Karlsruhe Reports in Informatics 2015,2
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Incremental and Compositional Probabilistic Analysis of Programs

Fouad ben Nasr Omri, Safa Omri, and Ralf Reussner

2015

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Incremental and Compositional Probabilistic Analysis of Programs under Uncertainty

Fouad ben Nasr Omri Safa Omri Ralf Reussner

Software Design and Quality

Institute for Program Structures and Data Organization, Faculty of Informatics
Karlsruhe Institute of Technology, Germany

fouad.omri@kit.edu safa.omri@student.kit.edu ralf.reussner@kit.edu

Symbolic execution has been applied, among others, to check programs against contract specifications or to generate path-based test suites. We propose to adapt symbolic execution to perform a probabilistic reasoning about possible executions of a program. We present a compositional and incremental approach to approximate the probability of a program path.

1 Introduction

Uncertainty is a common aspect of modern software systems. The growing complexity of interaction with third-party components, the heterogeneity of the behavior of users and the possible external and environmental disturbances (e.g., operating with sensor errors, robotic manipulators, etc.,) are introducing an uncertainty about the behavior of a program. Understanding the behavior of a program is essential to test it. In most cases, the tester is interested in knowing whether a behavior or a target event (e.g., invocation of a certain method, access to confidential data, uncaught exceptions, etc.,) can happen or not. However, we believe that we can better understand a program behavior when we know how *probable* a behavior can occur. We also believe that the uncertainty in the program inputs should be considered when analyzing a program's behavior.

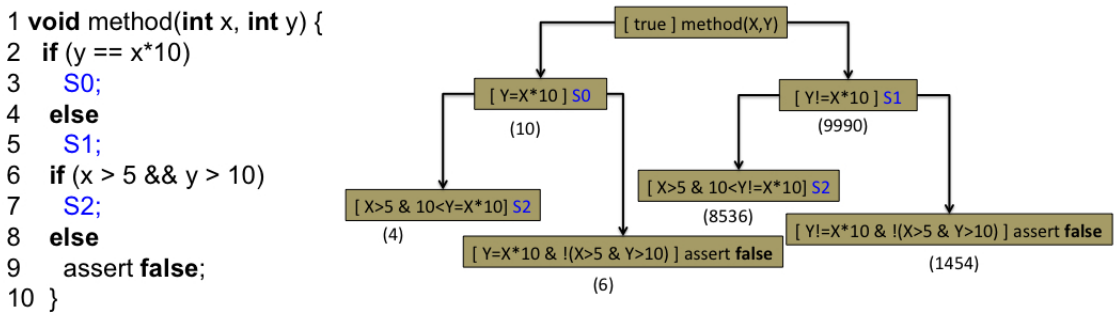
Static analysis techniques aim at obtaining information about the possible behavior of a program, without running it on concrete inputs. A program behavior is one or more program paths executed with input values from the programs input domain. A program path is a sequence of statements. Each program path has an input and an executed output which usually depends on the input. Each program path defines an equivalence class on the input values which can execute it. Symbolic execution is a static analysis technique for grouping program inputs which produce the same symbolic output. The output of symbolic execution is a set of path conditions. A path condition is a set of constraints on the program inputs. The satisfaction of the constraints lead to the execution of the program path represented by the path condition. The constraint solving techniques return usually a single solution to each path constraint. The solution to the constraint is used to execute the program along the solved path.

We propose to combine symbolic execution with constraints solution space quantification in order to compute probabilities for program paths. We present an incremental and compositional approach to compute the probability of a path condition. Our approach handles both numeric constraints as well as heap constraint (i.e., constraint defined over data structures). In order to estimate the probability of numeric constraints, we extended existing interval branch-and-prune algorithms with a Monte Carlo technique which integrates both stratified sampling and importance sampling. This allows us to handle complex nonlinear constraints with controlled accuracy. We evaluated our approach on a set of benchmarks from robotic and medicine domains. The preliminary results show the efficiency of our approach compared to recent research approaches.

2 Motivating Example

Consider the code in Figure 1. Assume we want to estimate the probability of not reaching line 9, where an exception can arise. Assume the input variables x and y range over the integer domain $[1...100]$. The input domain is then $10^2 \times 10^2 = 10^4$ possible input values. The input domain can be much larger in practice.

Figure 1: Illustrative example and its symbolic execution tree



Consider the symbolic execution tree in Figure 1. Each node in the tree is a branch constraint. We count for each branch constraint the number of values form the input domain that satisfy that constraint (the counter is in bracket under each constraint). Assume that the inputs are uniformly distributed within the input domain. The probability of hitting line 9 is then: $6/10000 + 1454/10000 = 1460/10000 = 0.146$. The low probability of hitting line 9 means that a large number of random tests should be executed before detecting it.

3 Background

3.1 Symbolic Execution

Our approach bases mainly on the ability to symbolically execute the code under consideration. Algorithm 1 shows an abstract procedure of our symbolic execution. For a given program starting with statement s and an initial update $\mathcal{U}_0$, the call of $symExe(\mathcal{U}_0, s, true, \emptyset)$ will return the path conditions of all feasible paths of the program. Until a branching condition is found the procedure accumulates the state changes in form of update expressions (lines 5-7). In the case of a branching statement a new path condition is constructed for each branch outcome based of the current path condition Φ and the branch conditions ($cond(s)$ and $\neg cond(s)$). Only if a constructed path condition is satisfiable, the corresponding branch code is further proceeded (lines 8-11).

For example, the constraint solver can decide that the following constraint is satisfiable: $(x \geq 10) \wedge ((x < 5) \vee (x > 90))$. The constraint solver can found a solution, e.g., $x = 95$. The found solution can serve as an input value in a test case. When the program path is executable, i.e., the corresponding path condition is satisfiable, one can ask how many possible input values satisfy the path condition. Generally, the more inputs satisfy the path condition, the more probable the path can be executed. We discuss this intuition in the next section.

Algorithm 1: An abstract symbolic execution procedure – symExe

Data: $\mathcal{U}$: *Update*, s : *Statement*, Φ : *Formula*, PCs : *Set<Formula>*
Result: PCs : *Set<Formula>*

```
1  begin
2      if s = ∅ then
3          PCs ← PCs ∪ Φ
4      else
5          while ¬branch(s) do
6              U ← U ∘ update(s)
7              s ← next(s)
8          if SAT(Φ ∧ {U}cond(s)) then
9              symExe(U, first(s), Φ ∧ {U}cond(s), PCs)
10         if SAT(Φ ∧ {U}¬cond(s)) then
11             symExe(U, next(s), Φ ∧ {U}¬cond(s), PCs)
12         return PCs
```

3.2 Constraints Solution Space Computation

When solving a constrained problem, one is usually interested in finding one solution or assessing that there is no solution at all. However, knowing the number of solutions can give a new perspective on the constrained problem. In mathematics, a set of linear inequalities form a bounded geometric object. A solution to the set of inequalities is one point in the geometric object. The number of possible solutions is the volume of the geometric object. In the context of program analysis, each branch predicate is represented as a Boolean combination of numeric constraints. Knowing the volume of predicates allows to do fine-grained analysis of the possible program behavior.

3.3 Interval Branch-and-Prune Algorithms

Consider a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ of unknowns. A constrained problem is defined by a set $\mathcal{C} = \{c_1, \ldots, c_l\}$ of l constraints and a bounded domain $\mathcal{D}_\mathbf{x} = \mathcal{D}_{x_1} \times \ldots \times \mathcal{D}_{x_n}$ where $x_k \in \mathcal{D}_{x_k} := \{r \in \mathbb{R} | a_k \leq r \leq b_k\}, k = 1, \ldots, n$.

A constraint may contain nonlinear expressions which can be not differentiable. Each variable is bounded in a closed interval. The Cartesian product $\mathcal{D}_\mathbf{x}$ is called a *box*. The solution set of the constrained problem defined by the constrains set $\mathcal{C}$ is the set of tuples from $\mathbf{x}$ that satisfy all the constraints in $\mathcal{C}$.

Interval Branch-and-Prune Algorithms generate a set of $n-$dimensional boxes whose union define the solution set of a constrained problem. A Branch-and-Prune Algorithm alternates iterativ branch and prune tasks to generate boxes from the initial bounded domain. The algorithm stops when a fixed precision is reached. The pruning task eliminates *inconsistent* values and hence reduces the size of a box. The branch task splits the box into smaller boxes.

4 Compositional Path Condition Solution Space Computation

We consider the problem of efficiently computing the solution space of an individual path condition. A Path condition is a conjunction of a set of branching constraints. In real world applications, a path condition can be very large (i.e., include a large number of branching constraints). We propose to split a path condition into disjoint sets of branching constraints whose solution space can be determined independently from each other.

Each branching condition defines a relation between its variables. Each variable has a definition domain. A condition ranges over a given definition domain and specifies which values from the domain of its variables are compatible to the relation. More formally, we introduce the following definitions.

Definition 4.1 *(Branching Constraint). A branching constraint c is a triple $<\mathcal{V}_c, \mathcal{B}_c, \mathcal{R}_c>$, where $\mathcal{V}_c$ is a set of l variables $<v_1, v_2, \ldots, v_l>$, $\mathcal{B}_c$ is the Cartesian product $I_1 \times I_2, \ldots \times I_n$ with I_k the definition domain of variable v_k, and $\mathcal{R}_c$ the constraint relation defined as:*

$$\mathcal{R}_c \subseteq \{<i_1, i_2, \ldots i_l> | i_1 \in I_1, i_2 \in I_2, \ldots, i_l \in I_l\}$$

$\mathcal{R}_c$ is a subset of the Cartesian product $I_1 \times I_2, \ldots I_l$ with I_k the definition domain of variable v_k and i_k a possible value for variable v_k.

The definition of a path condition follows from the definition of a branching condition as follows:

Definition 4.2 *(Path Condition). A path condition Φ is a triple $<\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$ where $\mathcal{V}_\Phi$ is a set of n variables $<v_1, v_2, \ldots, v_n>$, $\mathcal{B}_\Phi$ (a box) the Cartesian product $I_1 \times I_2, \ldots \times I_n$ of the variables definition domains where each variable v_i ranges over the interval I_i, and $\mathcal{C}_\Phi$ is a finite set of branching constraints expressed as linear or nonlinear equations or inequalities on subsets of the variables $\mathcal{V}$. Consequently, a path condition can be defined as $\Phi = \bigwedge_{c_i \in \mathcal{C}_\Phi} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$.*

Now we move to the definition of the solution space of a path condition. We start with defining a solution to a branching constraint:

Lemma 4.1 *(Branching Constraint Solution). A solution of a branching constraint $c = <\mathcal{V}_c, \mathcal{B}_c, \mathcal{R}_c>$, is a tuple $s_c \in \mathcal{R}_c$ where $s_c \subseteq \mathcal{B}_c$.*

Our ultimate goal is to characterize the complete set of solutions:

Lemma 4.2 *(Branching Constraint Solution Space). The solution space of a branching constraint $c = <\mathcal{V}_c, \mathcal{B}_c, \mathcal{R}_c>$, is a set of tuples $\mathcal{S}_c \subseteq \mathcal{B}_c$ where:*

- $\forall\, s \in \mathcal{S}_c :\ s \in \mathcal{R}_c$ *(only solutions inside the set)*
- $\forall_{b \in \mathcal{B}_c} b \notin \mathcal{S}_c :\ b \notin \mathcal{R}_c$ *(no solutions outside the solution space)*

We propose to split a path condition into a set of disjoint branching constraints that have input variables in common. We define dependent constraints as follows:

Definition 4.3 *(Dependent Branching Constraints). Two branching constraints $c_i = <\mathcal{V}_{c_i}, \mathcal{B}_{c_i}, \mathcal{R}_{c_i}>$ and $c_k = <\mathcal{V}_{c_k}, \mathcal{B}_{c_i}, \mathcal{R}_{c_k}>$ are called dependent if: $\mathcal{V}_{c_i} \cap \mathcal{V}_{c_k} \neq \emptyset$.*

We introduce now a dependency relation among the constraints of a path condition:

Definition 4.4 *(Constraint Dependence Relation). The constraint dependence relation $\mathcal{DEP} : \mathcal{C} \times \mathcal{C} \rightarrow$ Boolean, where $\mathcal{C}$ a set of constraints, is recursively defined as follows:*

- $\forall c \in \mathcal{C} : \mathcal{DEP}(c,c) = true$
- $\forall c_i, c_j \in \mathcal{C}$, *if* $\mathcal{V}_{c_i} \cap \mathcal{V}_{c_k} \neq \emptyset$, *then* $\mathcal{DEP}(c_i,c_j) = true$
- $\forall c_i, c_j, c_k \in \mathcal{C}$, *if* $\mathcal{DEP}(c_i,c_j) = true \wedge \mathcal{DEP}(c_j,c_k) = true$, *then* $\mathcal{DEP}(c_i,c_k) = true$

Intuitively, two constraints are dependent if they share at least one input variable.

Lemma 4.3 *(Independent Branching Constraint Solution Space). The solution space of the conjunction of two independent branching constraints c_i and c_j is $S_{(c_i \wedge c_j)} = \mathcal{S}_{c_i} \cup \mathcal{S}_{c_j}$.*

The dependency relation allows us to split a path condition in a set of disjoint sets containing independent constraints.

Definition 4.5 *(Path Condition Split). We can split the formula of a path condition $\Phi = \bigwedge_{c_i \in \mathcal{C}} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$ into mutually exclusive and collectively exhaustive sets of constraints (or sub-formulas) based on the constraint dependence relation $\mathcal{DEP}$ as follows:*

- $\mathcal{C}_\Phi = \cup_{i \in \{1,\ldots,m\}} C_i^s$
- *For $i \neq j$, the sets C_i^s and C_j^s are disjoint: $C_i^s \wedge C_j^s = \emptyset$.*
- $\forall c_i, c_j \in C_k^s$: $\mathcal{DEP}(c_i,c_j) = true$
- $\forall c_i \in C_i^s$ *and* $\forall c_j \in C_j^s$: $\mathcal{DEP}(c_i,c_j) = false$

The splitted path condition Φ is defined then as: $\Phi_{split} = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_m$, where $\Phi_i = \bigwedge_{c_k \in C_i^s} c_k = <\mathcal{V}_{\Phi_i}, \mathcal{B}_{\Phi_i}, C_i^s>$.

Note that the dependency relation (see Def. 4.4) is by construction an equivalence relation over the set of constraints. Note also that for two independent constraints c_1 and c_2, the satisfaction of c_1 is independent from the satisfaction of c_2. Additionally, for two independent constraint sets $\mathcal{C}_1$ and $\mathcal{C}_2$, the satisfaction of the constraints in $\mathcal{C}_1$ is independent form the satisfaction of the constraints in $\mathcal{C}_2$.

Lemma 4.4 *(Path Condition Solution Space). The solution space of a path condition $\Phi = \bigwedge_{c_i \in \mathcal{C}} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$ is a set of tuples $\mathcal{S}_\Phi \subseteq \mathcal{B}_\Phi$ where:*

- $\forall_{\mathcal{S} \in \mathcal{S}_\Phi} \forall_{s \in \mathcal{S}} \forall c \in \mathcal{C} :\ s \in \mathcal{R}_c$ *(only solutions for all path constraints inside the set)*
- $\forall_{\mathcal{B} \subset \mathcal{B}_c} \mathcal{B} \not\subset \mathcal{S}_\Phi : \exists_{b \in \mathcal{B}}\ b \notin \mathcal{R}_c$ *(no solutions outside the solution space)*
- $\mathcal{S}_\Phi = \mathcal{S}_{\Phi_{split}} = \cup_{i \in \{1,\ldots,m\}} \mathcal{S}_{\Phi_i}$

Remarks. The composition of the solution space of a path condition allows us to split the quantification of the solution space of a large path condition into the analysis of smaller and simpler constraints. This allows to parallelize the quantification procedure of the solution space. It also allows us to reuse already quantified constraints (i.e., caching).

5 Solution Space of Constraints over Finite Floating Domains

We consider now the problem of counting the solution space of constraints defined over finite floating domains. Counting the number of solution of constraints defined over continuous domains involves computing an integral over the geometric object formed by the constraints. However, the constraints may contain nonlinear expression which are not differentiable. For this reason, we approximate the solution space of a conjunction of dependent constraints with a set of boxes that cover the exact solutions

of the constraints. The union of the boxes is an over-approximation of the solution space but never an under-approximation.

The boxes representing the solution space are extracted using constraint propagation techniques [14]. Constraint propagation techniques implement local reasoning on constraints to eliminate inconsistent values from the definition domains of the constraints variables. Such techniques prune and subdivide the definition domain of the constraints until a stopping criteria is satisfied. Note that the definition domain of the constraints as a Cartesian product of intervals is a set of boxes (See Def. 4.2 and Def. 4.1).

Definition 5.1 *(Consistency). A set $\mathcal{B} \subseteq \mathcal{B}_\Phi$ is consistent with a path condition $\Phi = \bigwedge_{c_i \in \mathcal{C}} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$ iff it contains at least one solution of Φ. Otherwise, it is called inconsistent.*

In order to eliminate input values that do not satisfy a constraint, a projection function is associated with each constraint:

Definition 5.2 *(Projection Function). For a path condition $\Phi = \bigwedge_{c_i \in \mathcal{C}} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$, a projection π_c of a constraint $c \in \mathcal{C}_\Phi$ with a solution space $\mathcal{S}_c$, is a mapping between the subsets of $\mathcal{B}_\Phi$ where $\forall \mathcal{B} \subset \mathcal{B}_\Phi$:*

- $\pi_c(\mathcal{B}) \subseteq \mathcal{B}$
- $\forall_{b \in \mathcal{B}}\ b \notin \pi_c(\mathcal{B}) : b \notin \mathcal{S}_c$

Usually the implementation of projection functions relies on interval analysis methods (e.g., the interval newton method). The set of projection functions associated with the constraints are then used to eliminate values from the definition domain that do not satisfy the constraints. The pruning of a box is done using constraint propagation. When a projection function eliminates a value of a variable, this information is propagated to the other constraints depending on that variable. This process terminates when the projection functions cannot further eliminate values (i.e., reduce the boxes).

Definition 5.3 *(Constraint Propagation). For a path condition $\Phi = \bigwedge_{c_i \in \mathcal{C}} c_i = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$, let $\pi_{\mathcal{C}_\Phi}$ be the set of projections for all the constraints $\mathcal{C}_\Phi$. Constraint propagation $\mathcal{CP}$ defines a mapping between the the subsets of $\mathcal{B}_\Phi$ where $\forall \mathcal{B} \subset \mathcal{B}_\Phi$:*

- $\mathcal{CP}(\mathcal{B}) \subseteq \mathcal{B}$ *(contractance)*
- $\forall_{b \in \mathcal{B}}\ b \notin \mathcal{CP}(\mathcal{B}) : \exists c \in \mathcal{C}_\Phi\ b \notin \mathcal{S}_c$ *(correctness)*
- $\forall_{\pi \in \pi_{\mathcal{C}_\Phi}}\ \pi(\mathcal{CP}(\mathcal{B})) = \mathcal{CP}(\mathcal{B})$ *(fixed point)*

The pruning level we can achieve using constraint propagation is dependent on the ability of the projection function to identify value combinations that do not satisfy the analyzed constraint [4]. However, projection functions do not miss any solution [4]. In order to further prune the result boxes, the boxes are subdivided and constraint propagation is applied to each sub-box. Such algorithms are called branch-and-prune algorithms. Such algorithms terminate when for example the box is judged too small to be considered for branching.

Constraint reasoning techniques do not loose any solution during the process of approximating the solution space of a set of constraints. Consequently, using such techniques, we get a safe enclosure for the solution space of a constraint.

Constraint reasoning techniques maintain two coverings for the solution space $\mathcal{S}_\Phi$ of path condition Φ. We assume that the variables $\mathcal{V}_\Phi$ are defined over $\mathbb{R}$, i.e., $\mathcal{B}_\Phi \subseteq \mathbb{R}^{|\mathcal{V}_\Phi|}$

Definition 5.4 *(Outer Box Cover). An outer box cover of $\mathcal{S}_\Phi$ is a set of disjoint boxes $\mathcal{S}_\Phi^\square = \{B_1, \ldots, B_n\}$ where:*

- $\forall_{i\in\{1,\ldots,n\}} B_i \subseteq \mathcal{B}_\Phi \wedge vol(B_i) > 0$
- $\forall_{i,j\in\{1,\ldots,n\}\wedge i\neq j}\ vol(B_i \cap B_j) = 0$
- $\mathcal{S}_\Phi \subseteq \bigcup_{i=1}^{n} B_i$

vol(B_i) is computed as the product of the intervals forming the box B_i.
Complementary to the concept of outer box cover, we define the concept of inner box cover.

Definition 5.5 *(Inner Box Cover). An inner box cover of $\mathcal{S}_\Phi$ is a set of disjoint boxes $\mathcal{S}_\Phi^{\blacksquare} = \{B_1,\ldots,B_n\}$ where:*

- $\forall_{i\in\{1,\ldots,n\}} B_i \subseteq \mathcal{B}_\Phi \wedge vol(B_i) > 0 \wedge B_i \subseteq \mathcal{S}_\Phi$
- $\forall_{i,j\in\{1,\ldots,n\}\wedge i\neq j}\ vol(B_i \cap B_j) = 0$
- $\bigcup_{i=1}^{n} B_i \subseteq \mathcal{S}_\Phi$

The solution space $\mathcal{S}_\Phi$ of the path condition Φ is approximated with a joint cover of $\mathcal{S}_\Phi^{\boxplus} = <\mathcal{S}_\Phi^{\square}, \mathcal{S}_\Phi^{\blacksquare}>$ of the outer and inner cover box where $\mathcal{S}_\Phi^{\blacksquare} \subseteq \mathcal{S}_\Phi^{\square}$.

Constraints over Integer Domains and Mixed Domains: The presented approach works also for integer domains and mixed integer constrains (i.e., constraints which contain both integer and floating variables). As suggested in [5], we can handle integer variables as floating variables when each domain modification is followed by rounding the computed bounds to the nearest integer inside the interval domain. The resulting integer value is represented as a point interval to be conform to the definitions above of the solution space enclosure.

Disjunctive Domains: Consider the case when a variable x is defined over the union of intervals $[-100,2] \cup [7,100] \cup [200,500]$. We can define the variable x over the interval $[-100,500]$ and add the constraint $min(x-2, min(max(7-x, x-100), 500-x)) \leq 0$. Note that such operations are not differentiable. However, constraint reasoning techniques need only that the operations can be evaluated over the intervals.

6 Solution Space of Constraints Over Data Structures

The computation of the solution space for constraints over data structures deserves special interest. Such constraints are called heap constraints. The solution space in the case of data structure variables is discrete. Quantifying the solution space means counting the model formed by the constraints. As before, we restrict ourselves to finite input domains. Consequently, the number of possible heap nodes in the input domain is finite.

We propose to use Korat [7] to count the input data structure that satisfy a constraint over data structures within pre-defined bounds. Korat is a framework for the constraint-based generation of structurally complex inputs for Java programs. Korat provides also efficient counting of input data structures. Korat generates the inputs by solving constraints written as a boolean method called *repOk*. The body of such a method can contain any arbitrary complex predicate. The scope of the input domain is specified using specific Korat methods. Scope methods are used to specify bounds on the size of the input data structures and bounds on the definition domain of the primitive fields of the data structure.

We encode the constraints we obtain from symbolic execution as a predicate in the *repOk* methods. Korat counts then the data structures that satisfy the constraint for a given scope.

Example: Consider the Java code in Listing 1 for swapping a node in a linked list. The field `element` represent the integer value of the node. The field `next` represent the next node in the list. The method `swapNode` updates the input list which is referenced by the parameter `this`. The update is done through a nonlinear condition on the nodes `n` and `next`.

```
1  class Node {
2    int element;
3    Node next;
4
5    Node swapNode () {
6      if(next!=null) {
7        if(element > next.element) {
8          // location to analyze
9          Node n = next;
10         next = n.next;
11         n.next = this;
12         return n;
13        }
14     }
15     return this;
16   }
17 }
```

Listing 1: Example swapping a node in a linked list

We illustrate now the use of Korat to count the data structure models. First of all we scope our domain, and assume that the nodes can take the values 1 or 2. Additionally, we bound the size of the linked list to 2 nodes. These bounds are passed to Korat via its scope methods.

The path condition to reach the body of the second branching condition in the `swapNode` (i.e., the location at line 8) is:

`node!=null` $\wedge$ `node.next!= null` $\wedge$ `node.next!=node` $\wedge$ `node.element>node.next.element`

We pass the path condition to the `repOk` method of Korat. The total number of of valid input data structures that satisfy the path condition under the specified scope is 17. This means, there is 17 possible inputs to reach the location at line 8 of the code in Listing 1.

Remark: Constraints over numerical domains that contain transitive dependencies on the data structure encoded by the heap constraints are also counted by Korat.

7 Probability of Satisfying a Path Condition

The theory of probability is a classical model to deal with uncertainty. A probabilistic model is defined by a set of random variables and a set of events. A random variable is a function from the sample space to the real numbers. An event is an assignment of values to all the variables of the model.

We want to compute the probability of satisfying a path condition. In our case here, the model is the path condition and the random variables are the variables of the path condition. An event is an assignment of values to the variables such that the path condition is satisfied.

In order to specify a probabilistic model, a full joint probability distribution should be explicitly or implicitly used. This distribution assigns a probability measure to each possible event. Such distributions can be provided by on Operational Profile (OP). As defined by Musa [15] "an Operational Profile is a quantitative characterization of how a (software) system will be used".

OP Example: Consider a method with a single input variable x defined over a floating domain. A possible OP can be of the form $OP = \{(x \in [1,10], 0.3), (x \in [20,30], 0.7)\}$. This means that the probability that the variable x takes values from the interval $[1,10]$ is 0.3 and that it takes values from $[20,30]$ is 0.7.

More formally, an OP can be defined as $OP = \{(\mathcal{C}_i, p_i) | i \in \{1,...,L\}, \sum_{i=1}^{L} p_i = 1\}$: it is a set of pairs $(\mathcal{C}_i, p_i)$ where $\mathcal{C}_i$ represents constraints over the definition domain to describe a possible operational scenario, and p_i is the probability that an operational input belongs to $\mathcal{C}_i$.

7.1 Probability of a Path Condition over Data Structures

For heap path conditions, we use model counting as described in Section 6. Let $\#(c_i)$ denotes the function which counts the number of elements from a definition domain $\mathcal{D}$, which satisfy c_i. The probability of c_i is then: $\mathbb{P}(c_i) = \#(c_i)/\#\mathcal{D}$.

Consider we have the following operational profile, $OP = \{(\mathcal{C}_i, p_i) | i \in \{1,...,L\}, \sum_{i=1}^{L} p_i = 1\}$. For a path condition Φ, it follows from the law of total probability: $\mathbb{P}(\Phi|OP) = \sum_{i=1}^{L} \mathbb{P}(\Phi|\mathcal{C}_i).p_i$. Furthermore, it follows from the definition of conditional probability: $\mathbb{P}(\Phi|\mathcal{C}_i) = \mathbb{P}(\Phi \wedge \mathcal{C}_i)/\mathbb{P}(\mathcal{C}_i)$. Consequently, we obtain: $\mathbb{P}(\Phi|\mathcal{C}_i) = \sum_{i=1}^{L} \frac{\#(\Phi \wedge \mathcal{C}_i)}{\#(\mathcal{C}_i)}.p_i$.

7.2 Probability of a Path Condition over Numeric Domains

Definition 7.1 *(Probability of a Path Condition) The probability of a path condition $\Phi = <\mathcal{V}_\Phi, \mathcal{B}_\Phi, \mathcal{C}_\Phi>$ given the indicator function $\mathbb{1}_{\mathcal{S}_\Phi}(x) : \mathbb{R}^{|\mathcal{V}_\Phi|} \to \{1,0\}$ defined as follows:*

$$\mathbb{1}_{\mathcal{S}_\Phi}(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{x} \in \mathcal{S}_\Phi \\ 0, & \text{if } \mathbf{x} \notin \mathcal{S}_\Phi \end{cases}$$

is defined as:

$$\mathbb{P}(\Phi) = \int_{\mathcal{B}_\Phi} \mathbb{1}_{\mathcal{S}_\Phi}(\mathbf{x}).f_{\mathcal{V}_\Phi}(\mathbf{x})d\mathbf{x}$$

where $f_{\mathcal{V}_\Phi}$ is a full joint probability density function (p.d.f) over the path constraint variables $\mathcal{V}_\Phi$, $\mathcal{S}_\Phi$ the solution space of the path condition and $\mathcal{B}_\Phi$ the definition domain of the path condition.

Generally, the multidimensional integral in Def. 7.1 may have no closed form solution since the constraints of a path condition may define a complex nonlinear integration boundary. Our approach approximates the solution space of a path condition with a joint cover $\mathcal{S}_\Phi^{\boxplus} = <\mathcal{S}_\Phi^{\square}, \mathcal{S}_\Phi^{\blacksquare}>$.

Monte Carlo methods provide an approach to approximate the value of multidimensional integrals by randomly sampling N points in the multidimensional definition space and averaging the integral values at the samples.

Definition 7.2 *(Monte Carlo Integration) Let $\mathcal{S}_\Phi \subseteq \mathbb{R}^{|\mathcal{V}_\Phi|}$, and B a $|\mathcal{V}_\Phi|-$dimensional box. If we sample uniformly N random values $\{x_1, \ldots, x_n\}$ inside B, then by the law of large numbers it follows:*

$$\int_B \mathbb{1}_{\mathcal{S}_\Phi}(\mathbf{x}).f_{\mathcal{V}_\Phi}(x)dx \approxeq \widehat{I}_{\mathcal{S}_\Phi}(B, f_{\mathcal{V}_\Phi}) = \frac{\sum_{i=1}^{N} \mathbb{1}_{\mathcal{S}_\Phi}(x_i).f_{\mathcal{V}_\Phi}(x_i)}{N}.vol(B)$$

where $vol(B)$ the volume of the box B.

By the central limit theorem, one can estimate the uncertainty in the approximation of the Monte Carlo integration.

Definition 7.3 *(Standard Deviation of the Estimate) The standard deviation of the approximation of the integral $\widehat{I}_{\mathcal{S}_\Phi}(B, f_{\mathcal{V}_\Phi})$ follows from the central limit theorem as follows:*

$$\sigma(\widehat{I}_{\mathcal{S}_\Phi}(B, f_{\mathcal{V}_\Phi})) = \frac{vol(B)}{N}\sqrt{\sum_{i=1}^{N}(\mathbb{1}_{\mathcal{S}_\Phi}(x_i).f_{\mathcal{V}_\Phi}(x_i))^2 - \frac{(\sum_{i=1}^{N}(\mathbb{1}_{\mathcal{S}_\Phi}(x_i).f_{\mathcal{V}_\Phi}(x_i))^2}{N}}$$

The standard deviation describes a statistical estimate of the error on the integral approximation.

Definition 7.4 *(Approximate Probability of a Path Condition) Given a joint box cover $\mathcal{S}_\Phi^{\boxplus} = <\mathcal{S}_\Phi^{\square}, \mathcal{S}_\Phi^{\blacksquare}>$ of the solution space of a path condition Φ, an approximation for the probability of satisfying Φ is given by:*

$$[\mathbb{P}(\Phi)] = \sum_{B_i \in \mathcal{S}_\Phi^{\square}} \left[\frac{\sum_{i=1}^{N} \mathbb{1}_{\mathcal{S}_\Phi}(x_i).f_{\mathcal{V}_\Phi}(x_i)}{N}.vol(B_i) \right]$$

Monte Carlo Integration may suffer from a slow convergence rate especially when the approximated integral gets close to zero. One may need a large number of random samples N to approximate the probability to some given confidence. Stratified sampling and importance sampling are well-know techniques to reduce the variance of Monte Carlo integration methods. We integrate these techniques in our approximation as follows:

Definition 7.5 *(Approximate Probability of a Path Condition) Given a joint box cover $\mathcal{S}_\Phi^{\boxplus} = <\mathcal{S}_\Phi^{\square}, \mathcal{S}_\Phi^{\blacksquare}>$ of the solution space of a path condition Φ, an approximation for the probability of satisfying Φ is given by:*

$$[\mathbb{P}(\Phi)] = \sum_{B_i \in \mathcal{S}_\Phi^{\square}} \left[\frac{\sum_{i=1}^{N} \mathbb{1}_{\mathcal{S}_\Phi}(x_i).f_{\mathcal{V}_\Phi}(x_i)}{N}.\mathbb{P}(B_i) \right] = \sum_{B_i \in \mathcal{S}_\Phi^{\square}} \left[\widehat{p}_i.\mathbb{P}(B_i) \right]$$

The scheme $\sum_{B_i \in \mathcal{S}_\Phi^{\square}} \left[\widehat{p}_i.\mathbb{P}(B_i) \right]$ integrates both stratified sampling and importance sampling. Each box B_i can be written as the Cartesian product of intervals: $B_i : [a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_i, b_i]$. $\mathbb{P}(B_i)$ is defined then as: $\mathbb{P}(B_i) = \mathbb{P}_{x_1}([a_1, b_1]).\mathbb{P}_{x_2}([a_2, b_2]) \ldots \mathbb{P}_{x_i}([a_i, b_i])$ with $\mathbb{P}_{x_i}([a_i, b_i]) = \int_{a_i}^{b_i} f_i(x_i)dx_i$ and f_i the probability distribution function over the variable x_i. Such a distribution can be specified in an OP.

8 Looping Constructs: Incremental Probabilistic Analysis

Usually a bound on the exploration depth is set when executing a program symbolically. Instead of setting a static bound, we introduce a probabilistic bound P_{depth}. Given an OP, the user may be interested in only exploring program paths which have a probability of occurrence higher than P_{depth}. Algorithm 2 sketches our extension to symbolic execution to incrementally compute the path condition probabilities. Algorithm 2 returns a list of path conditions together with their computed probabilities. All the returned path conditions have a probability higher than P_{depth}. Algorithm 3 computes the probability of a conjunction of constraints. Algorithm 3 splits the conjunction (line 1) as defined in Def. 4.5. It then computes the probability of Φ as described in Section 7.

At each branch condition, algorithm 2 decides whether it should further explore the path or abandon it (lines 9-13).

Algorithm 2: An abstract incremental probabilistic symbolic execution procedure – IncProbSymExe

Data: $\mathcal{U}$: *Update*, s : *Statement*, Φ : *Formula*, *PCs* : *Set<Formula>*
$OP = \{(D_i, p_i) | i \in \{1,...,L\}, \sum_{i=1}^{L} p_i = 1\}$, P_{depth}

```
1  begin
2      if s = ∅ then
3          PCs ← PCs ∪ Φ
4      else
5          while ¬branch(s) do
6              U ← U ∘ update(s)
7              s ← next(s)
8          P(Φ ∧ {U}cond(s)) ← computeProbs(OP, Φ, {U}cond(s))
9          if P(Φ ∧ {U}cond(s)) ≥ P_depth then
10             symExe(U, first(s), Φ ∧ {U}cond(s), PCs)
11         P(Φ ∧ {U}¬cond(s)) ← computeProbs(OP, Φ, {U}¬cond(s))
12         if P(Φ ∧ {U}¬cond(s)) ≥ P_depth then
13             symExe(U, first(s), Φ ∧ {U}¬cond(s), PCs)
14         return <PCs, Probabilities>
```

Algorithm 3: Compute formula probability and search depth – computeProbs

Data: $OP = \{(D_i, p_i) | i \in \{1,...,L\}, \sum_{i=1}^{L} p_i = 1\}$,
Φ : *Formula*, c : *Formula*

```
1  begin
2      <Φ_dep, Φ_notdep> = split(Φ, c)
3      P(Φ) ← P(Φ(notdep)) //Previously computed and in cache
4      P(Φ) ← P(Φ) + P(Φ_dep | OP)
5      return P(Φ)
```

9 Implementation and Experiments

We describe in this Section the implementation and the evaluation of the ideas we presented so far.

Implementation: Our prototype implementation uses the symbolic execution engine of the KEY system [11]. In order to split a path condition into disjoint sets of dependent constraints (see Def. 4.5), we model the constraints of each path condition as an undirected graph. The nodes of the graph are the constraints and the edges encode a dependency between the constraints: when constraints share the same input variable, an edge is added between the corresponding nodes. The computation of the connected components of the graph delivers us the split. In order to approximate the solution space of the constraints in boxes, we base our implementation on a cheap interval branch-and-prune constraint propagation framework, RealPaver [10]. The original RealPaver defines a user defined stopping criteria for the branch-and-prune algorithm by specifying (i) a maximal time budget per query, or (ii) the number of boxes reported per query, or (iii) lower bound on the size of box eligible for branching. We extended

RealPaver by introducing a new stopping criteria which is more suitable to our probabilistic setting. Our goal when approximating the solution space of a path condition is the accurate computation of the probability of occurrence of that path condition. The stopping criteria we introduced to the branch-and-prune algorithm imposes a used defined accuracy to the probability enclosure computed with Monte Carlo integration over the outer box cover. This allows us to control the branching part toward the boxes with the highest uncertainty in their computed probability. Consequently, we can efficiently reduce the uncertainty on the computed probability. In the case that the user required accuracy is too sharp and the required accuracy cannot be reached (because of the accumulation of rounding errors), the branch-and-prune algorithm stops when the lower bound on the size of the boxes reached (there are no more eligible boxes).

All the following experiments are executed on an Mac Pro 2.66 Ghz with 8Gb of memory running OSX 10.9.

Experiments: Constraints over finite floating domains The following experiment evaluates how our approach compares with recently developed techniques, `VolComp` [3, 16] and `qCoral` [2, 6]. `VolComp` and `qCoral` are both recent techniques to approximate the probability of constraints. We use the built-in method `NProbability` (with the default parametrization) of the mathematical tool Mathematica [1] as a baseline for comparison. `NProbability` computes numerical integrals over predicates and probabilities and terminates when default accuracy requirements are met and notifies when the accuracy requirements are not met.

`VolComp` bounds the solution with an interval. `qCoral` as well as our approach report the approximated solution and a standard deviation of the approximation. Our approach was configured as follows: (i) for the Monte Carlo integration, we use $N = 1000$ random samples, (ii) we set the lower bound on the size of the boxes eligible for branching to 10^{-5} and (iii) we set the required accuracy to 0.005 (the stopping criteria of our approach). We used the same configuration for `qCoral`, except the accuracy stopping criteria, since `qCoral` do not provide such a feature. Both our approach and `qCoral` implement randomized algorithms. We report averaged estimate and standard deviation over 20 runs.

To compare the three approaches, we selected benchmarks from the publicly available `VolComp` benchmarks [3]. The comparison subjects are: (i) `ARTRIAL`: the Framingham artial fibrillation risk calculator, (ii) `CORONARY`: the Framingham hypertension risk calculator, (iii) `PACK`: a model of a robot packing objects with varying weights and (vi) `VOL`: controller for filling a tank with fluid at certain rates. The path conditions for these programs are produced using `VolComp`. Table 1 summarizes the comparison between our approach and `VolComp` and `qCoral`. The first column of Table 1 state the program event whose probability is computed. The second column *#PCs* states the number of path conditions that reach the event.

In summary, our approach was always faster than `qCoral`, `VolComp` and `NProbability`. Note that the performance of `NProbability` depends usually on advanced settings. The tuning of such settings requires a deep understanding of the mathematical properties of the function to integrate. Such an understanding may not be derived from the code during the analysis. The efficiency of our approach compared to `qCoral` can be explained by the fact that we apply Monte Carlo Integration only on the outer box cover of the approximated solution space. However, `qCoral` samples randomly over the whole approximated solution space. Our approach required more than 30 minutes to compute the `Vol` event *count* ≥ 20. This is caused by the accumulation of rounding errors. Rounding errors accumulation magnifies when the quantified probability is close to 1. The rounding errors increase the uncertainty an the estimate. More uncertainty means more sampling.

We notice that the estimates computed by our approach as well as the estimates computed by `qCoral` fall within the bounds extracted by `VolComp`. The estimates delivered by our approach were closer to the

exact solutions delivered by NProbability than the estimates produced by qCoral and VolComp. The precision of our approach is due to the integration of importance sampling and stratified sampling which reduce the uncertainty of the estimate. In addition, we control the branching step of the interval branch-and-prune algorithm toward branching the boxes with the highest uncertainty. This should decrease the overall uncertainty.

We observe that our approach as well as qCoral were equal slow for the benchmark PACK. The reason for that is that RealPaver generated only an outer box cover for the solution space. This means we sampled randomly over the whole solution space. This reduces the impact of our sampling strategy.

Table 1: Comparison of NProbability, VolComp, qCoral and our approach

Event	#*PCs*	NProbability		VolComp		qCoral			our approach		
		solution	time(s)	bound	time(s)	avg.estimate	avg.σ	avg. time(s)	avg.estimate	avg.σ	avg. time(s)
ARTRIAL											
score $\geq$ 10	442	0.1343	476	[0.1343, 0.1343]	341	0.1343	0	183	0.1343	0	117
err $\leq$ 5	2260	0.9467	3879	[0.9387, 9573]	1390.24	0.9389	2.80e-04	685	0.9464	1.26e-04	344
CORONARY											
err $\geq$ 5	320	0.0006	3.44	[0, 0.0172]	23	0.00047	3.29e-06	7.205	0.00051	1.87e-06	3.301
VOL											
count $\geq$ 20	24	1.0005	1538.9	[0, 1]	1842	1.0003	2.72e-05	1864	1.0003	7.82e-03	1856
PACK											
totalWeight $\geq$ 4	1132	0.95051	54	[0.95051, 1]	24	0.95038	1.00e-06	126.2	0.95046	1.12e-06	123.89

Experiments: Constraints over data structures The following experiments are conducted on a Binary Search Tree implementation adapted from [9]. The implementation can be found in the Appendix. We used our approach to compute the probability of reaching different branches in the code implementing the two methods add(n) and delete(n). Both methods take integer values as input. We bounded the scope input domain to data structures with 3 nodes with increasing data value ranges $[1 \dots 10]$, $[1 \dots 50]$ and $[1 \dots 100]$. We compute for different branches in the code all path conditions that reach the branch as well as their probabilities. The probability of the branch is approximated by the sum of the probabilities of the path conditions reaching it. The results are presented in Table 2. The probabilities are rounded for presentation purposes. The Branch Location column indicates the location in the code, # PCs refers to how many path conditions reach the branch and the three following columns show the computed probability to reach the branch. The parameter values are chosen uniform randomly from the intervals.

Table 2: Probability for covering branches in a Binary Search Tree

Method	Branch Location	# PCs	[1 ... 10]	[1 ... 50]	[1 ... 100]
add	1	7	6.1218×10^{-2}	6.4499×10^{-2}	6.8939×10^{-2}
	2	10	3.3261×10^{-1}	3.5542×10^{-1}	3.7677×10^{-1}
	3	7	6.1218×10^{-2}	6.4499×10^{-2}	6.8939×10^{-2}
	4	10	3.3261×10^{-1}	3.5542×10^{-1}	3.7677×10^{-1}
delete	5	7	4.3999×10^{-1}	4.7931×10^{-1}	4.9165×10^{-1}
	6	14	3.5834×10^{-1}	3.7464×10^{-1}	3.8802×10^{-1}
	7	14	5.3759×10^{-2}	5.7464×10^{-2}	6.1537×10^{-2}
	8	1	0.9399×10^{-6}	2.4310×10^{-7}	1.3728×10^{-9}

First observation to make is that there is no correlation between the number of path conditions reaching a branch and the probability of covering that branch. For example, the branch at location 7 of method delete is reached by 14 PCs and the probability to cover it is smaller than the branch at location 7 which is reached by only 7 PCs.

Considering the implementation code of the method add, the branches at locations 1 and 3 as well as the branches at location 2 and 4 are symmetric around the check whether the value to add is less or greater than current root value. This code aspect is captured by our probabilistic approach.

Next observation we can make is that for some branches the probability to reach them increases when the range of value increases. For example, adding values to the binary tree is easier when the range of values to select from is larger: it is less likely to select and add a value that is already in the binary tree.

The branch at location 8 in the method `delete` is the least likely to be reached. This event becomes more rare when the range of allowed input values increases. Based on the implementation code, this branch corresponds to the case when we try to delete the root node when the tree is empty. This is an unlikely behavior since it simulates deleting an element from an empty tree.

Scalability Remarks: Korat enumerates each possible data structure including all input values. Such an enumeration can be very expensive especially when the range of possible input values increases. For counting data structure models with values in $[1 \dots 10]$, Korat took less than 2 seconds on average. However, for values in the range $[1 \dots 50]$ Korat took 17 minutes and more than 2 hours on average for values in the range $[1 \dots 100]$.

10 Related Work

The presented work is related to many areas including statistical model checking [12], analysis of probabilistic programs [13], and integration methods over polyhedras [8].

We compute the probability of a path condition or more generally a set of path conditions that lead to a program behavior of interest. The techniques for the probability computation of the path conditions differ in the approach used to approximate the solution space, the distribution type of the input variables and the linearity of the constraints.

Geldenhuys et al. [9] present an approach that considers only uniform distributed input variables and linear integer arithmetic constraints. They used LattE Machiato [8] to count the solution space of the path conditions. One main difference between this work and ours is that we support complex nonlinear constraints and we use constraint propagation techniques to approximate the solution space. In addition our approach is not restricted to uniform distribution.

Sankaranarayanan et al. [16] recently proposed a technique to remove the restriction of uniform distribution by developing an algorithm for the under and over-approximation of probabilities. They use Linear Programming solvers to compute the over-approximations and heuristics to compute the under-approximation. However, their approach is limited to linear constraints. More recently, Borges et al. [6] proposed an approach for handling nonlinear constraints based on interval constraint propagation techniques and Monte Carlo integration. One main technical difference between this approach and our work is that our approach is incremental and computes probabilities at each branching constraint which allows for better scalability of symbolic execution. The approach of Borges et al. computes the probabilities after symbolic execution finishes. In addition, our work extends interval constraint propagation by allowing to control the efficiency of the solution space approximation. The approximation procedure is controlled based on a user-defined accuracy parameter on the computed probability of a target program behavior. Furthermore, our work makes use of the joint box cover structure computed by the interval constraint propagation techniques and applies Monte Carlo integration only on the outer cover. Borges et al. apply Monte Carlo integration on the whole approximated solution space. Consequently, their approach as shown in our experiments may require more samples to compute the probabilities with a given accuracy. Moreover, our work supports constraints over data structure which is not supported by Borges et al.

11 Conclusion

We have developed an incremental and compositional approach for the approximation of the solution space of complex nonlinear constraints. We also presented an approach for counting the solution space of constraints over data structures. This allows us to extend symbolic execution to perform a probabilistic analysis - the computation of path condition probabilities. We also allowed adding uncertainty about the input values of the analyzed program and take such uncertainty into account when computing the probability of a path condition. Our initial experiments are promising, however our approach has same scalability issues especially when counting solutions for constraints over data structures. We plan to explore optimization schemes to its performance. We also plan to open source our tool.

References

[1] *Mathematica, NProbability*. `http://reference.wolfram.com/language/ref/NProbability.html`. [Online; accessed 10-December-2014].

[2] *qCoral*. `http://pan.cin.ufpe.br/coral/QCORAL.html`. [Online; accessed 10-December-2014].

[3] *VolComp*. `http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis/`. [Online; accessed 10-December-2014].

[4] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers & Jean-François Puget (1999): *Revising Hull and Box Consistency*. In: *Proceedings of the 1999 International Conference on Logic Programming*, Massachusetts Institute of Technology, Cambridge, MA, USA, pp. 230–244.

[5] Frédéric Benhamou & William J. Older (1997): *Applying interval arithmetic to real, integer and Boolean constraints*. *JOURNAL OF LOGIC PROGRAMMING* 32(1), pp. 1–24.

[6] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu & Willem Visser (2014): *Compositional Solution Space Quantification for Probabilistic Software Analysis*. *SIGPLAN Not.* 49(6), pp. 123–132.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid & Darko Marinov (2002): *Korat: Automated Testing Based on Java Predicates*. *SIGSOFT Softw. Eng. Notes* 27(4), pp. 123–133.

[8] J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto & J. Wu (2012): *Software for Exact Integration of Polynomials over Polyhedra*. *ACM Commun. Comput. Algebra* 45(3/4), pp. 169–172.

[9] Jaco Geldenhuys, Matthew B. Dwyer & Willem Visser (2012): *Probabilistic Symbolic Execution*. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, ACM, pp. 166–176.

[10] Laurent Granvilliers & Frdric Benhamou (2006): *RealPaver: An Interval Solver using Constraint Satisfaction Techniques*. *ACM TRANS. ON MATHEMATICAL SOFTWARE* 32, pp. 138–156.

[11] Martin Hentschel, Reiner Hähnle & Richard Bubel (2014): *Visualizing Unbounded Symbolic Execution*. In Martina Seidl & Nikolai Tillmann, editors: *Proceedings of Testing and Proofs (TAP) 2014*, LNCS, Springer, pp. 82–98.

[12] Andrew Hinton, Marta Kwiatkowska, Gethin Norman & David Parker (2006): *PRISM: A Tool for Automatic Verification of Probabilistic Systems*. In: *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, Springer-Verlag, Berlin, Heidelberg, pp. 441–444.

[13] David Monniaux (2001): *An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs*. *SIGPLAN Not.* 36(3), pp. 93–101.

[14] Ramon E. Moore (1979): *Methods and Applications of Interval Analysis*.

[15] J.D. Musa (1993): *Operational profiles in software-reliability engineering*. *Software, IEEE* 10(2), pp. 14–32.

[16] Sriram Sankaranarayanan, Aleksandar Chakarov & Sumit Gulwani (2013): *Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths*. *SIGPLAN Not.* 48(6), pp. 447–458.

APPENDIX This appendix presents the code for the Binary Search Tree example used in this paper.

A Implementation Code of the Method add

```
1
2  public void add(int x) {
3       Node current = root;
4
5       if (root == null) {
6          root = new Node(x);
7          return;
8       }
9
10      while (current.value != x) {
11         if (current.value > x) {
12            if (current.left == null) {
13               //Location 1
14               current.left = new Node(x);
15            } else {
16               //Location 2
17               current = current.left;
18            }
19         } else {
20            if (current.right == null) {
21               //Location 3
22               current.right = new Node(x);
23            } else {
24               //Location 4
25               current = current.right;
26            }
27         }
28      }
29    }
```

B Implementation Code of the Method delete

```
1
2    public boolean delete(int x) {
3  //   Algorithm note: There are four cases to consider:
4  //   1. The node is a leaf.
5  //   2. The node has no left child.
6  //   3. The node has no right child.
7  //   4. The node has two children.
8
9       //initialize parent and current to root
10      Node current = root;
11      Node parent = root;
```

```
12
13       boolean isLeftChild = true;
14
15       if (current == null)
16          return false;
17
18       //while loop to search for node to delete
19       while(current.value != x) {
20          //assign parent to current
21          parent = current;
22          if(current.value > x) {
23             //Location 5
24                isLeftChild = true; //current is a left child
25                current = current.left; //make current's left child the current node
26           }
27          else {
28             //Location 6
29                isLeftChild = false; //current is a right child
30                current = current.right; //make current's right child the current node
31           }
32          if(current == null) { //data can't be found, break from loop
33             //Location 7
34             return false;
35          }
36       }
37 //    test for a leaf
38       if(current.left == null && current.right == null)
39       {
40          if(current == root) {//tree has a single node, make root null
41             //Location 8
42             root = null;
43          }
44          else if(isLeftChild) { //current is a left child so make its parent's left
                null
45             parent.left = null;
46          }
47          else {
48             parent.right = null; //current is a right child so make its parent's right
                   null
49          }
50       }
51 //    test for no right child
52       else if(current.right == null)
53          if(current == root) { //current is root so make root point to current's left
54             root = current.left; //old root gets deleted by garbage collector
55          }
56          else if(isLeftChild) { //current is a left child so make its parent's left
                point to it's left child
57             parent.left = current.left;
58          }
59          else { //current is a right child so make its parent's right point to it's
                left child
```

```
60            parent.right = current.left;
61          }
62  //    test for no left child
63        else if(current.left == null)
64          if(current == root) { //current is root so make root point to current's right
65            root = current.right; //old root gets deleted by garbage collector
66          }
67          else if(isLeftChild) { //current is a left child so make its parent's left
                point to it's right child
68            parent.left = current.right;
69          }
70          else { //current is a right child so make its parent's right point to it's
                right child
71            parent.right = current.right;
72          }
73  //    there are two children:
74  //    retrieve and delete the inorder successor
75        else {
76
77          Node successor = getSuccessor(current); //get successor
78
79          if(current == root) {
80            root = successor;
81          }
82          else if(isLeftChild) {
83            parent.left= successor; //set node to delete to successor
84          }
85           else {
86            parent.right = successor;
87           }
88  //       attach current's left to successor's left since successor has no left child
89          successor.left = current.left;
90         }
91        return true;
92     }
```