Compositional Solution Space Quantification for Probabilistic Software Analysis

Mateus Borges  
Federal University of Pernambuco, Brazil

Antonio Filieri  
University of Stuttgart, Germany

Marcelo d’Amorim  
Federal University of Pernambuco, Brazil

Corina S. Păsăreanu  
CMU SV/NASA Ames Research Center, United States

Willem Visser  
University of Stellenbosch, South Africa

Abstract
Probabilistic software analysis aims at quantifying how likely a target event is to occur during program execution. Current approaches rely on symbolic execution to identify the conditions to reach the target event and try to quantify the fraction of the input domain satisfying these conditions. Precise quantification is usually limited to linear constraints, while only approximate solutions can be provided in general through statistical approaches. However, statistical approaches may fail to converge to an acceptable accuracy within a reasonable time.

We present a compositional statistical approach for the efficient quantification of solution spaces for arbitrarily complex constraints over bounded floating-point domains. The approach leverages interval constraint propagation to improve the accuracy of the estimation by focusing the sampling on the regions of the input domain containing the sought solutions. Preliminary experiments show significant improvement on previous approaches both in results accuracy and analysis time.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification

Keywords Symbolic Execution, Monte Carlo Sampling, Probabilistic Analysis, Testing

1. Introduction
The pervasiveness of software, the heterogeneity of its users, and the growing complexity of interactions with third-party components are introducing a new degree of uncertainty about the execution conditions of a program, raising the quest for verification techniques able to deal with and to quantify such uncertainty, both in the problem and in the verification results. Probabilistic software analysis aims at quantifying the probability of a target event to occur, given a probabilistic characterization of the behavior of a program or of its execution environment. Examples of target events include an uncaught exception, the invocation of a certain method, or the access to confidential information.

In the past probabilistic software analysis has been mostly performed at model level [20], limiting its applicability to early software design stages or requiring explicit abstraction from the code. Some recent techniques have brought it at the code-level [11, 12, 30]. These techniques use a symbolic execution of the program to collect the symbolic constraints on the input that lead to the occurrence of the target events. The constraints are then analyzed to quantify how likely is an input, distributed according to certain usage profiles [11], to satisfy any of them.

The quantification of the solution space for a set of constraints is one of the main obstacles to the applicability of probabilistic software analysis in practice. In previous studies [11, 12], model counting techniques have been applied to count the number of points of a bounded integer domain that satisfy given linear constraints. These counts are then coupled with a probabilistic input usage profile, which establishes a probability distribution over the input domain, to assess the probability for the target event to occur in that usage profile. Sankaranarayanan et al. [30] proposed a different approach to quantify the solution space of linear constraints over bounded floating-point domains. They propose an iterative algorithm to compute tight over-approximating bounds of the actual solution space suitable for efficient volume computation. All these approaches are thus quite limited as they can only handle linear constraints. In this paper we tackle the problem for the general case of complex (non-linear, containing transcendental functions, etc.) mathematical constraints over floating-point domains with application to probabilistic software analysis.

When considering floating-point domains, the quantification of the solution space for a constraint usually involves the computation of an integral. Symbolic and numerical integration are usually impracticable in the case of complex mathematical constraints and multi-dimensional input domains due to the high computational time and memory demands of such methods [17]. Furthermore, despite the fact they could be able to provide exact results, symbolic methods cannot deal with problems whose solution cannot be expressed in analytical form, which is the case for many integration problems [15]. Statistical methods overcome the limitations in terms of feasibility and memory demand of both symbolic and numerical methods, allowing to deal with arbitrarily complex constraints. However, statistical methods are based on simulation and can only provide approximate results. While the accuracy of the results can be arbitrarily improved, the price to be paid is the in
increased number of simulations, which increases the analysis time, possibly making it unreasonably long.

This paper presents qCORAL, a compositional statistical approach for the efficient quantification of solution spaces for arbitrarily complex constraints over bounded floating-point domains. Our compositional approach further leverages interval constraint propagation to improve the accuracy of the obtained results with a limited number of additional experiments.

qCORAL follows a layered compositional strategy. It splits up the problem of estimating the solution space of a set of symbolic conditions (describing the program paths leading to the occurrence of the target event) into the problem of estimating the solution space of individual path conditions. qCORAL further decomposes each path condition into a set of independent clauses that can be analyzed separately. It then composes the estimated results back together. This divide-and-conquer strategy speeds up the analysis by reducing large problems into sub-problems that are easier to analyze, and also allows the reuse of partial results for clauses appearing in several constraints. For each independent clause, qCORAL further uses an off-the-shelf interval constraint solver [14] to break the solution spaces into even smaller regions, whose union necessarily bounds the solution space of the clause. qCORAL then uses stratified sampling [29], a well-known technique for speeding up the convergence of Monte Carlo simulations, to analyze the data from the independent regions, and to compose the results.

We evaluated qCORAL on a set of benchmarks taken from the literature and a set of real-world software taken from medicine and aerospace domains. Preliminary results show significant improvement over previous research approaches [30] and built-in routines of general purpose mathematical tools, both in terms of accuracy of results and analysis time.

2. Background

2.1 Symbolic Execution

Symbolic Execution [7, 18] is a program analysis technique which executes programs on unspecified inputs by using symbolic inputs instead of concrete data. For each executed program path, the analysis builds a path condition PC, which is a conjunction of boolean conditions characterizing the inputs that follow that path. This PC is built according to the branching conditions in the code and it is checked for satisfiability using off-the-shelf solvers. If a PC becomes unsatisfiable it means that the corresponding path is not feasible in the program (and the analysis backtracks). The execution paths followed during the symbolic execution of a program are characterized by a symbolic execution tree. The nodes represent program (symbolic) states and the arcs represent transitions between states. Traditional applications of symbolic execution include test case generation and error detection, with many tools available [6, 13, 28, 31]. Symbolic execution of loopings 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.

2.2 Interval Constraint Propagation

Interval constraint propagation (ICP) [8] is an algorithmic approach to compute interval solutions to equality and inequality numerical constraints. The input is a list of n-variable constraints and the output is a list of n-dimensional boxes. Each constraint is an equality or an inequality constraint, possibly involving non-linear analytical expressions and not necessarily differentiable. A box is a characterization of a subset of the Cartesian product of the domains of input variables. Geometrically speaking, a box is an hyperplane, which generalizes a 2-dimensional plane for higher dimensions. The approach guarantees that the union of all boxes reported on output contains all solutions. Hence, a problem is unsatisfiable if the union of all boxes is empty. RealPaver [14] is a well-known implementation of the approach. Consider the constraint \((1.5 - (x \cdot (1.0 - y))) = 0.0\), the following is one example box that RealPaver reports on output:

\[ x : [99.99925650834012, 100] \]
\[ y : [0.9849998884754217, 0.9850000000000001] \]

Even though the low and high values of these intervals are floating-point numbers, the box denotes a 2-dimensional region of real solutions. RealPaver uses two parameters to determine how tight the are the boxes reported on output: the decimal bound and the time budget. The bound is the number of decimal digits that limits the size of the smallest box that RealPaver can report and the time budget limits the time for computing boxes. Note that, irrespective of these parameters, the output should always include all real solutions. The smaller the decimal bound and the higher the time budget the tighter are the boxes reported. The approach is flexible to support a wide range of mathematical functions.

3. Probabilistic Software Analysis

Probabilistic software analysis is concerned with quantifying how likely software execution is to satisfy a given property. Probabilistic software analysis is relevant in contexts where the software is designed to exhibit uncertain or randomized behavior [30] or when the execution environment, including interactions with users, is characterized by a probabilistic profile [4]. In these situations, it is usually more relevant to quantify the probability of satisfying (violating) a given property than to just assess the possibility of such events to occur.

Figure 1 shows the main flow of the tool that we use to support probabilistic analysis. The tool takes on input a Java program and the usage profile for the program’s input variables. The usage profile includes the domain of input variables and the probability distribution associated with each domain. The output of the tool is an estimate of the probability for satisfying (or violating) a property of interest, e.g., an assertion in the code or a certain observable event of interest. Internally, the tool uses symbolic execution to produce a set of path constraints. In particular we use Symbolic PathFinder (SPF) [24] for the symbolic execution of Java bytecode, but other similar tools can be used. In this context, satisfaction of one individual path constraint implies the occurrence of the event of interest. The work reported in this paper focuses on the last processor in the figure pipeline, the “Probabilistic Analysis” component, which takes as input a disjunction of path conditions and, based on that, it computes the probability of the event to occur. This component is not dependent on any programming language.

![Figure 1. Probabilistic software analysis chain.](image)

3.1 Computing Probabilities

Let’s classify the path constraints produced with symbolic execution in the sets \(PC^T\) and \(PC^F\) based on whether they lead to the occurrence of the target event (T) or not (F).

Under the assumption that symbolic execution terminates and only produces constraints for complete paths, the path constraints define disjoint input sets and they cover the whole input domain.
In order to deal with possible non-termination due to looping constructs in the code, SPF actually performs bounded symbolic execution. Hence, if an event has not occurred within the symbolic execution bound, the corresponding PC is not included in PC\(^T\). On the other hand, since hitting the execution bound is an event observable though SPF, it is possible to introduce a third set of PCs containing those where the bound has been hit and quantify the probability of such sets as well; this probability can give a measure for the confidence in the results obtained within the bound (the lower the probability the higher the confidence). This approach has been applied, for example in [11], but for the sake of space we will not consider it here.

We define the probability of satisfying a given property, as the probability of an input distributed according to the usage profile to satisfy any of the path constraints in PC\(^T\). Formally, what we aim to compute is:

\[
\int_D \bar{\#}_{PC^T}(x) \cdot p(x)
\]

where \(D\) is the input domain defined as the Cartesian product of the domains of the input variables, \(p(x)\) is the probability of an input \(x\) to occur for the given usage profile, and \(\bar{\#}_{PC^T}(x)\) is the indicator function on PC\(^T\), that is a function that returns 1 when \(x\) satisfies any of the PCs in PC\(^T\), and 0 otherwise [26]. Equation (1) represents the expected probability for satisfying the target property.

### 3.2 Monte Carlo Simulation

In general, the constraints in PC\(^T\) can be non-linear or can make the integral ill-conditioned for numerical solutions [17]. To retain generality, qCoral builds on simulation-based methods. These methods have theoretically no limitations on the complexity of the constraints they can handle. However, they can take a long time to converge, especially in the presence of large, multidimensional domains. The simplest simulation-based method applicable to the problem at hand is the Hit-Or-Miss Monte Carlo method [29]. In practice, it consists in generating a sequence of \(n\) independent inputs \((x_0, x_1, \ldots, x_{n-1})\), according to the usage profile, and to count the number of hits, i.e., the number of samples that satisfy the constraints under analysis. The ratio of hits (fraction of samples which are hits) provides an efficient, unbiased, and consistent estimator of the number of hits, i.e., the number of samples that satisfy the constraints under analysis. The ratio of hits, \(\bar{\#}_{PC^T}(x)/n\), is the sample mean. Note that the more samples are collected, the closer \(\hat{X}\) gets to the integral of Equation (1). In particular, this convergence can be quantified by the estimator variance \(\text{Var}\[\hat{X}\]\). The closer the variance is to 0 the more accurate is the estimation.

\[
E[\hat{X}] = \bar{x} \quad \text{Var}[\hat{X}] = \frac{\bar{x}(1-\bar{x})}{n}
\]

where \(\bar{x} = \sum_{i=0}^{n-1} \bar{\#}_{PC^T}(x_i)/n\) is the sample mean. As illustrated in Figure 2, the probability of satisfying the constraints can be easily computed as the ratio between the area of the triangle identified by the constraint and the size of the domain, and it is exactly 1/4. By using hit-or-miss Monte Carlo with 10\(^4\) samples, we obtain mean (i.e., expected value) .2577 and variance .1913. Let’s consider that ICP produces four boxes, as shown in the figure, containing all solutions. Let’s consider we take 2,500 samples within each box, thus overall still 10\(^4\) samples as in the case of the regular (non-stratified) hit-or-miss Monte Carlo approach. Table 1 shows the boxes coordinates for each of the boxes, size, and corresponding estimates. The part of the domain not covered by the four boxes accounts for 3/4 of the domain, and in such region we already know that both the mean and the variance of a hit-or-miss estimator would be 0, because there are no solutions are in there. If we
4.1 Disjunction of Path Conditions

combine the estimators of the single boxes as per Equation (3), we obtain mean 2.494 and variance 0.00586, which is a significant improvement, despite the relatively small number of samples we took within each region. Further evaluation of the accuracy achievable through this approach will be provided in Section 6.1.

4. Compositional Estimation for Complex Constraints

Simulation techniques are highly sensitive to the number of samples they take – see the number of samples factor “n” in the denominator of Equation 2. Efficiency is therefore essential for these techniques: improved efficiency means one can sample more within the same time budget and therefore reduce the estimate variance.

We propose a compositional approach to efficiently estimate the solution space for complex mathematical constraints, encoding the disjoint path conditions obtained through symbolic execution. Our approach leverages the following two observations:

- **Disjunction:** From the semantics of symbolic execution, an input cannot satisfy more than one path constraint in \( PC^T \). Each path constraint in this set denotes one equivalence class of the input space. We therefore decompose the estimation for \( PC^T \) into the estimation of each individual PC in the set.

- **Conjunction:** Each individual path constraint PC is a composition of simpler independent predicates that can be estimated separately; furthermore these predicates occur across multiple path constraints in \( PC^T \), and therefore their estimates can be efficiently re-used.

Section 4.1 describes an approximate composition rule for the disjunction of two path constraints in \( PC^T \) and Section 4.2 provides a heuristic for dealing with the conjunction of many path conditions.

4.1 Disjunction of Path Conditions

Consider the set \( PC^T = \{ PC_1^T, PC_2^T, \ldots, PC_m^T \} \) of the PCs leading to the occurrence of the target event. The elements \( PC_i^T \) are pairwise disjoint by construction. Let \( \tilde{X}_i^T \) be the estimator of the probability, given an input profile, of satisfying the constraints in \( PC_i^T \) (i.e. an estimator of the integral in (1)). Since there is no intersection between any two sets \( PC_i^T, PC_j^T \), we can define an estimator \( \hat{X}_i^T \) of the disjunction between \( PC_i^T \) and \( PC_j^T \) as follows:

\[
\hat{X}_i^T = \tilde{X}_i^T + \tilde{X}_j^T
\]  

(4)

Since there is no intersection between any two sets \( PC_i^T \) and \( PC_j^T \), the expected value of \( \hat{X}_i^T \) can be straightforwardly computed as [26]:

\[
E[\hat{X}_i^T] = E[\tilde{X}_i^T] + E[\tilde{X}_j^T]
\]  

(5)

The following theorem gives a bound on the variance for the composed estimator \( \hat{X}_i^T \), based on the variance computed for the individual path conditions:

**Theorem 1.** Let \( PC_i^T, PC_j^T \in PC^T \), \( \tilde{X}_i^T \) and \( \tilde{X}_j^T \) estimators of the probability of satisfying \( PC_i^T \) and \( PC_j^T \) respectively. Let \( \hat{X}_i^T = \tilde{X}_i^T + \tilde{X}_j^T \), then:

\[
\text{Var}[\hat{X}_i^T] \leq \text{Var}[\tilde{X}_i^T] + \text{Var}[\tilde{X}_j^T]
\]  

(6)

**Proof.** Since \( \hat{X}_i^T \) is defined as the sum of two random variables, the following general relation holds [26]:

\[
\text{Var}[\hat{X}_i^T] = \text{Var}[\tilde{X}_i^T] + \text{Var}[\tilde{X}_j^T] + 2 \cdot \text{Cov}[\tilde{X}_i^T, \tilde{X}_j^T]
\]

where the covariance \( \text{Cov}[\tilde{X}_i^T, \tilde{X}_j^T] = E[\tilde{X}_i^T \cdot \tilde{X}_j^T] - E[\tilde{X}_i^T] \cdot E[\tilde{X}_j^T] \).

We already know from Equation (1) that

\[
E[\tilde{X}_i^T \cdot \tilde{X}_j^T] = \int_{\mathcal{D}} \mathbb{1}_{PC_i^T}(x) \cdot \mathbb{1}_{PC_j^T}(x) \cdot p(x)
\]

but the product of the indicator functions in the equation above is always zero because, for every input, at most one of the PCs can be satisfied. On the other hand, the probability of satisfying a PC cannot be negative, thus \( \text{Cov}(\tilde{X}_i^T, \tilde{X}_j^T) \leq 0 \).

Theorem 1 allows us to process each PC separately (possibly also in parallel) and then compose the partial results to estimate the probability of satisfying any of the PCs in \( PC^T \). The price for this approach is the need to provide an overestimation of the variance of the composition, although, as will be shown in Section 6, in practical applications the benefits of the variance reduction techniques on analyzing the single disjuncts often overcome the loss due to the conservativeness we took for disjuncts composition.

4.2 Conjunction of Constraints

We now consider the problem of efficiently computing the probability of satisfying an individual path condition PC. For a real application, a path condition could be very large and may include many constraints relating many input variables. We propose to partition these constraints into smaller sets that have in common the input variables, and whose satisfiability therefore can be determined independently from one another. A path condition \( PC \) is defined by the conjunction of several simpler constraints \( c_1 \land c_2 \land \ldots \land c_m \). Each constraint \( c_d \) can make an assertion about one or more input variables \( v_j \), or functions thereof. Let indicate it by \( A(c_d, v_j) \).

Let us introduce a dependency relation (Dep) among the variables appearing in a program:

**Definition 1.** Given the path conditions obtained by the symbolic execution of a program \( PC \), such that \( PC = c_1 \land c_2 \land \ldots \land c_m \), and let \( V \) be the set of the input variables of the program. The relation \( \text{Dep} \subseteq V \times V \) is recursively defined by the following statements:

- \( \forall v \in V : \text{Dep}(v, v) \)
- \( \forall v, v_j \in V : \exists c_d \in c_1 \land c_2 \land \ldots \land c_m \) \( A(c_d, v_j) \land A(c_d, v) \Rightarrow \text{Dep}(v, v_j) \)
- \( \forall v, v_j, v_k \in V : \text{Dep}(v, v_j) \land \text{Dep}(v_j, v_k) \Rightarrow \text{Dep}(v, v_k) \)

Intuitively, two input variables are depending on each other if they appear together in at least one constraint in the path condition. If this is the case, to evaluate the satisfaction of such conditions the assignments to both variables has to be evaluated at the same time.

Notice that Dep is by construction an equivalence relation over \( V \) and for this reason it induces a partition of \( V \). We refer to the sets composing this partitions as \( \{ \bar{V}_0, \bar{V}_1, \ldots, \bar{V}_m \} \). We can now extend the definition of \( A(v, \cdot) \) by stating that \( A(c_d, v_j) \) holds if there exists a variable \( v_j \in V \) such that \( A(c_d, v_j) \) holds.

Consider now a path condition \( PC \). Let us define the constraints \( \bar{C}_j = \bigwedge_{c_k \in A(c_d, v_j)} A(c_k, v) \). That is, \( \bar{C}_j \) is the conjunction of all the constraints \( c_k \) occurring in \( PC \) and containing any of the variables \( v_j \in V \). The probability of satisfying \( PC \) is the same as the probability of satisfying all the constraints \( \bar{C}_j \) at the same time. But since two constraints \( \bar{C}_j \) and \( \bar{C}_k \) do not share any variable by construction, the satisfaction of one of them is independent from the satisfaction of the other. This independence is straightforwardly
inherited by the hit-or-miss estimators \( \hat{X}_{ij} \) and \( \hat{X}_{ik} \) of the probability of satisfying \( C_{ij} \) and \( C_{ik} \), respectively. Indeed, the Monte Carlo procedures will generate random assignments for the sets of variables \( V_j \) and \( V_k \) independently.

Thanks to the independence of the estimators \( \hat{X}_{ij} \) and \( \hat{X}_{ik} \) of the probabilities of satisfying the constraints \( C_{ij} \) and \( C_{ik} \), the probability of \( C_{ij} \land C_{ik} \) can be estimated by \( \hat{X}_{jk} = \hat{X}_{ij} \cdot \hat{X}_{ik} \) having [26]:

\[
E[\hat{X}_{jk}] = E[\hat{X}_{ij}] \cdot E[\hat{X}_{ik}]
\]

\[
\text{Var}[\hat{X}_{jk}] = E[\hat{X}_{ij}]^2 \cdot \text{Var}[\hat{X}_{ik}] + E[\hat{X}_{ik}]^2 \cdot \text{Var}[\hat{X}_{ij}] + \text{Var}[\hat{X}_{ij}] \cdot \text{Var}[\hat{X}_{ik}]
\]

(7)

(8)

4.3 Observation

By applying the composition methods introduced in this section, we may obtain the following benefits. First, we can split the analysis of a large path condition into the analysis of several simpler constraints. Second, each of the simpler constraints will possibly involve only a subset of the input variables, making the generation of the samples faster. Furthermore, reducing the input space to only the variables actually involved in the constraint, would likely result to a better coverage of the input sub-domain, for a fixed number of samples.Finally, we remark that the simpler constraints can be analyzed efficiently using the interval constraint propagation with Monte Carlo stratified sampling described in the previous section.

4.4 Example

The following code snippet mimics a safety monitor for an autopilot navigation system. If the altitude is less than 9000 meters, the autopilot is allowed to control the position of the vehicle by manipulating the front and head flaps through floating point variables headFlap and tailFlap. These variables range in the interval \([-10,10]\).

```java
if (altitude <= 9000) {
    if (Math.sin(headFlap+tailFlap)>0.25) {
        callSupervisor();
    } else {
        callSupervisor();
    }
}
```

Listing 1. Example of events and complex conditions in Java.

If during the flight the safety monitor realizes that the relative position of the flaps violates the imposed safety conditions, it calls a human supervisor to take over the control of the vehicle. The same position of the flaps violates the imposed safety conditions, it calls a human supervisor to take over the control of the vehicle. The same position of the flaps violates the imposed safety conditions, it calls a human supervisor to take over the control of the vehicle. The same position of the flaps violates the imposed safety conditions, it calls a human supervisor to take over the control of the vehicle. The same position of the flaps violates the imposed safety conditions, it calls a human supervisor to take over the control of the vehicle.

5. Algorithms and Implementation

This section provides some implementation details of qCORAL. The input of qCORAL includes a set of disjoint constraints \( \text{PCs} \), representing the path conditions leading to the occurrence of the target event as obtained from the symbolic execution stage (see Section 3), and a description of the input domain, i.e., a map from floating-point input variables to their domain. The domain of each variable is a closed interval. We used SymbolicPathFinder [28] as our symbolic execution engine.

Algorithm 1 describes the main loop of qCORAL. It iterates over the input path constraints, processes each one, and combines the partial results as described in Section 4.1. Note that qCORAL returns an upper bound for the variance of a disjunction of path constraints, as defined in Equation (6). The procedure \( \text{computeDependencyRelation} \) returns the partition of the input variables according to Definition 1. To efficiently implement this procedure we rely on the Jung graph library\(^2\). We create an undirected graph, where a node corresponds to an input variable and an edge corresponds to a dependence between two variables. This procedure visits every path condition and when two variables appear in the same constraint it adds an edge between their corresponding nodes. The procedure then computes the weakly connected components of the graph (with complexity at most quadratic in the number of input variables).

Algorithm 2 computes results for one conjunctive clause. The procedure \( \text{extractRelatedConstraints} \) goes through all the con-

\(\text{Algorithm 1: qCORAL}
\)

\textbf{Input:} \( \text{PCs}, D \)

\textbf{Output:} mean, \( \text{var} \)

\begin{align*}
\text{dep} & \leftarrow \text{computeDependencyRelation}(\text{PCs}) \\
\text{mean} & \leftarrow 0; \text{var} \leftarrow 0
\end{align*}

\begin{algorithmic}
\For {pc \in \text{PCs}}
\State \( (\text{pcMean, pcVar}) \leftarrow \text{analyzeConjunction}(pc, D, \text{dep}) \)
\State \text{mean} \leftarrow \text{mean} + \text{pcMean}
\State \text{var} \leftarrow \text{var} + \text{pcVar}
\EndFor
\State \text{return} (\text{mean, var});
\end{algorithmic}

\(\text{Algorithm 2 computes results for one conjunctive clause. The procedure } \text{extractRelatedConstraints} \text{ goes through all the con-}\)

\(\text{Algorithm 2 computes results for one conjunctive clause. The procedure } \text{extractRelatedConstraints} \text{ goes through all the con-}\)

\(\text{http://jung.sourceforge.net}\)
plexified as that, when caching is active, the error in local sampling may be am-
made a trade-off between accuracy and efficiency. It is possible
to evidence of their estimator with all the others. Conceptually caching
be safely stored in a cache and reused thanks to the global indepen-
dent optimization, the results obtained for an independent constraint can
be increased. Section 6 evaluates the impact of caching in terms of precision and time.

for varSet ∈ dep do
part ← extractRelatedConstraints(pc, varSet)
if !cache.contains(part) then
  stratSampling(part, varSet, D)
  cache.put(part, ⟨partMean, partVar⟩)
⟨partMean, partVar⟩ ← cache.get(part)
mean ← mean + partMean
if var = Nil then
  var ← partVar
else
  mean ← mean + partVar + partMean^2 + var + var * partVar

return ⟨mean, var⟩;

Algorithm 2: analyzeConjunction

Algorithm 3: stratSampling

Input: part, varSet, D
Output: mean, var
mean ← 0; var ← 0
boxes ← icp(part, varSet, D)
for box ∈ boxes do
  ⟨boxMean, boxVar⟩ ← hitOrMiss(part, varSet, D)
  boxWeight ← size(box)/size(D)
  mean ← mean + boxWeight * boxMean
  var ← var + boxWeight^2 * boxVar
return ⟨mean, var⟩;

6. Evaluation

Our evaluation addresses the following research questions:

- RQ1. What is the accuracy of qCORAL estimates?
- RQ2. How does qCORAL compares with the built-in numerical integra-
tion routines of Mathematica and with VolComp when handling linear constraints?
- RQ3. How the different features of qCORAL affect accuracy and time when handling complex constraints?

All experiments have been run on an Intel Core i7 920 64-bit machine, with 2.67GHz and 8GB, running Ubuntu 12.04.

6.1 RQ1: What is the accuracy of qCORAL estimates?

This experiment evaluates how accurate qCORAL is for computing the volume of several geometric figures for which analytical solutions are widely known. We used the symbolic integration routines of Mathematica to obtain the exact volumes of these solids, while for qCORAL we computed them as the fraction of a domain of known size.

Table 2 summarizes the results. We evaluated the accuracy of our approach for 10^3, 10^4, 10^5, and 10^6 as maximum number of samples for the Monte Carlo procedure. We run 30 times each configuration and reported the average value and standard deviation over the population of estimated volumes.

We grouped our subjects in three groups: convex polyhedra, solids of rotation, and intersection of solids. Except for the convex polyhedra cases, all other subjects contain non-linear constraints and mathematical functions, namely exponentiation and square root. For these subjects the compositional approach does not provide benefits, since the three variables characterizing each solid are tightly dependent one another. We will come back to compositionality later in this section.

All the experiments completed within 2 seconds, with the exception of Icosahedron and Rhombicuboctahedron for 10^6 samples (4 and 7 seconds, respectively). Notably, thanks to Interval Constraint Propagation (ICP) and stratified sampling, even with a relatively small number of samples, qCORAL provided a reasonably accurate result for most of the subjects.

Finally, consider the case of Cube: the standard deviation is 0 because, regardless the number of samples, qCORAL was always able to find the exact solution thanks to ICP. Indeed, the real subject a 3D box and RealPaver can exactly identify it, driving the estimation error to zero.

1 http://www.wolfram.com/mathematica
6.2 RQ2: How qCORAL compares with Mathematica and VolComp

This experiment evaluates how qCORAL (with both stratified sampling and the compositional analysis) compares with existing techniques for quantifying solution spaces of linear constraints. As baseline for comparison, we use the built-in procedure for numerical integration of the commercial tool Mathematica, NIntegrate. This procedure is available off-the-shelf and we run it with its default settings. We also include VolComp, a recently developed tool producing as output a tight closed interval over the real numbers containing the requested solution.

The built-in numerical integration routine of Mathematica performs a recursive procedure called Global Adaptive Integration [21]. On each recursive call, it analyzes the behavior of the integrand function within the integration domain and automatically selects an integration rule to apply. Mathematica supports a broad variety of integration rules, e.g. Trapezoidal, Simpson, or Newton-Cotes. After each iteration, the procedure analyzes the obtained results and if they do not meet the accuracy goals, the integration domain is bisected and the integration routine is invoked on each of the two parts. Partial results are then combined, similarly to the case of stratified sampling. The procedure terminates when the default accuracy requirements are met or when the recursion depth limit has been reached. Since NIntegrate guarantees the accuracy of the result (or gives a notification when its requirements are not met), it provides a reference to evaluate how tight are the intervals produced by VolComp and how precise are the estimates produced by qCORAL.

The techniques we evaluate in this experiment report solutions in different formats. Mathematica reports a single point solution,
which is exact up to an informed bound. VolComp reports an interval solution that bounds the exact point solution, and qCORAL reports an estimate on the exact solution and a statistical variance for that estimate. In place of the variance, for qCORAL we report the standard deviation (square root of the variance), which is in the same unit scale of the estimate. Furthermore, since qCORAL implements a randomized algorithm, the estimate and the standard deviation we report are averaged over 30 executions.

To compare the tools, we used the VolComp benchmark, which is publicly available [2]. The subjects we selected from the benchmark are: the Framingham atrial fibrillation risk calculator (Atrial), a steering controller to deal with wind disturbances (Cart), the Framingham coronary risk calculator (Coronary), an eGFR estimator of chronic kidney’s disease (Egfr Epi and Egfr Epi Simple), an inverted pendulum (Invpend), a model of a robot that packs objects of different weights in a carton with the goal to maximize total weight within a certain limit (Pack), and a controller for filling up a tank with fluid (Vol).

We re-ran the benchmarks from [2] using their scripts. The PCs were obtained by translating PCs produced with their frontend. All experiments ran until completion. Table 3 shows the comparison between qCORAL and VolComp [2]. Notice that the two techniques provide different types of results. VolComp uses iterative bounding to return an interval containing the solution. qCORAL returns a statistical estimator characterized by its expected value and its variance: the expected value is the most likely value as it comes out from qCORAL analysis; the variance provides a measure of the uncertainty of the estimate; such uncertainty could be used to quantify the probability the real value belongs to an interval, for example by using Chebyshev’s inequality [26].

The first 4 columns of Table 3 state the targeted assertion, how many paths reach the assertion true, how many conjuncts (and) are there on these paths and lastly how many arithmetic operations (with the number of unique operations). For each tool we then show the estimated probability of satisfying the assertions specified in the first column. We make the following observations from these results:

- **ARTRIAL.** For one of the three assertions Mathematica takes very long to produce results and VolComp takes more than 10m to finish. qCORAL produces an accurate result very quickly.
- **CART.** This subject produces a highly skewed polynomial [30], which is known to be a case where branch-and-bound techniques do not perform well. Indeed, results of VolComp are not good for this case. qCORAL also suffers as it relies on RealPaver which also uses branch-and-bound techniques. For this reason, we observe a high standard deviation in our results relative to other subjects. Note that despite this limitation qCORAL can still report an estimate close to the exact solution. Mathematica can easily handle the subject in this case.
- **CORONARY and EGFR.** These are the best cases for qCORAL. As compared to other tools, qCORAL reports results very quickly and precisely.
- **PACK.** qCORAL is slower for the last two cases of PACK. The reason for this high cost is the high number of paths and the high interdependence among the variables on each path constraint, which reduces the impact of our divide and conquer strategy for conjuncts. We observed that for one case in this subject Mathematica actually misses the interval; this is highlighted in grey color. This happened because the default settings of NIntegrate does not allow to collect enough points for the numerical integration to converge. This situation is reported by the tool. Note that on some cases rounding the result to four decimal digits made the results of VolComp be different than those of Mathematica and qCORAL by the last digit. These have to be considered just rounding error.
- **VOL** is a case that stresses qCORAL. Mathematica finishes in more than 20m and VolComp returns the full range 0.0-1.0, i.e. it did not perform any pruning. qCORAL reports the expected result of ~1.0. Note that the estimate that qCORAL reports is actually slightly greater than 1, for this case. This is due to the propagation of errors in the estimation which magnifies when the exact probability is close to the corner case 1. Mathematica also reports a result > 1 due to the finite accuracy of numerical integration, partially due to the use of off-the-shelf settings for such complex subject.

In summary, we observed that qCORAL reports estimates very close to the exact point solutions that Mathematica’s numerical integration method reports (apart from the case of PACK where Mathematica reported no convergence and a wrong). However, numerical integration is potentially expensive when the number of variables grows [27]. This observation is confirmed for the cases of ATRIAL and PACK, where Mathematica takes respectively more than 1h to complete. In most of the experiments, especially those on complex subjects, qCORAL resulted faster than both off-the-shelf use of Mathematica’s numerical integration function and VolComp. For specific problems advanced settings of Mathematica may improve its performance, though a deeper understanding of the mathematical nature of the problem might be required, which might be not straightforwardly derived from the code under analysis. We also notice that the results of qCORAL have been consistent with those of VolComp: the qCORAL estimates fall within the corresponding VolComp intervals almost always, up to the accuracy (last decimal digit might be dangling because of rounding); the only exception is f1 > f > 0.1, where the variance is anyway large enough to account for the deviation (3.29×10⁻⁴). Concluding, we observe that qCORAL has provided a reasonable balance between time efficiency and precision when handling linear constraints on this benchmark suite.

### 6.3 RQ3: How the different features of qCORAL affect accuracy and time when handling complex constraints?

This experiment evaluates how different configurations of qCORAL compare with respect to precision and time. We considered two implementations of Monte Carlo Hit-or-Miss: one from Mathematica (baseline) and one from qCORAL: qCORAL{1}. The empty braces indicate that no feature from qCORAL has been enabled in this configuration. The configuration qCORAL[STRAT] incorporates stratified sampling in the analysis of individual path conditions. The configuration qCORAL[STRAT, PARCACHE] cumulatively incorporates partitioning and caching. We considered the following subjects from the aerospace domain in this experiment:

- **Apollo.** The Apollo Lunar Autopilot is a Simulink model that was automatically translated to Java using the Vanderbilt toolset [23]. The model is available from MathWorks ⁶. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g. Math.sqrt). The code contains 2.6KLOC, deployed in a single package with 54 classes. We analyzed 5,779 path constraints for this subject.
- **TSAFE.** The Tactical Separation Assisted Flight Environment (TSAFE) is designed to prevent near misses and collisions of

---

⁴ [http://www.framinghamheartstudy.org](http://www.framinghamheartstudy.org)

⁵ [http://nephron.com/epi_equation](http://nephron.com/epi_equation)

aircraft that are predicted to happen in the near future (from 30s to 3m). The Conflict Probe module of TSAFE tests for conflicts between a pair of aircraft within a safe time horizon. The two aircraft may either be flying level or engaged in turns of constant radius. The following math functions appear in the path constraints of this subject: \( \cos, \sin, \tan, \) and \( \sqrt{\cdot} \). The Turn Logic module of TSAFE computes the change in heading required once an impending loss of separation between two aircraft is detected. It assumes a constant turning radius for the aircraft making the maneuver. Path constraints for this subject contain the \( \tan^{-1} \) function. Each of these modules is about 50 LOC. We analyzed respectively 23 path constraints on Conflict and 225 path constraints on Turn Logic.

These case studies contain complex constraints. They have 2.6KLOC for Apollo and 50 each for the two TSAFE components analyzed. Since no properties were defined, instead of fabricating a property, we generated all the PCs (using Symbolic PathFinder with search bound equal to 50) and selected a percentage of PCs to quantify the path probabilities. We generated 5779 PCs for Apollo and 225 for TSAFE; the latter is smaller but involves complex mathematical functions and has high dependence among variables. We arbitrarily picked the first 70% of the PCs in a bounded depth-first order. We picked 70% of the paths so to avoid obtaining a probability close to 0 or 1, not to bias the evaluation for the Monte Carlo estimation towards its worst cases (some extreme cases have already been evaluated in Table 3). This selection mimics a property that is satisfied on some of the paths and not on the rest.

Table 4 shows experimental results. Note the reported estimations are not exactly 70% because different PCs have different solution spaces. Different columns show different configurations of qCORAL, each line denotes one different subject, and each group of lines denotes a different maximum number of samples allowed for simulation in qCORAL. Results show that the addition of feature STRAT results in a significant reduction in the variance of the estimate at the expense of an overhead in time. This additional cost is justified by the multiple invocations of the ICP solver; one call per disjunctive clause of the input formula. Adding the feature PARTCACHE, i.e. performing compositional analysis, may further improve precision (see Section 4), however, considering the most time-consuming cases, it always reduces time. Increasing the maximum sample size from 1K to 100K results in higher time savings as the cost for analyzing each subproblem increases. While the relationship between execution time and maximum number of samples is approximatively linear for the basic configuration, the impact of stratified sampling and compositionality on execution time depends on the specific subject.

### Table 4. Comparison of different configurations of qCORAL on different sampling rates.

<table>
<thead>
<tr>
<th>subject</th>
<th>Mathematics</th>
<th>Monte Carlo</th>
<th>qCORAL{STRAT}</th>
<th>qCORAL{STRAT,PARTCACHE}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>estimate (\sigma) time(s) estimate (\sigma) time(s) estimate (\sigma) time(s) estimate (\sigma) time(s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apollo</td>
<td>0.62521  0.02126 65.44 0.01342 18.17 0.62231 0.01262 62.98 0.62461 0.00972 45.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conflict</td>
<td>0.49794  0.01308 6.66 0.01691 0.44 0.50016 0.00027 1.53 0.50016 0.00027 1.65</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Turn Logic</td>
<td>0.72254  0.02250 2.18 0.71913 0.01338 0.64 0.72282 0.01462 1.22 0.72282 0.01462 1.62</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 7. Related Work

There are many related works to ours, including probabilistic abstract interpretation [22], probabilistic model checking [16] and volume computations [9]. A comprehensive list of related work that is relevant here too can be found in [30]. Here we focus on the works we consider the most closely related.

We calculate the probability of a path, or more generally a set of paths, being executed. Typically these paths lead to an event of interest, such as an assertion violation for example. Techniques for doing such probabilistic analyses differ in the type of input distributions they consider, the language features supported, and the approach used to calculate the number of solutions.

One approach in this area was that of Geldenhuys et al. [12] that considered uniform distributions for the inputs, linear integer arithmetic constraints, and used LattE Machiato [9] to count solutions of path conditions produced during symbolic execution. Their goal was to calculate the probability of covering branches and assertion violations in the code. One of the main technical differences between this work and others based on symbolic execution (including the present paper) is that probabilities are calculated at each branching point rather than after symbolic execution is finished. Here we support complex constraints, including non-linear constraints, and we use a statistical approach to count solutions. Our proposed approach is compositional and hence we believe it is applicable in the incremental setting from [12].

Sankaranarayanan et al. [30] and Filieri et al. [11] recently proposed similar techniques to compute probabilities of violating state assertions. Both techniques remove the restriction of uniform distributions, although in the latter case it is by discretizing the domain into small uniform regions. As with [12] both approaches only consider linear constraints. Their techniques also build on symbolic execution to compute a relevant set of symbolic paths of a program that leads execution to some assertion of interest. Both techniques estimate probability of exercising those paths and violating the corresponding assertion from such a set. Sankaranarayanan et al. developed a customized algorithm for under and over-approximations of probabilities. They use Linear Programming (LP) solvers to compute over-approximations and heuristics-based “ray-shooting” algorithms to compute under-approximations, which is applicable for convex polyhedra. Filieri et al. used the LattE Machiato [9] tool to compute probabilities. Our technique complements the works of Sankaranarayanan et al. and Filieri et al. by supporting a wider range of constraints. Experimental results show that although there is a potential loss in precision by using our analysis, when that loss occurs it is small.

Bouissou et al. [5] handles non-linear 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 un-
certainty of non-linear computations. However they cannot handle conditional branches as we can here. More recently Adje et al. [3] extended this work to allow conditional statements as well. The main difference with our approach lies in that they use an abstraction based approach whereas we use a statistical approach. We can thus handle a wider set of non-linear constraints such as complex mathematical functions (sine, cosine, etc.). In future work we would like to do an empirical comparison between these two approaches on the examples that both can handle.

Pavese et al. [25] propose the use Monte Carlo simulation and model inference [10] to optimize probabilistic model checkers, such as PRISM [16]. Conceptually, the idea is to obtain a symbolic computational model from a set of traces obtained with sampling and feed that model to the model checker. Experimental results show that their technique can obtain tighter lower bounds for the mean time to first failure (MTTF) of network protocol models with rare failures. Our work is orthogonal to theirs. Our technique applies to imperative programs, such as those for controlling vehicles while their technique applies to probabilistic programs. Our technique expects as input a summary of the behavior characterized with a set of complex constraints.

8. Conclusion

We have developed a compositional statistical approach for the quantification of solution spaces for arbitrary complex mathematical constraints with application to probabilistic software analysis. The approach is enhanced by an acceleration procedure to improve the accuracy of the simulation based analysis with a limited number of additional experiments. Our initial experimental results are promising. In principle, our statistical approach can be applied to both integer and floating-point domains, but in practice it performs poorly for integer constraints if compared to counting-based techniques (particularly problematic are the equality constraints). Our future work will therefore focus on handling mixed integer and floating-point constraints, possibly through a combination with exact volume computations or guidance heuristics. We also plan to use the compositional approach in an incremental symbolic execution setting and to combine the technique with a statistical (as opposed to systematic) exploration of program paths as guided by the estimated conditional probabilities for the conditions in the code.

Acknowledgments

This work is partly supported by NSF Awards CCF-1329278 and CCF-1319858 (Păsăreanu). Mateus Borges is supported by FACEPE graduate fellowship number IBPG-0668-1.03/12.

References