Semantic Importance Sampling for Statistical Model Checking*

Jeffery P. Hansen, Lutz Wrage, Sagar Chaki, Dionisio de Niz, and Mark Klein

Carnegie Mellon University, Pittsburgh, PA, USA
{jhansen,lwrage,chaki,dio,mk}@sei.cmu.edu

Abstract. Statistical Model Checking (SMC) is a technique, based on Monte-Carlo simulations, for computing the bounded probability that a specific event occurs during a stochastic system's execution. Estimating the probability of a "rare" event accurately with SMC requires many simulations. To this end, Importance Sampling (IS) is used to reduce the simulation effort. Commonly, IS involves "tilting" the parameters of the original input distribution, which is ineffective if the set of inputs causing the event (i.e., input-event region) is disjoint. In this paper, we propose a technique called Semantic Importance Sampling (SIS) to address this challenge. Using an SMT solver, SIS recursively constructs an abstract indicator function that over-approximates the input-event region, and then uses this abstract indicator function to perform SMC with IS. By using abstraction and SMT solving, SIS thus exposes a new connection between the verification of non-deterministic and stochastic systems. We also propose two optimizations that reduce the SMT solving cost of SIS significantly. Finally, we implement SIS and validate it on several problems. Our results indicate that SIS reduces simulation effort by multiple orders of magnitude even in systems with disjoint input-event regions.

1 Introduction

As systems become more complex, there is a growing demand for efficient and precise techniques to verify correctness of their behavior. In this paper, we target a common probabilistic verification problem – estimating the probability of an event Φ (e.g., some sort of failure) during the execution of a system $\mathcal{M}$ that takes stochastic inputs (e.g., sensor readings, task execution times, etc.) Analytic solutions to this problem (e.g., probabilistic model checking, see Section 2) do not scale to many real-world systems due to complexity. We focus on an alternate approach called Statistical Model Checking (SMC) [16], which relies on Monte-Carlo-based simulations to solve this verification task more scalably.

* This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. DM-0002083.

© Springer-Verlag Berlin Heidelberg 2015
C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 241–255, 2015.
DOI: 10.1007/978-3-662-46681-0_21

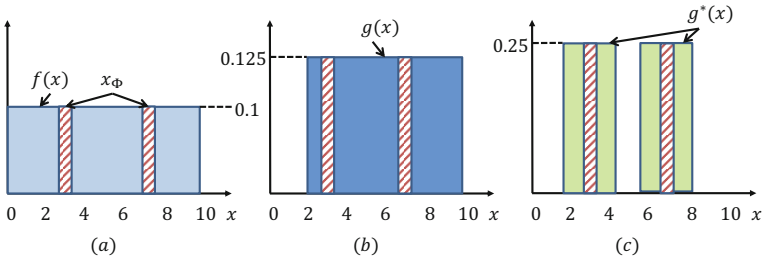

Fig. 1. Example of SIS; f = original input distribution; g = tilted distribution; g^* = distribution produced by SIS

SMC produces two results – the estimate $\hat{p}$ of the probability p of Φ and a measure of precision e. The key challenge in SMC is "simulation explosion" – the number of simulations required to achieve a high e becomes prohibitively large if p is small (i.e., Φ is rare). Importance Sampling [11,14] (IS) has been shown to address this challenge. Suppose the random input x to $\mathcal{M}$ has distribution f. In IS, we first perform SMC under a different input distribution g that makes Φ more likely (i.e., increases p), and then adjust the result back to f.

Traditionally, importance Sampling is implemented by "tilting" the parameters of the input distributions to increase the likelihood of Φ. However, tilting is less effective if the set of inputs that cause Φ, i.e., the input-event region denoted x_Φ, is disjoint. For instance, this happens when analyzing a program where Φ only occurs if the execution follows one of several control-flow paths, each triggered by a distinct input range. Figure 1(a) shows such a case. The actual input distribution f is uniform in the range $[0, 10]$, and $x_\Phi = [2.99, 3.01] \cup [6.99, 7.01]$. Figure 1(b) shows a tilted distribution g uniform in the range $[2, 10]$. While g makes Φ more likely than f, it still assigns positive weight to large parts – e.g., $(3.01, 6.99)$ – of the input space that do not belong to x_Φ.

In this paper, we address this challenge, and make three specific contributions. First, we develop a new technique to construct more precise input distributions for IS – such as g^* shown in Figure 1(c) – even when the input-event region is disjoint. This technique, which we call Semantic Importance Sampling (SIS), takes as input a description of $\mathcal{M}$ and f, and recursively computes a precise "over-approximation" of x_Φ in the form of an abstract indicator function (AIF). In each step of the recursion, SIS constructs a verification condition using $\mathcal{M}$ and f and checks its satisfiability with an SMT solver to eliminate parts of the input space that are not in x_Φ. The algorithm outputs an AIF represented by a set of "input cubes", i.e., a disjunction of intervals [7] over the input variables of $\mathcal{M}$. Subsequently, SIS uses the AIF to construct a precise input distribution, and perform SMC with IS. By using the semantics of $\mathcal{M}$, SIS successfully applies concepts and techniques used widely in the verification of non-deterministic systems (such as abstraction, SMT solving, and verification conditions) to the analysis of stochastic systems, building new bridges between the two disciplines.

The most expensive component of SIS are the calls to the SMT solver. Our second contribution is two optimizations to SIS that reduce the number of SMT calls while maintaining correctness. Finally, we implement SIS in a tool called OSMOSIS and use it to verify a number of stochastic systems with rare events. Our results indicate that SIS reduces the number of simulations significantly, in some cases by a factor of over 600, and verification time by an order of magnitude or more. Furthermore, our optimizations reduce both the number of SMT calls and overall SMT solving time, typically by a factor of 2. All our tools and examples are available at `andrew.cmu.edu/~schaki/misc/osmosis.zip`.

2 Related Work

Probabilistic model checking [15] (PMC) is an automated, algorithmic approach for computing numerical properties of stochastic systems. In PMC, the system is modeled as a finite state probabilistic automaton, e.g., a discrete time Markov chain (DTMC), a continuous time Markov chain (CTMC), or a Markov decision process (MDP) which is exhaustively explored in the analysis. The property is expressed as formula in a temporal logic, e.g., probabilistic Computation Tree Logic (PCTL) [8]. Verification consists of exhaustive exploration of the state-space to construct equations which are then solved numerically. In contrast, we follow the SMC approach, which is based on Monte-Carlo simulations. An excellent comparison between PMC and SMC is provided by Younes et al. [17].

SMC [16,2] has been applied to a wide variety of systems including stochastic hybrid automata [4], and real time systems [5]. Methods proposed to increase SMC performance include importance splitting [10], and importance sampling [14]. Importance sampling, upon which our approach is based, has been long known in the statistics literature[11] and has recently come to the attention of the SMC community[3]. Approaches proposed to finding the importance sampling bias function include Cross-Entropy Method[2,9] and Coupling [1].

Borges et al., [13] proposes a technique for estimating failure probabilities in software based on stratified sampling. Their technique differs from ours in that they partition the input space based on path conditions in the model, whereas we use an approach that modifies the input distribution.

Luckow et al. [12] have developed techniques for exact and approximate analysis of stochastic systems with non-determinism. They use symbolic execution and learning to iteratively construct schedulers under which worst-case (or best-case) behavior of the system is observed. This approach can be seen as an extension of statistical model checking to concurrent systems. They do not use importance sampling, and could benefit from our techniques.

3 Background

Consider a system $\mathcal{M}$ with finite vector of random inputs x. Assume that $\mathcal{M}$ is deterministic, i.e., its behavior is fixed for a fixed value of x. The SMC problem is to estimate the probability that $\mathcal{M}$ satisfies a property Φ, denoted $\mathcal{M} \models \Phi$,

given a joint probability distribution f on x, i.e., to estimate $p = Pr[\mathcal{M} \models \Phi]$. We assume that whether $\mathcal{M} \models \Phi$ under input x can be determined by simulating $\mathcal{M}$ for finite time. Specifically, we assume that $\mathcal{M}$ is a program that terminates under all inputs, and $\mathcal{M} \models \Phi$ under input x iff the execution of $\mathcal{M}$ under input x violates an assertion (representing a desired safety property) in $\mathcal{M}$.

Let us write $x \sim f$ to mean x is distributed by f. SMC involves a series of Bernoulli trials, modeling each trial as a Bernoulli random variable having value 1 with probability p, and 0 with probability $1 - p$. For each trial i, a random vector $x_i \sim f$ is generated, and the system $\mathcal{M}$ is simulated with input x_i to generate a trace σ_i. The trial's outcome is 1 if Φ holds on σ_i, and 0 otherwise.

Define an indicator function $I_{\mathcal{M}\models\Phi} : x \rightarrow \{0, 1\}$ that returns 1 if $\mathcal{M} \models \Phi$ under input x, and 0 otherwise. Then, when $x \sim f$, the probability that $\mathcal{M} \models \Phi$ holds will be $p = E[I_{\mathcal{M}\models\Phi}(x)] = \int I_{\mathcal{M}\models\Phi}(x) f(x) dx$ which can be estimated as:

$$\hat{p} = \sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) \tag{1}$$

where N is the number of trials and $x_i \sim f$. We will refer to this estimator as the Crude Monte-Carlo (CMC) estimator. The precision of $\hat{p}$ is quantified by its "relative error" $RE(\hat{p}) = \frac{\sqrt{Var(\hat{p})}}{E[\hat{p}]}$ where $Var(\hat{p})$ is the variance of the estimator. It is known[2] that for Bernoulli trials, relative error is related to the number of trials N and the probability of the event p as:

$$RE(\hat{p}) = \sqrt{\frac{1-p}{pN}} \approx \frac{1}{\sqrt{pN}} \qquad N = \frac{1-p}{pRE^2(\hat{p})} \approx \frac{1}{pRE^2(\hat{p})} \tag{2}$$

Importance Sampling. From (2) we see that the number of simulations needed to achieve a fixed precision with SMC increases rapidly as the target event becomes rarer. Importance Sampling [14] (IS) has been applied [2] to address this challenge effectively by reducing $Var(\hat{p})$. The key idea behind IS is to first simulate $\mathcal{M}$ under a different input distribution g to reduce the variance of the estimator, and then mathematically adjust the result back to the original distribution f as:

$$p = \int I_{\mathcal{M}\models\Phi}(x) \frac{f(x)}{g(x)} g(x) dx = \int I_{\mathcal{M}\models\Phi}(x) W(x) g(x) dx \tag{3}$$

where $W : x \rightarrow \frac{f(x)}{g(x)}$ is a *weight function*. The estimator for this form is:

$$\hat{p} = \sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) W(x_i) \tag{4}$$

where the $x_i \sim g$. The biggest challenge in applying IS effectively is choosing a "good" g that will reduce $Var(\hat{p})$. Typically this is done by "tilting" f by changing its distribution parameters (mean, variance etc.) However, as discussed, tilting is not effective if Φ is disjoint in the input space. In effect, SIS constructs a good g even in such cases. We describe SIS in detail in the next section.

4 Semantic Importance Sampling

To explain SIS, we begin with a known result [2] that there always exists an optimal IS distribution:

$$g^{\diamond}(x) = \frac{I_{\mathcal{M}\models\Phi}(x)f(x)}{p} \tag{5}$$

for which $Var(\hat{p}) = 0$, i.e., if IS is done with $g = g^{\diamond}$, then a single sample is sufficient to compute $\hat{p}$. However, there are two challenges to using $g^{\diamond}$ for IS: (i) $g^{\diamond}$ depends on p, the answer we are actually looking for; and (ii) $g^{\diamond}$ also depends on the indicator function $I_{\mathcal{M}\models\Phi}$, but since this function represents $\mathcal{M} \models \Phi$ itself, it may be too complex to represent analytically.

The key insight behind SIS is to construct an *abstract indicator function* (AIF) $I^*_{\mathcal{M}\models\Phi} : x \to \{0,1\}$ such that: (i) $\forall x\ I_{\mathcal{M}\models\Phi}(x) = 1 \Rightarrow I^*_{\mathcal{M}\models\Phi}(x) = 1$; and (ii) $I^*_{\mathcal{M}\models\Phi}$ is simple enough to represent analytically. Note that $\{x \mid I^*_{\mathcal{M}\models\Phi}(x) = 1\}$ is an over-approximation of the set of inputs under which $\mathcal{M} \models \Phi$. This AIF induces the following IS distribution and weight function:

$$g^*(x) = \frac{I^*_{\mathcal{M}\models\Phi}(x)f(x)}{p^*} \tag{6}$$

$$W^*(x) = \frac{f(x)}{g^*(x)} = \frac{f(x)p^*}{I^*_{\mathcal{M}\models\Phi}(x)f(x)} = \frac{p^*}{I^*_{\mathcal{M}\models\Phi}(x)} \tag{7}$$

where $p^* = E[I^*_{\mathcal{M}\models\Phi}(x)]$ is the probability that for an input $x \sim f$, $I^*_{\mathcal{M}\models\Phi}(x) = 1$. Note that as the function $I^*_{\mathcal{M}\models\Phi}$ approaches $I_{\mathcal{M}\models\Phi}$, g^* also approaches $g^{\diamond}$. In the limit, $I^*_{\mathcal{M}\models\Phi} = I_{\mathcal{M}\models\Phi}$ implies $g^* = g^{\diamond}$.

Probability Estimation and Relative Error in SIS. Substituting $W^*(x)$ from (7) into (4), we get the SIS estimator for $p = E[I_{\mathcal{M}\models\Phi}(x)]$ given $x \sim f$ as:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i)W^*(x_i) = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i)\frac{p^*}{I^*_{\mathcal{M}\models\Phi}(x_i)} \tag{8}$$

with $x_i \sim g^*$ used in this importance sampled estimator. Note from (6) that $I^*_{\mathcal{M}\models\Phi}(x_i)$ is always 1 when $x_i \sim g^*$, thus this term can be dropped from the summation. Also, since p^* is a constant (8) simplifies to:

$$\hat{p} = \frac{p^*}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) \tag{9}$$

This can be split into a raw part and a scalar part as $\hat{p} = p^* \times \hat{p}_{\text{raw}}$, where:

$$\hat{p}_{\text{raw}} = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) \tag{10}$$

Since $\hat{p}_{\text{raw}}$ is an unweighted average of Bernoulli random variables, its relative error can be estimated [2] as:

$$RE(\hat{p}_{\text{raw}}) \approx \frac{1}{\sqrt{p_{\text{raw}}N}} \tag{11}$$

Furthermore, since $\hat{p} = p^* \times \hat{p}_{\text{raw}}$, and p^* is a constant, the relative error for $\hat{p}$ is the same as the relative error of $\hat{p}_{\text{raw}}$, i.e., $RE(\hat{p}) = RE(\hat{p}_{\text{raw}})$.

4.1 The SIS Algorithm

The SIS algorithm involves the following steps:

1. Recursively construct the AIF $I^*_{\mathcal{M}\models\Phi}$.
2. Calculate p^*.
3. Use SMC to estimate $\hat{p}_{\text{raw}}$ with desired $RE(\hat{p}) = RE(\hat{p}_{\text{raw}})$, using $I^*_{\mathcal{M}\models\Phi}$ to draw random inputs from g^*. Output $\hat{p} = p^* \times \hat{p}_{\text{raw}}$.

The core of SIS is Step 1, the generation of the AIF. We describe this in the following sections by first discussing our representation of the AIF, then describing the recursive algorithm.

AIF as a Cube Set. We assume that the input x to $\mathcal{M}$ is a vector of M independent[1], but not necessarily identically distributed random variables. For each dimension x_j in x, let F_j be the Cumulative Distribution Function (CDF), F_j^{-1} be the inverse CDF (or quantile function), and $u_j = F_j(x_j)$ be the quantile domain variable. Now let ξ be an M-dimensional axis-aligned input domain hypercube defining an interval $[l_j, h_j]$ on each input variable x_j for $1 \leq j \leq M$. We also define the quantile domain hypercube c defined by the ranges $[F_j(l_j), F_j(h_j)]$ for each dimension. We use the notation $c = F(\xi)$ and $\xi = F^{-1}(c)$ to transform cubes between the input and quantile domains. We will use the terms *input cube* and *quantile cube* to refer to cubes in the input and quantile domains, respectively. When the term *cube* is used without qualification we will assume quantile cubes. We can now represent the AIF in terms of a quantile cube set C^* as:

$$I^*_{\mathcal{M}\models\Phi}(x) = \begin{cases} 1 \text{ if } \exists c \in C^* \mid F(x) \in c \\ 0 \text{ otherwise} \end{cases} \tag{12}$$

where $(\forall x\ I_{\mathcal{M}\models\Phi}(x) = 1) \Rightarrow (\exists c \in C^* | F(x) \in c)$ (i.e., all inputs where $\mathcal{M} \models \Phi$ holds are covered by some cube in C^*).

Cube Splitting. Let ξ_U be the input cube defining the support of the input distribution function f. The corresponding quantile domain cube $c_U = F(\xi_U)$ will have a range of $[0, 1]$ on each dimension. We call this the level-0 cube. We write c/j to mean the cube formed by splitting the interval on u_j in c in half, and retaining only the upper half. Similarly, $c/\overline{j}$ is the result of a similar operation where the lower half of the interval is retained. Note that we can split on the same variable multiple times. A level-k cube is the result of k splits on the level-0 cube. For example if c_U is the level-0 cube, then $c_U/1/\overline{1}$ is the level-2 cube in which the interval for u_1 is $[0.5, 0.75]$. After each split, the probability that an input drawn from f falls in the result is halved. Thus, the probability of an input drawn from f falling in a level-k cube is $\frac{1}{2^k}$.

[1] Non-independent random inputs y are replaced by a function $h(x)$ of independent random variables x, which is folded into $I_{\mathcal{M}\models\Phi}(y)$ to yield $I_{\mathcal{M}\models\Phi}(h(x))$.

```
(1)     CubeSet aifGen(SMT φ,Cube c)
(2)     {
(3)        if (Solve(φ, F^{-1}(c)) == UNSAT) return ∅;
(4)        if (level(c) == L_max) return {c};
(5)        int j = (level(c)/G)  % M;
(6)        Cube c_0 = c/j; Cube c_1 = c/j;
(7)        return aifGen(φ, c_0) ∪ aifGen(φ, c_1);
(8)     }
```

Fig. 2. Basic AIF Generation Algorithm; G=variable grouping factor, M=number of inputs, $L_{\max}$=recursion depth limit, `Solve` = satisfiability check via SMT solver

Recursive AIF Construction. Generation of the AIF $I^*_{\mathcal{M}\models\Phi}$ is performed recursively through the hierarchical use of an SMT solver. The basic algorithm `aifGen` is shown in Figure 2. It takes as input the SMT representation φ of the indicator function $I_{\mathcal{M}\models\Phi}(x)$, and the input cube c over which to generate an abstraction. It is assumed that φ is constructed so as to be SAT for inputs x iff $I_{\mathcal{M}\models\Phi}(x) = 1$. Constant $L_{\max}$ is the maximum recursion depth. `aifGen` returns the subset of level-$L_{\max}$ cubes in C^* within cube c. C^* representing the AIF as defined in (12) can then be determined by calling `aifGen`, and passing the level-0 cube c_U as c.

The algorithm works as follows. At Line 3, the SMT solver is applied to the model φ over the cube $\xi = F^{-1}(c)$. The cube is applied to the model by modifying the assertions in the model corresponding to the intervals on the input variables. The SMT solver can return SAT, UNSAT or UNKNOWN (e.g., if it times out). If the result is UNSAT, then $\mathcal{M} \models \Phi$ does not hold in the input space described by c, and so it returns the empty set. If the result is SAT or UNKNOWN, we continue with the rest of the algorithm. While an UNKNOWN result will reduce the efficiency of the algorithm, the result will still be sound.

At Line 4, the level of the current cube c is checked against the specified maximum recursion depth $L_{\max}$. If we are at that maximum recursion depth, we simply return the set containing just the cube c.

At Line 5, we choose an input variable index on which to split the current cube. In our current implementation, we simply cycle through the variables round-robin by using the current level modulo the total number of input variables M. Integer division by a variable grouping factor G allows us to choose the same variable G levels in a row before moving to the next variable. It is possible that other methods of choosing the splitting order may lead to more efficient abstractions, however we have not yet explored this area.

At Lines 6-7, we split the cube c around the selected variable u_j forming the cubes c_0, and c_1 for the lower and upper half of the CDF interval on variable u_j in c. We then recursively call the generation algorithm on those two sub-cubes and return the union of the cube sets returned by each call.

Calculation of p^*. Recall that $p^* = E[I^*_{\mathcal{M}\models\Phi}(x)]$ given $x \sim f$. Since: (i) all cubes in the set C^* returned by `aifGen` are level-$L_{\max}$, (ii) they are non-overlapping, (iii) there are $2^{L_{\max}}$ level-$L_{\max}$ cubes, and (iv) each cube covers equal probability in f, then p^* can be calculated from the ratio of the number of cubes in C^* to the total number of level-$L_{\max}$ cubes as:

$$p^* = \frac{|C^*|}{2^{L_{\max}}} \tag{13}$$

4.2 Optimized AIF Generation

The most expensive component of `aifGen` are the calls to `Solve`. We now present two optimizations that can reduce the number of calls.

Optimization 1: Skip on UNSAT. Consider the algorithm in Figure 2. Notice that at the point where we split the cube at Line 6, we already know that cube c is not UNSAT. The means that if one of the child cubes c_0 or c_1 is UNSAT, the other one must be SAT[2]. To take advantage of this, we modify the algorithm to take an additional boolean argument `assumeSAT` indicating we should skip the call to `Solve` and assume it returns SAT when `assumeSAT` is true. Then we make the first recursive call on c_0 with `assumeSAT` set to false. If this call returns the empty set, then the result for that half was UNSAT, and we pass true for `assumeSAT` when making the recursive call on c_1, otherwise we make the recursive call with `assumeSAT` set to false and execute `Solve` as normal.

Optimization 2: Counter-Example Reuse. A second optimization is possible by making use of the counter-example returned by `Solve` when the result is SAT. In this case, we assume that `Solve` returns, as counter-example, a cube ξ_d containing a satisfying solution. We convert ξ_d to a quantile cube $c_d = F(\xi_d)$. If c_d is completely contained by one of the child cubes in the recursive call, we can skip the call to `Solve` for that call. We require c_d to be completely contained since the counter-example cube ξ_d returned by `Solve` is a cube in which there *exists* a solution to the SMT formula, but *not all* points in the cube are necessarily a solution. In most cases c_d will be contained by one or the other of the child cubes in the recursive calls, but it is possible that c_d could fall on an edge and thus not be applicable to either recursive call. In this case, it is still possible that Optimization 1 can apply. We assume that `Solve` will return the empty cube $\emptyset$ when the result is UNKNOWN which will suppress use of this optimization for the child invocations. It can be shown that if there are k calls to `Solve` without this optimization, that there will be $\lfloor \frac{k}{2} \rfloor + 1$ with this optimization as long as: (i) `Solve` never returns UNKNOWN, and (ii) the counter-example c_d returned by `Solve` always falls in one of the two sub-cubes. This sets an upper bound of 1/2 on the amount by which calls to `Solve` can be reduced.

Optimized AIF Generation Algorithm. Figure 3 shows the fully optimized abstract indicator function incorporating both of the optimizations discussed above. Line 3 tests for conditions that allow us to skip the SMT check. In the case that we are skipping a check, we can pass the existing c_d to the child recursive calls since it may apply to one of those calls as well. When doing the SMT check with `Solve` at Line 4, we include an additional return parameter ξ_d in which the counter-example cube is returned. We assume that the empty cube $\emptyset$ is returned if the result is not SAT. At Line 5 we convert the input cube ξ_d

[2] It could be UNKNOWN if result from cube c is UNKNOWN, but without loss of soundness we treat an UNKNOWN as SAT for the purpose of this optimization.

```
(1)   CubeSet aifGen(SMT I,Cube c,boolean assumeSAT,Cube c_d)
(2)   {
(3)     if (!assumeSAT && c_d != ∅ && !(c_d ⊆ c)) {
(4)       if (Solve(I, F^{-1}(c), &xi_d) == UNSAT) return ∅;
(5)       c_d = F(ξ_d);
(6)     }
(7)     if (level(c) == L_max) return {c};
(8)     int j = (level(c)/G) % M;
(9)     Cube c_0 = c/j; Cube c_1 = c/j;
(10)    CubeSet s_0 = aifGen(I, c_0, false, c_d);
(11)    CubeSet s_1 = ∅;
(12)    if (s_0 == ∅) s_1 = aifGen(I, c_1, true, c_d);
(13)    else s_1 = aifGen(I, c_1, false, c_d);
(14)    return s_0 ∪ s_1;
(15)  }
```

Fig. 3. Optimized Abstract Indicator Function (AIF) Generation Algorithm; G=variable grouping factor, M=number of input, $L_{\max}$=recursion depth limit

to a quantile cube c_d. Lines 12 to 13 implement Optimization 1. If $s_0 = \emptyset$, then the result of the test for c_0 was UNSAT and we can assume that the test for c_1 will be SAT.

4.3 Statistical Model Checking

After generating the AIF $I^*_{\mathcal{M}\models\Phi}$, and computing p^* with (13), the last step in SIS is the actual SMC. As previously mentioned, we draw samples from the distribution g^* as defined in (6), then use (10) to estimate the raw probability $\hat{p}_{raw}$ and scale this by p^*.

Random Input Generation. To generate a random input from g^*, we recognize that this is the equivalent of generating an input x from f and accepting only those for which $I^*_{\mathcal{M}\models\Phi}(x) = 1$. We do this by first randomly selecting a cube c from C^* with uniform probability since each cube has equal probability of containing a sample drawn from f. We then choose a uniform vector $u \in c$ and use the inverse CDF to generate the input vector as $x = F^{-1}(u)$.

No. Of Samples. From (2), the number of samples N^* needed to estimate $\hat{p}_{raw}$ is:

$$N^* = \frac{1 - p_{raw}}{p_{raw} RE^2(\hat{p}_{raw})} = \frac{1 - p/p^*}{p/p^* RE^2(\hat{p}_{raw})} \tag{14}$$

From (9), we know that $RE(\hat{p}) = RE(\hat{p}_{raw})$. Assuming small p and $p^* \gg p$, the speedup due to SIS can be estimated as:

$$\frac{N}{N^*} = \frac{\frac{1-p}{pRE^2(\hat{p})}}{\frac{1-p/p^*}{p/p^* RE^2(\hat{p}_{raw})}} = \frac{1-p}{p^* - p} \approx \frac{1}{p^*} \tag{15}$$

5 Osmosis

We implemented SIS in a tool called OSMOSIS. The input to OSMOSIS is a description of $\mathcal{M}$ in an annotated version of C, with the target property Φ defined as

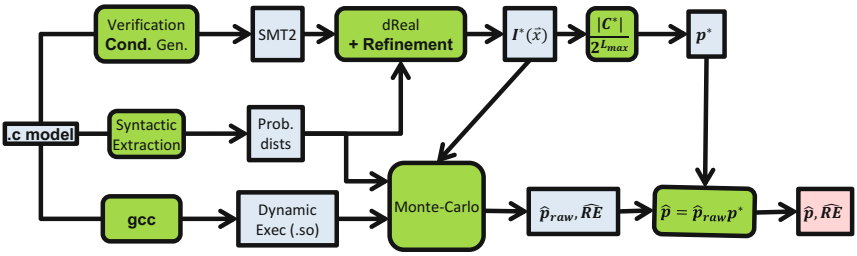

Fig. 4. Architecture of OSMOSIS Tool

`ASSERT()` statements. OSMOSIS calculates the probability of an `ASSERT()` failure via SIS, using dReal[6] as the back-end SMT solver.

Osmosis Architecture. Figure 4 shows the architecture of OSMOSIS. The input model is processed by: (i) `gcc` to generate a dynamic executable; (ii) a syntactic extractor which looks for `//@dist` declarations to determine the input space and distributions; and (iii) a verification condition generator that generates an SMT formula corresponding to the C model. Then `aifGen` (from Figure 2 or Figure 3) is used to build the AIF $I^*_{\mathcal{M}\models\Phi}$. This AIF is used to calculate p^*, and in conjunction with the dynamically loaded executable for $\mathcal{M}$ to estimate $\hat{p}_{\text{raw}}$ and $RE(\hat{p}_{\text{raw}})$. Finally, $\hat{p}$ is calculated using p^* and $\hat{p}_{\text{raw}}$.

Osmosis Input Format. Figure 5(a) shows an example OSMOSIS input model. The annotations at Lines 4 and 5 indicate the inputs to the model. Line 4 defines a random input named "a" with a uniform distribution between 0 and 5. Line 5 defines a random input named "b" with a normal distribution with mean 3, standard deviation 1 which has been censored to be between 0 and 5. Where appropriate, we refer to the model input collectively as the vector x.

There are two special functions/macros in OSMOSIS models: (i) `ASSERT()` defines a condition that is expected to be true; and (ii) `INPUT_D()` accesses a random input declared in an annotation. The suffix `_D` on `INPUT_D()` indicates the return type of `double`. In Figure 5(a), Lines 8 and 9 access inputs "a" and "b" and place them in C variables also named "a" and "b". Some computations are performed on lines 10 and 11, then finally an assertion is made on Line 13. The `#include` on Line 1, allows the model include the special OSMOSIS functions to be compiled by a standard compiler such as `gcc` for use in the SMC phase.

SMT Generation. In order to implement `Solve`, OSMOSIS translates the C model into a verification condition represented as an SMT formula φ, which is in essence, a representation of the indicator function $I_{\mathcal{M}\models\Phi}$, i.e., any input value x satisfies φ iff $I_{\mathcal{M}\models\Phi}(x) = 1$. In constructing φ, stochastic inputs defined by the `//@dist` annotations in the C model use the same variable name as the declaration. The model is also converted to single-static-assignment form so that each local variable is assigned once. A generation number is appended to each variable name and is incremented for each assignment to that variable.

Conditional (`if`) statements are translated by generating a variable for the condition, then translating both branches as consequences of implications of the

```
(1)  #include "osmosis_model.h"
(2)
(3)
(4)  //@dist a=uniform(min=0,max=5)
(5)  //@dist b=normal(mean=3,std=1,
                        min=0,max=5)
(6)  void model()
(7)  {
(8)    double a = INPUT_D("a");
(9)    double b = INPUT_D("b");
(10)   double c = a + b;
(11)   double d = (a - b)/2.0;
(12)
(13)   ASSERT(sin(c)*cos(d) <= 0.999);
(14) }
```

(a)

```
(1)  (set-logic QF_NRA)
(2)  (declare-fun a () Real)
(3)  (declare-fun b () Real)
(4)  (declare-fun a_1 () Real)
(5)  (declare-fun b_1 () Real)
(6)  (declare-fun c_1 () Real)
(7)  (declare-fun d_1 () Real)
(8)  (assert (>= a 0))
(9)  (assert (<= a 5))
(10) (assert (>= b 0))
(11) (assert (<= b 5))
(12) (assert (= a_1 a))
(13) (assert (= b_1 b))
(14) (assert (= c_1 (+ a_1 b_1)))
(15) (assert (= d_1 (/ (- a_1 b_1) 2)))
(16) (assert (not (<= (* (sin c_1)
                 (cos d_1)) 0.999)))
(17) (check-sat)
(18) (exit)
```

(b)

```
if (a > b)
   a = cos(a*b);
```

(c)

```
(assert (= _C1 (> a_1 b_1)))
(assert (or (not _C1) (= a_2 (cos (* a_1 b_1)))))
(assert (or _C1 (= a_2 a_1)))
```

(d)

Fig. 5. (a) OSMOSIS Input Example; (b) SMT for OSMOSIS Input Example; (c) a conditional statement; and (d) its translation to SMT

condition, or the compliment of the condition. If there are differing numbers of assignments to a variable in the branches, then an additional assertion is added to reconcile the generation numbers of the variables. For example, the C statement in Figure 5(c) generates the SMT assertions in Figure 5(d). Loop (`while` and `for`) statements are unrolled and must include an annotation to indicate the maximum loop count. Note that the construction of φ is effective and linear in the size of the model.

Finally, `ASSERT()` conditions are negated since we are interested in testing if there are any inputs that can result in an assertion failure. All `ASSERT()` statements are merged into a single SMT assertion comprised of a disjunction of the compliments of the expressions in the C input model.

Figure 5(b) shows the φ generated from the $\mathcal{M}$ given in Figure 5(a). Line 8 through 11 define the intervals in the stochastic inputs. Lines 12 and 13 are the assignments from the stochastic inputs to the local C variables from Lines 8 and 9 of the input model. Lines 14 and 15 correspond to the local variables assignments in Lines 10 and 11 of the C model. Finally, Line 16 is derived from the `ASSERT()` statement on Line 13 of the C model.

Monte-Carlo Simulation. The final step of OSMOSIS is Monte-Carlo simulation to estimate $\hat{p}_{\text{raw}}$ using (10). Each Bernoulli trial in this simulation is conducted by directly executing the dynamically loadable executable of the model. The model source file is compiled by `gcc`, dynamically loaded, then repeatedly called for each trial. Before each execution a random vector $x \sim g^*$ is generated as described above and used to initialize a global array. A global flag variable indicating success/failure is also cleared. The function `INPUT_D()` indexes and returns a value from the input array. The `ASSERT()` statement tests the condition,

Table 1. AIF Generation Results; In=number of inputs; $L_{\max}$=recursion depth limit; G=variable grouping factor, Time=generation time in seconds; none, 1, 2 and 1+2 indicate which optimizations were used

					dReal Calls				Time	
Name	In	$L_{\max}/G$	p^*	$1/p^*$	none	1	2	1+2	none	1+2
simple	2	10/1	5.859×10^{-3}	169	49	38	26	26	0.15	0.1
		12/1	2.197×10^{-3}	455	73	57	40	40	0.21	0.1
hockey	2	10/1	3.516×10^{-2}	28.4	255	213	142	137	315	228
		12/1	1.148×10^{-2}	87.1	391	328	214	211	364	255
backoff	6	10/4	1.797×10^{-1}	5.6	479	451	240	240	33	14
		12/4	1.797×10^{-1}	5.6	1583	1551	792	792	61	28
bounce	2	10/1	2.997×10^{-2}	33	117	86	59	59	91	53
		12/1	1.221×10^{-2}	81	221	163	111	111	150	84

and if the condition fails it sets the global flag to `true` and returns. Success or failure of the trial is recorded based on the value of the flag variable. Trials resulting in an `ASSERT()` fail correspond to inputs x_i where $I_{\mathcal{M}\models\Phi}(x_i) = 1$, and those where the `ASSERT()` does not fail correspond to inputs where $I_{\mathcal{M}\models\Phi}(x_i) = 0$. Trials are conducted until a target relative error is met.

6 Results

To evaluate our technique, we tried OSMOSIS on the following problems:

Simple. The example problem from Figure 5a.

Hockey. An air hockey puck is given a random impulse from a random direction. We test if it stops on a target after zero or more bounces.

Backoff. An exponential backoff problem in which two senders attempt up to 3 communications. Failure occurs if transmission for either exceeds a deadline.

Bounce. A ball is launched at a random initial angle and velocity. We test if it falls in a small hole after potentially bouncing a number of times.

Each of these problems has the characteristic that the failure region is disjoint in the input space. For example, in the hockey problem there are multiple paths by which the puck can reach the target. All experiments were performed under Linux Ubuntu 12.04 on a 2.2GHz Intel Core i7 machine with 16 Gb of RAM. We used a 60 second timeout for each call to dReal (after which it returns UNKNOWN). However, we experienced no timeouts on any of our test problems.

Table 1 shows the results for AIF generation. For each example, we adjust the recursion depth limit and the variable grouping factor (number of successive times each input is split while recursing). We used a larger G for the "backoff" example because we observed that a higher G improves performance for models with many inputs. Recall from (15) that $1/p^*$ is an estimate for the expected speedup $\frac{N}{N^*}$ of SIS versus Crude Monte-Carlo (CMC). Note that while we use $L_{\max}$ to limit the recursion depth while generating the AIF, a breadth-first

implementation of `aifGen` could potentially use p^*, terminating when we have achieved a sufficient gain, or when there is insufficient improvement from one level to the next. The four columns under "dReal Calls" show the number of calls that were made to dReal using no optimization, using Optimization 1 only, using Optimization 2 only and using both optimizations (see Section 4.2).

We see that both optimizations are effective at reducing the number of calls, but that Optimization 2 performs better, reducing the number of calls as well as total SMT solving time by half in most cases. Also, while there is some benefit to using both optimizations together, the additional advantage is relatively small. This is because when using both optimizations together, Optimization 1 can only be applied when the counter-example employed by Optimization 2 falls on a cube boundary, or when analysis of a parent cube timed out and is UNKNOWN.

Finally, the "Time" column shows the time to generate $I^*_{\mathcal{M}\models\Phi}$ in seconds. Times using no optimization (none), and using both optimizations (1+2) are shown to demonstrate the impact of the optimization techniques. Note that in our current implementation, we do not parallelize the calls to dReal, which could lead to additional gains.

Table 2. SMC Results; $RE = RE(\hat{p})$=target relative error; G=grouping factor

Name	RE	$L_{\max}/G$	$\hat{p}$	N	N/N^*	Time (sec.) SMC	Time (sec.) total
simple	0.01	CMC	5.95×10^{-4}	1.68×10^{7}	–	6	6
		10/1	5.89×10^{-4}	8.95×10^{4}	187	<0.1	0.1
		12/1	6.03×10^{-4}	2.64×10^{4}	636	<0.1	0.1
	0.001	CMC	5.910×10^{-4}	1.69×10^{9}	–	580	580
		10/1	5.910×10^{-4}	8.92×10^{6}	189	4	4.1
		12/1	5.910×10^{-4}	2.72×10^{6}	304	1	1.1
hockey	0.01	CMC	6.18×10^{-4}	1.58×10^{7}	–	6.8	6.8
		10/1	6.18×10^{-4}	5.59×10^{5}	28.3	0.3	228.3
		12/1	6.22×10^{-4}	1.74×10^{5}	90.1	0.1	255.1
	0.001	CMC	6.215×10^{-4}	1.61×10^{9}	–	687	687
		10/1	6.214×10^{-4}	5.56×10^{7}	29.0	25	253
		12/1	6.212×10^{-4}	1.74×10^{7}	92.5	8	263
backoff	0.01	CMC	1.21×10^{-4}	8.24×10^{7}	–	25	25
		10/4	1.20×10^{-4}	1.50×10^{7}	5.5	6	20
		12/4	1.21×10^{-4}	1.50×10^{7}	5.5	6	34
	0.001	CMC	1.193×10^{-4}	8.38×10^{9}	–	2,593	2,593
		10/4	1.190×10^{-4}	1.51×10^{9}	5.5	553	567
		12/4	1.194×10^{-4}	1.50×10^{9}	5.6	543	571
bounce	0.01	CMC	2.96×10^{-5}	3.337×10^{8}	–	133	133
		10/4	3.00×10^{-5}	8.464×10^{6}	39	4.1	57.1
		12/4	2.97×10^{-5}	4.104×10^{6}	81	2.0	86.1
	0.001	CMC	2.989×10^{-5}	3.345×10^{10}	–	13,619	13,619
		10/4	2.993×10^{-5}	8.474×10^{8}	39.5	432	485
		12/4	2.994×10^{-5}	4.068×10^{8}	82	209	293

Table 2 shows the results from the SMC phase of OSMOSIS. For each sample problem, we show the results for target relative errors (RE) of 0.01 and 0.001. At each target RE, we compare CMC with SIS using two different recursion depth limits as shown in the $L_{\max}/G$ column. The probability estimate for each experiment is shown in the $\hat{p}$ column. We see that the estimates for CMC and SIS are very close for each problem, and that as expected the agreement for those at a relative error of 0.001 are closest.

The column labeled N shows the number of samples needed to achieve the target relative error for each experiment, and the column labeled N/N^* shows the improvement of SIS over CMC. We can see improvements ranging from a factor of 5 to a factor of over 600. When we compare the measured N/N^* to the values predicted by $1/p^*$ in Table 1, we see good agreement. For example, in the "hockey" problem with a recursion depth of 10, we got 28.4 as the predicted improvement, compared to measured improvements of 28.3 for a target RE of 0.01 and 29.0 for a target RE of 0.001. Note our predictor is based on the assumption that $p^* \gg p$, and so is slightly less accurate for examples such as "simple" where this does not hold.

That last two columns show the verification time for the SMC phase alone, and for the total time including the abstract indicator function generation time shown in Table 1. We see that SIS outperforms CMC in all cases where verification is expensive, often by an order of magnitude or more. Also since the cost for generating the abstract indicator function is fixed regardless of the target RE, there will always be some target RE for which SIS outperforms CMC.

7 Conclusion

Statistical model checking (SMC) is a prominent approach for rigorous analysis of stochastic systems using Monte-Carlo simulations. In this paper, we developed a new technique, called Semantic Importance Sampling (SIS), to advance the state-of-the art in applying SMC to compute the probability of a rare event using a small number of simulations. SIS uses the semantics of the target system to recursively compute an abstract indicator function (AIF), which is subsequently employed to perform SMC. We also present two optimizations to SIS that reduce the number of calls to SMT solvers needed to compute the AIF. We have implemented SIS in a tool called OSMOSIS, and experimented with a number of examples. Our results indicate that SIS reduces cost of SMC by orders of magnitude, and our optimizations, in combination, reduce the cost of SMT solving by half. We believe that extending SIS to analyze stochastic systems compositionally, and combining it with symbolic simulation techniques, are important directions for future research.

References

1. Barbot, B., Haddad, S., Picaronny, C.: Coupling and importance sampling for statistical model checking. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 331–346. Springer, Heidelberg (2012)
2. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 1–12. Springer, Heidelberg (2011)
3. Reijsbergen, D., et al.: Rare event simulation for highly dependable systems with fast repairs. In: Proceedings of the 7th International Conference on Quantitative Evaluation of Systems (2010)
4. David, A., Du, D., Guldstrand Larsen, K., Legay, A., Mikučionis, M.: Optimizing Control Strategy Using Statistical Model Checking. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 352–367. Springer, Heidelberg (2013)
5. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
6. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT Solver for Nonlinear Theories over the Reals. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 208–214. Springer, Heidelberg (2013)
7. Gurfinkel, A., Chaki, S.: Boxes: A Symbolic Abstract Domain of Boxes. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 287–303. Springer, Heidelberg (2010)
8. Hansson, H., Jonsson, B.: A Logic for Reasoning about Time and Reliability. Formal Aspects of Computing (FACJ) 6(5), 512–535 (1994)
9. Jegourel, C., Legay, A., Sedwards, S.: Cross-entropy optimisation of importance sampling parameters for statistical model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 327–342. Springer, Heidelberg (2012)
10. Jegourel, C., Legay, A., Sedwards, S.: Importance Splitting for Statistical Model Checking Rare Properties. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 576–591. Springer, Heidelberg (2013)
11. Kahn, H.: Stochastic (monte carlo) attenuation analysis. Tech. Rep. P-88, Rand Corp. (1949)
12. Luckow, K.S., Pasareanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proc. of ASE (2014)
13. Borges, M., et al.: Compositional solution space quantification for probabilistics software analysis. In: Proceedings of PLDI: Programming Language Design and Implementation (June 2014)
14. Srinivasan, R.: Importance Sampling: Applications in Communications and Detection. Engineering online library, Springer (2002)
15. Stoelinga, M.: Alea jacta est: verification of probabilistic, real-time and parametric systems. Ph.D. thesis, University of Nijmegen, the Netherlands (2002)
16. Younes, H.L.S.: Verification and planning for stochastic processes with asynchronous events. Ph.D. thesis, Carnegie Mellon University (2004)
17. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)