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 asses 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 transcedental 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-

PLDI '14, June 9 - 11 2014.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

creased 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 aerospatial 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 looping programs may result in an infinite symbolic execution tree. For this reason, symbolic execution is typically run with a (user-specified) bound on the search depth.

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 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.

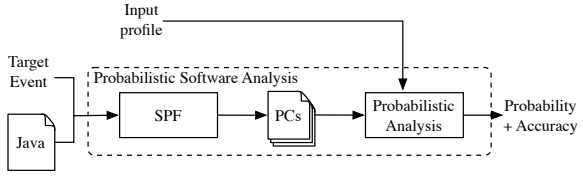

Figure 1. Probabilistic software analysis chain.

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

[18, 28]. However, 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:

$$\int_D \mathbb{1}_{PC^T}(\mathbf{x}) \cdot p(\mathbf{x}) \tag{1}$$

where D is the input domain defined as the Cartesian product of the domains of the input variables, $p(\mathbf{x})$ is the probability of an input $\mathbf{x}$ to occur for the given usage profile, and $\mathbb{1}_{PC^T}(\mathbf{x})$ is the indicator function on PC^T, that is a function that returns 1 when $\mathbf{x}$ 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 $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$, 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 for the probability of satisfying the constraints [26]. In particular such estimator $\hat{X}$ has a binomial distribution with expected values and variance given by the following equations [26]:

$$E[\hat{X}] = \bar{x} \qquad \mathrm{Var}[\hat{X}] = \frac{\bar{x} \cdot (1-\bar{x})}{n} \tag{2}$$

where $\bar{x} = \sum_{i=0}^{n-1} \mathbb{1}_{PC^T}(\mathbf{x}_i)/n$ 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* Var: the closer the variance is to 0 the more accurate is the estimation.

3.3 Interval Constraint Propagation and Stratified Sampling

Despite their generality, hit-or-miss Monte Carlo methods may suffer from a slow convergence rate [17], especially if the probability of the target event gets close to zero [32]. If additional information is known about the problem under analysis, it can be exploited to improve the estimation performance, i.e. to reduce the estimator variance. A well established method for variance reduction is the *stratified sampling* [19, 29]. This approach consists in partitioning the sample space, which in our case corresponds to the input domain D, into disjoint subsets $(\{R_0, R_1, \ldots, R_m\})$, called *strata*. Each strata R_i can be analyzed separately via hit-or-miss Monte Carlo, obtaining the corresponding estimator $\hat{X}_i$. Since the sampling processes within each region is independent, and assuming we take the same number of samples on each strata [29], the strata estimators can be combined to obtain an estimator $\hat{X}$ over the original sample space, with the following expected value and variance [29]:

$$E[\hat{X}] = \sum_i w_i \cdot E[\hat{X}_i] \qquad \mathrm{Var}[\hat{X}] = \sum_i w_i^2 \cdot \mathrm{Var}[\hat{X}_i] \tag{3}$$

where w_i is defined as $w_i = \mathrm{size}(R_i)/\mathrm{size}(D)$ and it denotes the size of region R_i relative to the size of the domain D. Since the strata constitute a partition of D, $\sum_i w_i = 1$. The expected values and variance of the estimators $\hat{X}_i$ are obtained with Equation (2).

It has been shown that the variance of an estimator obtained through stratified sampling cannot be worse than the variance obtained running Hit-or-Miss Monte Carlo integration on the entire domain, although only a suitable choice of strata provides significant benefits [29]. Optimal stratification criteria can be defined for each problem, although this usually requires additional information about the problem, which is not always available or easy to estimate [19].

In our case, the set of constraints composing a PC are assumed to be formalized as the conjunction of mathematical inequalities, which may involve also non-linear analytical functions. By definition, the indicator function of Equation (1) evaluates to 1 iff the input satisfies the path constraint, that is, if it belongs to the locus of the solutions of the system of mathematical inequalities composing it. The exact computation of these solutions is usually infeasible, but interval constraint propagation can help to identify a set of subregions of the domain containing them. For this paper, we use the tool RealPaver [14], which provides interval constraint propagation for systems of non-linear inequalities over real domains. Given a set of inequalities and a bounded domain, RealPaver identifies a set of non-overlapping boxes whose union contain all the solutions of the problem. The boxes may be tight, meaning they only contains solutions, or loose, containing both solutions and other points. Since no solution exists outside those boxes, there is no need to analyze such region through hit-or-miss Monte Carlo, since we already know that the estimator would converge to 0, with variance 0, since the integral of Equation (1) evaluates to 0.

Example. As an example of the use of Interval Constraint Propagation (ICP) for variance reduction, let us consider the constraint $x \leq -y \ \wedge \ y \leq x$, where values of variables x and y are uniformly distribution over the domain $[-1, 1]$. 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 .19131. 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

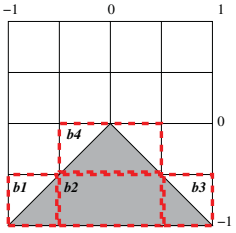

Figure 2. Example illustrating interval constraint propagation with stratified sampling.

	x	y	w	$E[\hat{X}]$	$\mathrm{Var}[\hat{X}]$
b_1	$[-1,-.5]$	$[-1,-.5]$	$.25/4$	.5012	.2501
b_2	$[-.5,-.5]$	$[-1,-.5]$	$.5/4$	1	0
b_3	$[.5,1]$	$[-1,-.5]$	$.25/4$	.508	.25
b_4	$[-.5,.5]$	$[-.5,0]$	$.5/4$	.4904	.25

Table 1. Variance reduction for the example case.

combine the estimators of the single boxes as per Equation (3), we obtain mean .2494 and variance .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 constraints.

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 $\hat{X}_i^T$ 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 $\hat{X}^T$ of the disjunction between PC_i^T and PC_j^T as follows:

$$\hat{X}^T = \hat{X}_i^T + \hat{X}_j^T \tag{4}$$

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

$$E\left[\hat{X}^T\right] = E\left[\hat{X}_i^T\right] + E\left[\hat{X}_j^T\right] \tag{5}$$

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

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

$$Var\left[\hat{X}^T\right] \leq Var\left[\hat{X}_i^T\right] + Var\left[\hat{X}_i^T\right] \tag{6}$$

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

$$\mathrm{Var}\left[\hat{X}^T\right] = \mathrm{Var}\left[\hat{X}_i^T\right] + \mathrm{Var}\left[\hat{X}_i^T\right] + 2 \cdot \mathrm{Cov}\left[\hat{X}_i^T, \hat{X}_j^T\right]$$

where the covariance $\mathrm{Cov}[\hat{X}_i^T, \hat{X}_j^T] = E[\hat{X}_i^T \cdot \hat{X}_j^T] - E[\hat{X}_i^T] \cdot E[\hat{X}_j^T]$. We already know from Equation (1) that

$$E[\hat{X}_i^T \cdot \hat{X}_j^T] = \int_D \mathbb{1}_{PC_i^T}(\mathbf{x}) \cdot \mathbb{1}_{PC_j^T}(\mathbf{x}) \cdot p(\mathbf{x})$$

but the product of the indicator functions in the equation above is alway 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 $\mathrm{Cov}(\hat{X}_i^T, \hat{X}_j^T) \leq 0$. □

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_i is defined by the conjunction of several simpler constraints $c_{i0} \wedge c_{i1} \wedge \ldots c_{im}$. Each constraint c_{ik} can make an assertion about one or more input variables v_j, or functions thereof. Let indicate it by $A(c_{ik}, v_j)$.

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 $\{PC_i\}$, such that $PC_i = c_{i0} \wedge c_{i1} \wedge \ldots c_{im}$, and let V be the set of the input variables of the program. The relation $Dep \subseteq V \times V$ is recursively defined by the following statements:*

- $\forall v \in V \ : \ Dep(v, v)$
- $\forall v_i, v_j \in V \ : \ \exists c_{kl} \, A(c_{kl}, v_i) \wedge A(c_{kl}, v_j) \implies Dep(v_i, v_j)$
- $\forall v_i, v_j, v_k \in V \ : \ Dep(v_i, v_k) \wedge Dep(v_k, v_j) \implies Dep(v_i, v_j)$

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. Let us refer to the sets composing this partitions as $\{\tilde{V}_0, \tilde{V}_1, \ldots, \tilde{V}_l\}$. We can now extend the definition of $A(\cdot, \cdot)$ by stating that $A(c_{ik}, \tilde{V}_j)$ holds if there exists a variable $v_j \in \tilde{V}_j$ such that $A(c_{ik}, v_j)$ holds.

Consider now a path condition PC_i. Let us define the constraints $\tilde{C_{ij}} = \bigwedge_{c_{ik}: A(t_{ik}, \tilde{V}_j)} c_{ik}$. That is, $\tilde{C_{ij}}$ is the conjunction of all the constraints c_{ik} occurring in PC_i and containing any of the variables from $\tilde{V}_j \subseteq V$. The probability of satisfying PC_i is the same as the probability of satisfying all the constraints C_{ij} at the same time. But since two constraints C_{ij} and C_{ik} do not share any variable by construction, the satisfaction of one of them is independent from the satisfaction of the other[1]. This independence is straightforwardly

[1] As support of this statement, notice that $Pr(C_{ij} | v_k = \bar{v_k}^1) = Pr(C_{ij} | v_k = \bar{v_k}^2)$, where $v_k \notin \tilde{V}_j$ and $\bar{v}_k^1$ and $\bar{v}_k^2$ are any two valid assignments for v_k. Thus, we can conclude that $Pr(C_{ij} | v_k = \bar{v_k}^1) = Pr(C_{ij})$.

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 $\tilde{V}_j$ and $\tilde{V}_k$ independently.

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

$$\mathrm{E}\left[\hat{X}_{ijk}\right] = E\left[\hat{X}_{ij}\right] \cdot E\left[\hat{X}_{ik}\right] \quad (7)$$

$$\mathrm{Var}\left[\hat{X}_{ijk}\right] = E\left[\hat{X}_{ij}\right]^2 \cdot \mathrm{Var}\left[\hat{X}_{ik}\right] + E\left[\hat{X}_{ik}\right]^2 \cdot \mathrm{Var}\left[\hat{X}_{ij}\right] + \mathrm{Var}\left[\hat{X}_{ij}\right] \cdot \mathrm{Var}\left[\hat{X}_{ik}\right] \quad (8)$$

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 $[-10, 10]$.

```
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 procedure is actuated if the vehicle get over 9000 meters of altitude.

For this example, our target event is the call of a supervisor and the code substantiates this event with function `callSupervisor`. We want to quantify the probability of such event to occur given a certain probabilistic profile for variables `headFlap`, `tailFlap`, and `altitude`. Such profile can be obtained by monitoring data of similar vehicles. For the sake of simplicity, let us consider an uniform profile for the three variables over their respective domains. We restrict altitude to the range $[0, 20000]$. Symbolic execution produces the following path constraints that reach our target event:

$PC_1^T : altitude > 9000$

$PC_2^T : altitude \leq 9000 \wedge Math.sin(headFlap \cdot tailFlap) > 0.25$

We want to quantify the solution space over $PC^T = \{PC_1^T, PC_2^T\}$. We illustrate how `qCORAL` analyzes these constraints in the following.

PC_1^T consists of a single atomic constraint predicating only on the value of variable `altitude`, thus no further decomposition is possible. This constraint can be analyzed by applying the techniques introduced in Section 3, obtaining for its statistical estimator $\hat{X}_1^T$: $E\left[\hat{X}_1^T\right] = 0.55$ and $\mathrm{Var}\left[\hat{X}_1^T\right] = 0$. Note that the variance of the estimator is 0 thanks to the help of ICP solver, which is able to identify one tight a box for this constraint.

Algorithm 1: `qCORAL`

Input: PCs, D
Output: $mean, var$
$dep \leftarrow$ computeDependencyRelation(PCs)
$mean \leftarrow 0$; $var \leftarrow 0$
for $pc \in PCs$ **do**
  $\langle pcMean, pcVar \rangle \leftarrow$ analyzeConjunction(pc, D, dep)
  $mean \leftarrow mean + pcMean$
  $var \leftarrow var + pcVar$
return $\langle mean, var \rangle$;

In contrast to PC_1^T, PC_2^T is a conjunction of two boolean expressions. According to Definition 1, we know that variables *headFlap* and *tailFlap* depend on one another, while *altitude* does not depend on any other variables. For this reason, as discussed in Section 4.2, we can analyze the two constraints separately and then merge results. For the atomic constraint $altitude \leq 9000$ we obtain the estimator $\hat{X}_{2,1}^T$ having $E\left[\hat{X}_{2,1}^T\right]$=0.45, with variance 0. For the constraint $sin(headFlap \cdot tailFlap) > 0.25$, although its solution space does not fit in a single box, the use of ICP-based stratified sampling does still help in exploiting the geometry of the solution space. For this constraint the estimator $\hat{X}_{2,2}^T$ has $\mathrm{Var}\left[\hat{X}_{2,2}^T\right]$=$8.103406 \cdot 10^{-6}$ and $E\left[\hat{X}_{2,2}^T\right]$=0.417975. Equations (7) and (8) enable `qCORAL` to compose the estimators of sub-formulas of PC_2^T to obtain the estimator $\hat{X}_2^T$, with $E\left[\hat{X}_2^T\right]$=0.188089 and $\mathrm{Var}\left[\hat{X}_2^T\right]$=$1.64094 \cdot 10^{-6}$.

Finally, Equation (5) enables `qCORAL` to obtain the estimator $\hat{X}^T$, with $E\left[\hat{X}^T\right] = E\left[\hat{X}_1^T\right] + E\left[\hat{X}_2^T\right] = 0.738089$, while $\mathrm{Var}\left[\hat{X}^T\right] \leq \mathrm{Var}\left[\hat{X}_1^T\right] + \mathrm{Var}\left[\hat{X}_2^T\right] = 1.64094 \cdot 10^{-6}$. In this example, the exact probability, rounded to the 6th digit is 0.737848.

5. Algorithms and Implementation

This section provides some implementation details of `qCORAL`. The input of `qCORAL` includes a set of disjoint constraints *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 Symbolic PathFinder [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 *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 and 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 *extractRelatedConstraints* goes through all the con-

[2] http://jung.sourceforge.net

Algorithm 2: analyzeConjunction

Input: pc, D, dep
Output: $mean, var$
mean$\leftarrow$ 1; var$\leftarrow$ Nil
for $varSet \in dep$ **do**
  $part \leftarrow$ extractRelatedConstraints($pc, varSet$)
  if !cache.contains($part$) **then**
    $\langle partMean, partVar\rangle \leftarrow$ stratSampling($part, varSet, D$)
    cache.put($part$, $\langle partMean, partVar\rangle$)
  $\langle partMean, partVar\rangle \leftarrow$cache.get($part$)
  $mean \leftarrow mean * partMean$
  if $var = Nil$ **then**
    $var \leftarrow partVar$
  else
    $var \leftarrow mean^2 * partVar + partMean^2 * var + var * partVar$
return $\langle mean, var\rangle$;

juncts in the path constraint pc and projects those containing variables in $varSet$. Note that each $varSet$ contains the variables in a partition induced by the dependency relation as computed by $computeDependencyRelation$. The partial results for each set of independent constructs is computed by means of the procedure $stratSampling$, and then combined with the previous ones. As an optimization, the results obtained for an independent constraint can be safely stored in a cache and reused thanks to the global independence of their estimator with all the others. Conceptually caching makes a trade-off between accuracy and efficiency. It is possible that, when caching is active, the error in local sampling may be amplified as qCORAL samples a partition only once. However, since the subproblems whose results are cached are simpler than the original one (especially in the cases when only a small subset of input variables are related to one another), the accuracy we can obtain is usually higher, especially considering that, thanks to caching, the time budget for each simulation can be increased. Section 6 evaluates the impact of caching in terms of precision and time.

Algorithm 3 computes results for one partition using stratified sampling and Hit-or-Miss Monte Carlo. Argument *varSet* defines the integration variables that needs to be considered to quantify *part* through simulation.The procedure *icp* makes a call to our the ICP solver. The estimate and variance obtained with stratified sampling on each box are cumulatively added to return variables *mean* and *var*. Although the methodology proposed in this paper is general for any usage profile, our current implementation uses uniform profiles only. We plan to evaluate the impact of using more complex probability input distributions. This change affects the sampling generation within the *hitOrMiss* method.

RealPaver. We used RealPaver [14] for Interval Constraint Propagation (ICP). Section 2.2 describes input and output of an ICP solver. Our RealPaver configuration is constrained with the following stop criteria: time budget per query of 2s, a bound on the number of boxes reported per query of 10, and a lower bound on the size of the computed boxes of 3 decimal digits. These parameters are set empirically, based on previous experiences with the tool. All other parameters are fixed. We note that the time budget, in particular, enables one to calibrate the time spent with domain stratification and time spent with simulation.

Algorithm 3: stratSampling

Input: $part, varSet, D$
Output: $mean, var$
$mean \leftarrow 0$; $var \leftarrow 0$
$boxes \leftarrow$ icp($part, varSet, D$)
for $box \in boxes$ **do**
  $\langle boxMean, boxVar\rangle \leftarrow$ hitOrMiss($part, varSet, D$)
  $boxWeight \leftarrow$ size(box)/size(D)
  $mean \leftarrow mean + boxWeight * boxMean$
  $var \leftarrow var + boxWeight^2 * boxVar$
return $\langle mean, var\rangle$;

6. Evaluation

Our evaluation addresses the following research questions:

- **RQ1.** What is the accuracy of qCORAL estimates?
- **RQ2.** How qCORAL compares with the built-in numerical integration routines of Mathematica[3] and with VolComp [30] 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 revolution, 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.

[3] http://www.wolfram.com/mathematica

subject	*Analytical Solution*	10^3 samples		10^4 samples		10^5 samples		10^6 samples	
		estimate	*error* (σ)	*estimate*	*error* (σ)	*estimate*	*error* (σ)	*estimate*	*error* (σ)
Convex Polyhedra									
Tetrahedron	0.512682	0.5151	0.0623	0.5144	0.0210	0.5122	0.0050	0.5128	0.0012
Cube	8.0	8.0	0.0	8.0	0.0	8.0	0.0	8.0	0.0
Icosahedron	2.181695	2.2043	0.1471	2.1948	0.0440	2.1843	0.0133	2.1829	0.0039
Rhombicuboctahedron	14.333333	14.4027	0.3111	14.3286	0.1403	14.3382	0.0416	14.3330	0.0114
Solids of Revolution									
Cone	1.047198	1.0577	0.0509	1.0495	0.0223	1.0462	0.0060	1.0471	0.0019
Conical frustrum	1.8326	1.8483	0.1291	1.8385	0.0451	1.8340	0.0115	1.8326	0.0034
Cylinder	π	3.1470	0.0451	3.1424	0.0189	3.1417	0.0045	3.1415	0.0017
Oblate spheroid	16.755161	16.7723	0.4242	16.7428	0.1479	16.7560	0.0586	16.7550	0.0122
Sphere	$4/3 \cdot \pi$	4.1930	0.1060	4.1857	0.0370	4.1890	0.0146	4.1887	0.0031
Spherical segment	113.557882	113.5982	2.9401	113.4129	0.8739	113.5646	0.2628	113.5618	0.0887
Torus	1.233701	1.2277	0.0467	1.2305	0.0127	1.2327	0.0042	1.2337	0.0013
Intersection									
Two spheres intersection	56.5485	56.7837	1.9705	56.6357	0.6703	56.5498	0.2413	56.5480	0.0600
Cone-cylinder intersection	2.7276	2.7315	0.1401	2.7276	0.0500	2.7289	0.0136	2.7285	0.0036

Table 2. Microbenchmarks.

Assertion	*Num. Paths*	*Num. Ands*	*Num. Ar. Ops.*	`NIntegrate`		`VolComp`		`qCORAL{STRAT,PARTCACHE}` - 30k		
				solution	*time(s)*	*bounds*	*time(s)*	*avg. estimate*	avg. σ	*avg. time(s)*
ARTRIAL										
$points \geq 10$	442	1,484	0 (0)	0.1343	5.05	[0.1343, 0.1343]	3.62	0.1343	0.00e+00	1.03
$points - pointsErr \geq 5$	2,439	1,740	443 (3)	0.0005	89.26	[0.0005, 0.0005]	40.0	0.0005	1.00e-06	1.51
$pointsErr - points \leq 5$	2,260	68,630	19,125 (3)	0.9350	4,179.36	[0.9340, 0.9364]	771.1	0.9352	1.63e-04	4.14
CART										
$count \geq 3$	44	1,209	638 (3)	0.9746	7.26	[0.9390, 1.0000]	32.29	0.9739	1.12e-02	4.18
$count \geq 1$	47	1,296	681 (3)	0.9826	7.66	[0.9470, 1.0000]	33.74	0.9818	1.11e-02	4.39
CORONARY										
$tmp \geq 5$	320	195	62 (3)	0.0006	3.44	[0.0006, 0.0006]	3.93	0.0006	1.90e-06	0.92
$tmp \leq -5$	274	31	8 (3)	0.0001	0.86	[0.0001, 0.0001]	1.99	0.0001	4.29e-07	0.57
EGFR EPI										
$f1 - f \geq 0.1$	45	547	31 (3)	0.1264	1.98	[0.1264, 0.1264]	0.60	0.1262	3.29e-04	1.61
$f - f1 \geq 0.1$	44	422	33 (2)	0.0986	1.69	[0.0986, 0.0986]	0.50	0.0986	4.80e-05	1.42
EGFR EPI (SIMPLE)										
$f1 \leq 4.4 \wedge f \geq 4.6$	13	163	12 (3)	0.5388	0.83	[0.5387, 0.5389]	0.46	0.5389	8.71e-04	0.85
$f1 \geq 4.6 \wedge f \leq 4.4$	14	101	9 (3)	0.3012	0.65	[0.3012, 0.3012]	0.14	0.3012	0.00e+00	0.66
INVPEND										
$pAng <= 1$	1	54	229 (3)	0.0507	1.15	[0.0000, 0.1225]	6.20	0.0515	7.82e-04	0.79
PACK										
$count \geq 5$	1,103	16,414	0 (0)	0.9546	57.44	[0.9546, 0.9546]	16.45	0.9546	0.00e+00	2.18
$count \geq 6$	906	12,080	0 (0)	0.3898	41.76	[0.3898, 0.3898]	12.59	0.3898	0.00e+00	1.94
$count \geq 7$	924	0	0 (0)	0.1428	45.48	[0.1427, 0.1427]	13.75	0.1428	0.00e+00	2.05
$count \geq 10$	821	840	0 (0)	0.0002	4.41	[0.0002, 0.0002]	5.24	0.0002	0.0000	1.25
$totalWeight \geq 6$	954	14,850	6,948 (2)	0.2462	5066.20	[0.2522, 0.2800]	104.8	0.2663	2.72e-05	68.79
$totalWeight \geq 5$	1,030	16,186	7,578 (2)	0.6771	70.16	[0.6369, 0.7155]	60.15	0.6772	1.67e-04	82.98
$totalWeight \geq 4$	1,132	17,972	8,420 (2)	0.9592	54.48	[0.9592, 0.9592]	16.93	0.9592	0.00e+00	92.33
VOL										
$count \geq 20$	24	13,824	882,508 (3)	1.0005	1245.30	[0.0000, 1.0000]	3.76	1.0001	5.18e-03	821.11

Table 3. Comparison of `NIntegrate` (default numerical integration method from Mathematica [1]), `VolComp`, and `qCORAL` with features `STRAT` and `PARTCACHE` enabled. Note that the comparison is restricted to linear constraints and the comparison metrics are not the same.

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 [1]. 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 [4] atrial fibrillation risk calculator (ATRIAL), a steering controller to deal with wind disturbances (CART), the Framingham coronary risk calculator (CORONARY), an eGFR [5] 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 $\sim$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 $f1 - f \geq 0.1$, where the variance is anyway large enough to account for the deviation ($3.29 \cdot 10^{-4}$). 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{}. 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,PARTCACHE} 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

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

[6] http://www.mathworks.com/products/simulink/demos.html

subject	Monte Carlo						qCORAL{STRAT}			qCORAL{STRAT,PARTCACHE}		
	Mathematica			qCORAL{}								
	estimate	σ	*time(s)*	*estimate*	σ	*time(s)*	*estimate*	σ	*time(s)*	*estimate*	σ	*time(s)*
1K samples												
Apollo	0.62521	0.02126	65.44	0.62780	0.01342	18.17	0.62211	0.01262	62.98	0.62461	0.00972	45.04
Conflict	0.49794	0.01308	0.66	0.49927	0.01691	0.44	0.50016	0.00027	1.53	0.50016	0.00027	1.65
Turn Logic	0.72254	0.02250	2.18	0.71913	0.01338	0.64	0.72282	0.01462	1.22	0.72282	0.01462	1.62
10K samples												
Apollo	0.62538	0.00896	221.75	0.62574	0.00458	130.75	0.62610	0.00438	206.89	0.62497	0.00315	97.43
Conflict	0.49992	0.00451	1.10	0.49906	0.00509	0.92	0.50015	0.00010	2.41	0.50015	0.00010	2.31
Turn Logic	0.72029	0.00777	5.97	0.72208	0.00436	2.46	0.72162	0.00490	2.83	0.72162	0.00490	3.35
100K samples												
Apollo	0.62451	0.00278	1723.45	0.62540	0.00160	1263.02	0.62550	0.00143	1583.02	0.62524	0.00089	625.47
Conflict	0.50064	0.00526	5.12	0.50011	0.00156	5.71	0.50017	0.00002	10.14	0.50017	0.00002	8.23
Turn Logic	0.72209	0.00257	41.85	0.72195	0.00116	18.81	0.72238	0.00105	16.77	0.72237	0.00105	18.16

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

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, pow, sin, sqrt, and tan. 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 atan2 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, to not 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.

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

[1] Mathematica NIntegrate, 2013. http://reference.wolfram.com/mathematica/ref/NIntegrate.html.

[2] VolComp, 2013. http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis/.

[3] A. Adje, O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *Verified Software: Theories, Tools and Experiments (VSTTE'13)*, 2013.

[4] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issues and challenges. *Computer*, 39(10):36–43, 2006.

[5] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94:189–201, 2012.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2):10:1–10:38, 2008.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE TSE*, 2(3):215–222, 1976.

[8] E. Davis. Constraint propagation with interval labels. *Artificial Intelligence*, 32(3):281–331, July 1987.

[9] J. A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, and J. Wu. Software for Exact Integration of Polynomials Over Polyhedra. *ACM Communications in Computer Algebra*, 45(3/4):169–172, 2012.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *SCP*, 69(1-3):35–45, 2007.

[11] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *ICSE*, pages 622–631, 2013.

[12] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *ISSTA*, pages 166–176, 2012.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[14] L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32:138–156, 2006.

[15] R. Haggarty. *Fundamentals of mathematical analysis*. Addison-Wesley New York, 1989.

[16] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444. 2006.

[17] F. James. Monte carlo theory and practice. *Reports on Progress in Physics*, 43(9):1145, 1980.

[18] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385—394, July 1976.

[19] D. P. Kroese, T. Taimre, and Z. I. Botev. *Handbook of Monte Carlo Methods*, volume 706. John Wiley & Sons, 2011.

[20] M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *ESEC-FSE*, pages 449–458, 2007.

[21] M. A. Malcolm and R. B. Simpson. Local versus global strategies for adaptive quadrature. *ACM Transactions on Mathematical Software*, 1 (2):129–146, June 1975.

[22] D. Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *POPL*, pages 93–101, 2001.

[23] C. S. Pasareanu, J. Schumann, P. Mehlitz, M. Lowry, G. Karasai, H. Nine, and S. Neema. Model based analysis and test generation for flight software. In *SMC-IT*, pages 83–90, 2009.

[24] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[25] E. Pavese, V. A. Braberman, and S. Uchitel. Automated reliability estimation over partial systematic explorations. In *ICSE*, pages 602–611, 2013.

[26] W. Pestman. *Mathematical Statistics*. De Gruyter, 2009.

[27] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[28] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. ASE, pages 179–180. ACM, 2010.

[29] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer-Verlag New York, Inc., 2005.

[30] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *PLDI*, pages 447–458, 2013.

[31] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *TAP*, pages 134–153, 2008.

[32] P. Zuliani, C. Baier, and E. M. Clarke. Rare-event verification for stochastic hybrid systems. In *HSCC*, pages 217–226, 2012.