<Node>();
    Node iterator = head;
    while(iterator!=null){
      // check acyclic
      if(!nodes.add(iterator))
9         return false;
10      //check bounds
11      if(iterator.elem<1||iterator.elem>10)
12        return false;
13      if(nodes.size>6)
14        return false;
15
16      iterator=iterator.next;
17    }
18    return true;
19  }
20
21  public static void main(String[] args){
22    List L0 = new List();
23    L0=(List) Debug.makeSymbolicRef("L0",L0);
24    if(L0!=null)
25      assert (L0.repOkacyclic());
26  }
27 }
```

The symbolic execution of the program in Listing 1.2 collects as successful path conditions (i.e. not leading to an exception) all the symbolic structures representing an acyclic list with at most 6 nodes, whose elements are integers between 1 and 10.

The main method summarizes the steps required for counting these structures:

1. Create a symbolic instance of the structure. In SPF syntax, see lines 22 and 23.
2. Execute the method `repOkacyclic`. This step drives the execution toward exploring all the valid structures, expanding and concretizing all of them, i.e. leaving only symbolic numeric variables to be analyzed.
3. Preempt the execution from exploring structure outside the valid domain or not satisfying the constraints. For brevity, we just assert the `repOkacyclic` predicate.

The total number of acyclic lists can thus be obtained applying established model counting solutions on the success path conditions, which now predicate only on the numeric fields. The result in this case would be 6,543,210 acyclic lists out of 7,654,321 lists with up to 6 nodes (and elements between 1 and 10).

Note that encoding the `repOk` is subtle, as it not only encodes the given constraints but it also includes code to enumerate all the structures up to bounds given in the finitization. Similar to Korat, the structure of the `repOk` is crucial to the efficiency of the method. If `repOk` would first enumerate all structures and only then determine if they are valid (according to the given constraints) our approach would not benefit from lazy initialization (but it would still benefit from solving the numeric constraints separately).

In our implementation we provide a code skeleton for enumerating all data structures (to which users can add their constraints). Input bounds are provided in a configuration file. Internally SPF backtracks when the bounds on heap nodes are reached. Bounds on numeric fields are fed directly to the constraint solvers.

3.5 Probabilistic Software Analysis

Counting the instances of a data structure satisfying a given predicate enables us to compute the probability of target program events to occur, given a specific usage profile.

As an example let us compute the probability of failure (in this case, throwing a `NullPointerException`) when executing the `swapNode` method of Listing 1.1. Assume a usage profile that specifies that the input list is acyclic with probability 0.9

and it is cyclic with remaining probability 0.1. There is only one failure symbolic path (revealed by a null pointer exception in the evaluation of the if condition). The path condition for the failure path, as revealed by SPF, is

$$input \neq null \wedge input.next = null$$

Since this path condition is only satisfiable for acyclic lists, we get the probability of failure $Pr^f(P)$:

$$0.9 \cdot \frac{\sharp(input \neq null \wedge input.next = null \wedge acyclic(input))}{\sharp(acyclic(input))}$$

The results of model counting are $\sharp(input \neq null \wedge input.next = null \wedge acyclic(input)) = 10$ and $\sharp(acyclic(input)) = 1,111,111$, for lists with up to 6 nodes and `elem` ranging over 1..10, giving probability of failure $8.1 \cdot 10^{-6}$.

One can argue that we should simply correct the error in method `swapNode` (for example adding a `null` check). However imagine a scenario where this method is part of a large code base and that usage (calling context) of the method indicates that the probability of the list being null is very small. In such cases PSA becomes very useful, for example, to “rank” the errors according to the likelihood of occurrence, allowing developers to focus on high probability errors first. More example applications of PSA will be discussed in Section 4.

Embedded usage profiles. In the computation above we have followed the approach in [15] and computed the effects of the UP after the path constraints have been collected. An alternative way introduced in [28] consists in embedding the usage profile as “preconditions” (assume statements) in the code. Listing 1.3 shows an example of embedded UP for the analysis of the `swapNode` method. For the usage profile that states that the input list is acyclic 90% of the time (and cyclic 10%) We use a symbolic variable, *up*, uniformly distributed in the range $1 \leq up \leq 100$, for controlling the distribution of the input values.

Listing 1.3: Embedded UP for the List example.

```
List L0 = new List();
L0=(List)Debug.makeSymbolicRef("L0",L0);
if(up<=90){
      Debug.assume(L0!=null && L0.repOkacyclic());
}else{
      Debug.assume(L0!=null && L0.repOkcyclic());
}
L0.swapNode();
```

The assume statements are implemented using the built-in `Debug.assume()` method from SPF [32]. The failure probability can then be computed using model counting for the numeric constraints encoded in the path conditions for the failure paths.

Both ways for handling UPs are supported in our tool with analogous performance overhead, leaving to the developer the choice whether keeping the UP and the code separated or included in the same file.

3.6 Optimizations

In this section we describe optimizations included in our analysis that allow us to improve scalability.

BLISS (Bounded Lazy Initialization with Sat Support) [35], is an optimization specifically tailored to improve the lazy initialization of data structures during symbolic execution. Data structures usually obey strong restrictions on their structure and stored data, under the form of *class invariants*. Some typical invariants are “the items in this list are sorted”, or “if a node is red, then both its children are black" (for red–black trees). BLISS exploits known class invariants to compute *tight bounds* on the data structure fields. Intuitively, a tight field bound is a relational upper bound (set of pairs) on the (relational) semantics of Java class fields. A tight field bound for a Java field f, is a binary relation between unique field identifiers that only relates pairs $\langle i_1, i_2 \rangle$ that are feasible, i.e., for which there exists a structure satisfying both the class invariant and the canonical labeling of identifiers, that includes in the memory heap objects with identifiers i_1 and i_2 such that $i_1.f = i_2$.

During lazy initialization, whenever an object o is dereferenced through a (symbolic) object field f, three possibilities have to be considered, namely [25]:

- $o.f$ is initialized as the `null` value,
- $o.f$ is initialized as a pre existing object o' in the memory heap, and
- $o.f$ is initialized as a new object o''.

The tight field bounds allow to reduce the choices in the first and second case (the latter being the most expensive). BLISS prunes those symbolic executions where the (partially) symbolic memory heap contains enough information to determine it can not be extended into a feasible heap. Intuitively, the concrete parts of the partially symbolic heap are translated as constraints that are conjoined with an automatically generated propositional description of the class invariant. A satisfiability checker is used to determine whether the symbolic parts of the heap can be concretized into a fully concrete memory heap satisfying the class invariant. Those partially symbolic heaps producing a negative outcome can be safely pruned from the symbolic execution process. BLISS can improve lazy initialization significantly [35] and occupies a natural place in the context of this work.

As already mentioned, in this paper we focus on integer constraints, whose models are counted with Latte [2] (to cope with floating-point numbers and nonlinear constraints, it is straightforward to use qCoral [7] in place of Latte). The complexity is in terms of the number of variables and the number of constraints. For large constraints, the procedure could be very time consuming. We address this problem by first simplifying the constraints and using a divide-and-conquer approach [15] that divides large path conditions into *independent constraints* which can then be solved independently. Intuitively two constraints are independent if the sets of variables they constrain have no intersection. The approach facilitates caching and reusing counting results.

4 Implementation and Experience

In this section we report an experimental comparison of an implementation of our approach with Korat (Section 4.1) and a set of case studies demonstrating its applicability for probabilistic software analysis (Section 4.2).

We implemented our approach on top of SPF [32] v6. The collection of path conditions followed by the probability computations are implemented by means of JPF listeners. Experiments were performed on a workstation with Intel Core i7-2600 processor with a 3.40 GHz clock speed and 8 GB DDR3 RAM, running Linux 3.2.0. 6 GB of heap memory were allocated for the Java virtual machine. All the times are in seconds. TO means the execution has been interrupted after a timeout of 5 hours. OOM means the execution ran out of memory.

4.1 Comparing with Korat

We compared the result and execution time of our approach, with and without BLISS (denoted SPF and SPF+BLISS), versus Korat on counting the valid instances of four known data structures showing different complexity dimensions: linked list, red-black tree, binomial heap and AVL tree. We vary the number of nodes in each structure and the size of the domain of values that can be stored. We remark that Korat has not been designed for model counting, but its smart enumeration capabilities can be used for this goal [15] making it a good baseline for comparison. For BLISS we used the field bounds from previous studies.

The results are reported in Table 1. The columns SPF, SPF+BLISS, and Korat report the analysis time for our approach with and without BLISS and for Korat, respectively.

LinkedList. This data structure implements a doubly linked list where each node contains an integer field. Korat fails to explore the whole input domain within 5 hours for all the cases where the list was composed of 10 nodes and the integer domain contained 20 or more elements. On the other hand, SPF-based analyses terminate in less than 2 seconds for all the considered cases. Notably, due to the simplicity of this structure, the benefit of adding BLISS does not yield any perceivable improvements.

RedBlackTree. Red-black trees are significantly more complex than linked lists, both because of the higher number of references involved and the preservation of their invariants, which requires rebalancing techniques to guarantee the red-black property [13]. The main bottleneck of Korat remains on the size of the integer domain, when the number of nodes grows. On the other hand, for smaller integer domains the increased complexity of the structure has a modest impact on the performance of Korat. The number of nodes has instead a significant impact on the performance of SPF-based tools, with SPF running out of memory already with 8 nodes. Introducing BLISS reduces significantly the execution time and memory consumption in this case, since it prevents the symbolic execution to explore unnecessary invalid structures. This allows it to cope with larger instances.

BinomialHeap. Despite being operationally simpler than red-black trees, binomial heaps can also be characterized by a set of invariants making BLISS more effective in detecting invalid structures before their complete exploration. This results in a shorter execution time of SPF+BLISS with respect to SPF. Notice how SPF and SPF+BLISS scale better than Korat even for small sizes of the integer domain.

AVLTree. AVL trees are search trees whose rebalancing is triggered by the violation of a simpler invariant than red-black trees (the heights of the subtrees of every node can differ by at most one). In this case BLISS produces a smaller improvement compared to the case of red-black trees, though still reducing the analysis time of SPF. Korat achieves a good scalability over this structure, though if does not scale for larger instances where the integer domain has 20 elements or more.

Table 1: Comparison of SPF, SPF+BLISS, and Korat (time in seconds).

	Nodes	**Ints**	**Count**	**SPF**	**SPF + BLISS**	**Korat**
LinkedList	5	20	168,421	0	0	0
		50	6,377,551	0	0	3
		70	24,357,971	0	0	13
		100	101,010,101	0	0	51
	10	10	1,111,111,111	1	1	986
		20	538,947,368,421	1	1	TO
		50	1,992,984,693,877,551	1	1	TO
		70	40,938,441,884,057,971	1	1	TO
		100	1,010,101,010,101,010,101	1	1	TO
	15	10	1,111,111,111	1	1	TO
		20	1,724,631,578,947,368,421	1	1	TO
		50	622,807,716,836,734,693,877,551	1	1	TO
		70	68,805,239,274,536,231,884,057,971	1	1	TO
		100	10,101,010,101,010,101,010,101,010,101	1	1	TO
	Nodes	**Ints**	**Count**	**SPF**	**SPF + BLISS**	**Korat**
RedBlackTree	5	10	3,197	27	8	0
		20	146,093	26	8	3
		50	17,912,981	27	8	363
		70	100,606,073	26	8	476
		100	618,318,461	26	8	2,796
	8	10	13,037	OOM	300	1
		20	10,378,733	OOM	358	238
		50	33,633,553,781	OOM	354	TO
		70	570,417,679,113	OOM	366	TO
		100	10,968,862,744,061	OOM	357	TO
	10	10	14,101	OOM	2,738	8
		20	55,795,117	OOM	2,754	2,936
		50	1,943,776,206,661	OOM	2,841	TO
		70	71,482,977,220,937	OOM	2,742	TO
		100	3,021,060,476,356,221	OOM	2,774	TO
	Nodes	**Ints**	**Count**	**SPF**	**SPF + BLISS**	**Korat**
BinomialHeap	5	6	2,016	2	1	0
		10	19,371	2	1	0
		20	497,616	2	1	0
		50	42,613,101	2	1	28
		70	223,543,216	2	1	152
		100	1,305,473,076	2	1	880
	10	11	276,834,504	30	13	382
		20	70,790,816,523	30	13	TO
		50	482,258,613,959,406	30	12	TO
		70	13,057,541,269,423,978	30	13	TO
		100	439,699,627,791,397,061	30	13	TO
	15	16	1,320,960,601,687,363	562	129	TO
		20	31,844,676,603,881,568	559	128	TO
		50	19,743,228,678,771,046,522,656	562	130	TO
		70	2,836,624,163,763,256,508,895,748	560	133	TO
		100	562,643,897,792,832,103,640,559,436	569	129	TO
	Nodes	**Ints**	**Count**	**SPF**	**SPF + BLISS**	**Korat**
AVLTree	8	4	25	47	39	0
		10	6,893	66	61	0
		20	5,617,865	66	59	92
		50	18,955,370,261	69	59	TO
		70	323,071,208,925	67	58	TO
		100	6,232,176,942,521	67	58	TO
	10	4	25	190	139	0
		10	7,393	303	249	2
		20	24,093,465	308	253	932
		50	745,531,143,261	303	253	TO
		70	26,986,817,918,525	307	255	TO
		100	1,128,548,943,898,521	304	252	TO
	13	4	25	707	423	0
		10	7,393	2,804	2,408	17
		20	95,928,665	3,318	2,983	TO
		50	194,611,435,515,261	3,362	3,000	TO
		70	24,729,749,799,273,725	3,343	2,977	TO
		100	3,588,938,338,577,002,521	3,330	2,951	TO

In summary, for SPF-based techniques varying the size of the integer domain does not produce significant variations in the analysis time, while an enumeration-based approach unable to symbolically abstract the integer fields of the structures suffers scalability issues even for relatively small integer domains (20 or 50 elements). On the other hand, the complexity of the references structure is the main bottleneck of SPF-based tools, which are required to enlarge the scope of symbolic execution, reducing the benefits of lazy initialization. The use of BLISS is particularly beneficial when rich invariants (which impose strong requirements on structures) are available, allowing to prune symbolic execution paths heading toward the exploration of invalid structures.

4.2 Probabilistic Analysis

Probabilistic software analysis can be used for answering questions like:

1. *Which program methods are worth focusing on to improve the software responsiveness perceived by the users?*
2. *How likely is a bug to show up when the program is used according to a specific profile?*
3. *What is the perceived reliability of software for different classes of usages?*

In this section we report three example applications of our probabilistic software analysis technique by casting these question on small program snippets manipulating data structures to show the applicability scope of this technique.

1. Rotations in a red-black tree. Red-black trees are kept *almost balanced* after every insertion or deletion [13]. This is achieved by a potentially expensive rotation operation. Considering the insertion of an integer value within the range $0 - 20$ into a tree having from 0 to 4 nodes, *how frequently should we expect a rotation will be required to rebalance the tree?*

Though this is just an example, answering this kind of question allows one to quantify the frequency a certain method is expected to be invoked during a program execution. This would help assessing the global impact of improving the efficiency of a specific method, and support the decisions of a developer.

In this example, the problem space is given by the set of all the valid trees with up to 4 nodes and the finite subset of integers between 0 and 20. This space counts $567,882$ elements. Out of them, inserting the integer value into the tree requires at least one rotation operation in $168,112$ cases, about 29.6%.

This type of information can also be exploited to compute a complexity index for operations on data structures tailored to specific usage profiles the program is expected to handle.

2. Assessing the criticality of an actual bug. Class BinomialHeap used as part of the examples in this paper was first used as part of a benchmark in [39]. In [17] it was determined that method `extractMin` had a subtle bug that required a binomial heap with at least 13 nodes to be exposed. An example of an input exposing this bug is given in [17, Fig. 6]. A consequence of this bug is that upon execution of `extractMin`, the resulting binomial heap no longer satisfies its required property (attribute size no longer reflects the actual number of nodes in the binomial heap).

Although there is a bug in this structure, *how likely is this bug to actually show up when the* `extractMin` *method is invoked?*

Table 2: Number of inputs exposing the extractMin bug.

Nodes	Ints	# Valid	# Failing	Fail prob	Time (s)
12	0..12	70,401,948,540	0	0	28 + 74
	0..20	14,829,486,591,568	0	0	28 + 73
	0..30	1,269,649,449,162,048	0	0	28 + 71
13	0..13	1,921,213,899,450	1,546,032,456,492	0.804	49 + 151
	0..20	278,713,724,302,816	235,789,399,182,528	0.845	49 + 142
	0..30	36,285,348,047,086,752	31,636,080,812,285,208	0.871	48 + 147

In order to count the number of inputs that lead to a failure state (one in which attribute size does not model the actual number of nodes in the resulting binomial heap), we analyzed the code in Listing 1.4.

Listing 1.4: Bug in BinomialHeap (BH).

```
public static void main(String[] args) {
  BH B0 = new BH();
  B0 = (BH) Debug.makeSymbolicRef("B0", B0);
  if (B0 != null && B0.repOk()){
    B0.extractMin();
    assert B0.size == B0.numNodes());
  }
}
```

Executing symbolically the main method allows the `repOk` to generate all valid structures. Those structures that violate the assert statement generate errors that are caught by the underlying JVM, which then stores the numeric path conditions for further counting of failing instances. Table 2 presents our results.

When 12 or less elements are inserted in the heap, the bug will never show up (confirming previous evaluations regarding this bug [23,35]). So users following this behavior will not notice the presence of the bug.

When at least 13 elements are inserted, there is the chance for the bug to show up. However, how likely this is to occur in practice heavily depends on the size of the domain allowed for the integer values. Indeed, the bug does not systematically occur for every possible set of elements. Looking at Table 2, when only integers between 0 and 13 are allowed (with each value having the same probability), more than 80% of the executions will violate the assertion. These figures can also be used to assed the difficulty of catching such bug with naive randomized testing.

In Table 2, we report for each given numbers of nodes and integer values, the number of valid inputs in the state space, the number of inputs leading to a failure, as well as the probability of running into a faulty outcome. Running time is presented under the form $t_1 + t_2$, with t_1 the time required to compute the number of valid inputs, and t_2 the time required to compute the failing ones. Notice also in this case how times increase as the number of nodes increases, yet remain stable for a number of nodes despite the number of integer values considered.

3. Impact of different usage profiles. In the following we consider the impact of different usage profiles on the running example of the *List* from Listing 1.1. We consider the case where we have at most 6 nodes and numeric values in the range $1..10$.

In Section 3.5 we evaluate the probability of throwing an exceptions when executing the method `swapNode` on the List example. The usage profile we considered was: 10%

cyclic lists and 90% acyclic lists. Since we had $1,111,111$ acyclic lists and for 10 cases of these the exception is thrown (see Section 3.5), while none of the $6,543,210$ acyclic ones lead to an exception, the failure probability under this profile can be computed as:

$Pr^f(P) = .10 \cdot 0/6543210 + .9 \cdot 10/1111111 = 8.1 \cdot 10^{-6}$

How does this probability change if the input lists were distributed differently? Let us consider the case where we have 90% chance of a list being not null and 10% chance that the list is null. Obviously there is only one list that is null and the remaining $7,654,320$ cases are not null. Therefore, we obtain the following probability for failure:

$Pr^f(P) = .9 \cdot 10/7654320 + .1 \cdot 0/1 = 1.1758 \cdot 10^{-6}$

The last case is where we use the length of the list in the usage profile. Let us consider there is an 80% chance that the list length is less than 4, and a 20% chance the list has at least 4 nodes (and no more than 6 as per the finitization). There are $4,321$ lists with up to 3 nodes and $7,650,000$ lists of size 4 and more. Notice that none of the lists with 4 or more nodes can cause an exception. The probability for exception is thus:

$Pr^f(P) = .8 \cdot 10/4321 + .2 \cdot 0/7650000 = 1.85 \cdot 10^{-3}$

Concluding, the different usage profiles make a substantial difference in the probability of an exception being thrown for the analyzed null pointer dereference. This illustrates the importance of usage profiles when performing probabilistic software analysis, which is in turn able to provide quantitative results tailored for each different (probabilistic) assumption about the usage of the software.

5 Threats to Validity

We used data structures as examples. These and similar examples have been frequently used as case studies in the evaluation of SPF and come as examples with the Korat distribution, making them appropriate for the comparison.

Computing bounds and writing declarative invariants pose extra burden on the users of SPF+BLISS. This is not a part of the technique, and the user may decide not to use the BLISS optimization. Yet BLISS naturally fits in this research as one may conclude from the experiments reported in Table 1.

We did not verify the implementation. For all the subjects where at least two of the three methods completed the analysis within the time bound of 5 hours, the resulting counts matched, cross validating their correctness for the cases under investigation.

6 Related Work

Several model counting tools are available but they do not support data structures directly. Birnbaum et al. [6] present an algorithm for counting (boolean) models of propositional formulas. Barvinok's algorithm uses Integer Linear Programming (ILP) to count integer models [4]. As already mentioned, LattE[14] implements (an enhanced version of) Barvinok's algorithm. RelSat solves instances of propositional SAT using constraint satisfaction problem (CSP) look-back techniques [1].

Several (dynamic) symbolic execution techniques encode data-structure constraints using a theory of select/store (e.g. KLEE [10]). In such techniques there is no need to explicitly initialize the references as they can deal with symbolic references. Note however that the counting of data-structure models can not be done simply on the symbolic formulas, using e.g. [11] for counting over SMT constraints. E.g. one can not simply count all the (cyclic and acyclic) lists up to size 100 by applying SMT-based model counting over a constraint that encodes “true”. Instead, our procedure, that blends explicit enumeration with symbolic reasoning, could be used.

The SMC tool [29] addresses constraints on strings. It counts model for constraints written in a string language expressive enough to model constraints arising from JavaScript applications and UNIX C utilities. It uses a technique that leverages generating functions as a basic primitive for combinatorial counting, and it is therefore quite different than our approach, which aims at handling arbitrary data structures.

Our work is also related to probabilistic program analysis [21], probabilistic abstract interpretation [30] and probabilistic model checking [22]. We discuss this below.

Probabilistic analysis based on symbolic execution has been described in e.g., [18,36,15]. Geldenhuys et al. [18] considered uniform distributions for the inputs, linear integer arithmetic constraints, and used LattEMacchiato [14] to count solutions of path conditions produced during symbolic execution. Sankaranarayanan et al. [36] and Filieri et al. [15] proposed similar techniques to compute probabilities of violating program assertions. Both techniques remove the restriction of uniform distributions. As with [18] both approaches only consider linear constraints. Sankaranarayanan et al. developed algorithms based on Linear Programming (LP) solvers for under and over-approximations of probabilities. Filieri et al. used the LattEtool to compute probabilities. Follow-on work provides treatment of nondeterminism [28] and describes statistical exploration of symbolic paths [16]. Another simulation-based approach for the analysis of probabilistic programs has been proposed in [31].

The technique in [7] proposed a compositional quantification of the solution space based on Monte Carlo estimation. The approach can deal with arbitrarily complex numeric constraints over floating-point domains. Bouissou *et al.* [8] and Adje *et al.* [3] handle non-linear numeric constraints with a combination of abstraction based on affine and p-box arithmetic. The approach relies on the use of noise variables to represent the uncertainty of non-linear computations.

Lazy initialization is related to materialization of summary nodes in shape analysis [40]. However its application to model counting is new.

7 Conclusions

We presented an technique for model counting over constraints on complex data structures with numeric fields. The technique uses symbolic execution with lazy initialization to compute the satisfying heap structures, while keeping the constraints on numeric data symbolic. The valid assignments for the numeric constraints are then solved with off-the-shelf model counting procedures that target numeric domain. Further field bounds and various constraint optimizations are used to speed-up the technique. Experimental results highlighted the benefits of the proposed technique.

There are many avenues for future work. First note that it is the responsibility of the user to write the complex (Java) predicates; further the user needs to make sure that the predicates in the usage profile are disjoint. To ease this burden we have defined patterns for some commonly used predicates (such as acyclic and size for linked lists) that can be used and modified easily. In the future we would like to explore established logics, such as separation logic, to simplify the specification task. We will then need to synthesize the Java predicates encoding them. We also plan to explore runtime analysis to derive profiles directly from running systems. Further we plan to apply the model counting technique in the security domain.

References

1. RelSat tool. `http://code.google.com/p/relsat/`.
2. LattE, 2013. `https://www.math.ucdavis.edu/~latte/`.
3. Assale Adje, Olivier Bouissou, Jean Goubault-Larrecq, Eric Goubault, and Sylvie Putot. Static analysis of programs with imprecise probabilistic inputs. In *VSTTE*, volume 8164, pages 22–47. Springer, 2014.
4. Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, November 1994.
5. Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. ESEC/FSE, pages 267–277, 2011.
6. Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *J. Artif. Intell. Res. (JAIR)*, 10:457–477, 1999.
7. Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *PLDI*, pages 123–132. ACM, 2014.
8. Olivier Bouissou, Eric Goubault, Jean Goubault-Larrecq, and Sylvie Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94:189–201, 2012.
9. Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. ISSTA, pages 123–133, 2002.
10. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
11. Dmitry V. Chistikov, Rayna Dimitrova, and Rupak Majumdar. Approximate counting in SMT and value estimation for probabilistic programs. TACAS, pages 320–334, 2015.
12. Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Soft. Eng.*, 2(3):215–222, 1976.
13. T.H. Cormen. *Introduction to Algorithms, 3rd Edition:*. MIT Press, 2009.
14. J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, and J. Wu. Software for Exact Integration of Polynomials Over Polyhedra. *ACM Commun. Comput. Algebra*, 45(3/4):169–172, 2012.
15. Antonio Filieri, Corina S. Pasareanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *ICSE*, pages 622–631, 2013.
16. Antonio Filieri, Corina S. Pasareanu, Willem Visser, and Jaco Geldenhuys. Statistical symbolic execution with informed sampling. In *FSE*, pages 437–448, 2014.
17. Juan P. Galeotti, Nicolas Rosner, Carlos G. L´opez Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. ISSTA, pages 25–36, USA, 2010.
18. Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.

19. Carlo Ghezzi, Mauro Pezzè, Michele Sama, and Giordano Tamburrelli. Mining behavior models from user-intensive web applications. In *ICSE*, pages 277–287, 2014.
20. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.
21. Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *ICSE FOSE*, pages 167–181, 2014.
22. Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444. 2006.
23. Galeotti J., Rosner N., Lopez Pombo C., and Frias M.F. Taco: Efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Soft. Eng.*, 39(9):1283–1307, sept 2013.
24. Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.
25. Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. TACAS, pages 553–568. Springer Berlin Heidelberg, 2003.
26. James C. King. Symbolic execution and program testing. *Comm. ACM*, 19(7):385—394, July 1976.
27. Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET*, pages 99–102, 2003.
28. Kasper Luckow, Corina S. Păsăreanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. ASE, pages 575–586. ACM, 2014.
29. Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. PLDI, page 57, 2014.
30. David Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *POPL*, pages 93–101, 2001.
31. Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: An efficient mcmc sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI, July 2014.
32. 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.
33. W.R. Pestman. *Mathematical Statistics*. De Gruyter, 2009.
34. Quoc-Sang Phan, Pasquale Malacaria, Corina S. Păsăreanu, and Marcelo D'Amorim. Quantifying information leaks using reliability analysis. SPIN, pages 105–108. ACM, 2014.
35. N. Rosner, J. Geldenhuys, N. Aguirre, W. Visser, and M.F. Frias. Bliss: Improved symbolic execution by bounded lazy initialization with sat support. *IEEE Trans. Soft. Eng.*, (99), 2015.
36. Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, 2013.
37. Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.
38. Alexandru Turjan, Bart Kienhuis, and Ed F. Deprettere. A compile time based approach for solving out-of-order communication in kahn process networks. In *ASAP*, pages 17–28, 2002.
39. Willem Visser, Corina S. Pasareanu, and Radek Pelanek. Test input generation for Java containers using state matching. ISSTA, pages 37–48, 2006.
40. Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, pages 530–545, 2004.
41. Paolo Zuliani, André Platzer, and Edmund M. Clarke. Bayesian statistical model checking with application to simulink/stateflow verification. In *HSCC*, pages 243–252. ACM, 2010.