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 addresses 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 maganitude even in systems with disjoint input-event regions.

1 Introduction

Many systems deployed in the real-world are stochastic, i.e., their behavior depends on random inputs (e.g., sensor readings, task execution times, etc.) As these 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 verification problem – estimating the probability of an event τ (e.g., some sort of failure) during the execution of a stochastic system $\mathcal{M}$. 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) [11], 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-0001772

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **18 OCT 2014**	2. REPORT TYPE **N/A**	3. DATES COVERED
4. TITLE AND SUBTITLE **Semantic Importance Sampling for Statistical Model Checking**		5a. CONTRACT NUMBER 5b. GRANT NUMBER 5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **; ; ; Wrage /Jeffery Hansen LutzChaki /SagarNiz /Dionsio deKlein /Mark**		5d. PROJECT NUMBER 5e. TASK NUMBER 5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S) 11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release, distribution unlimited.**		
13. SUPPLEMENTARY NOTES **The original document contains color images.**		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**SAR**	**3**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

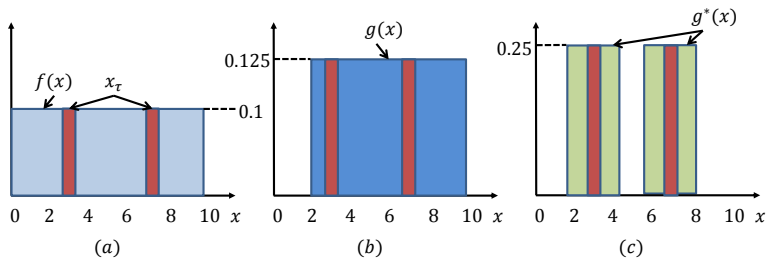

Fig. 1. Example of SIS; $f(x)$ = original input distribution; $g(x)$ = tilted distribution; $g^*(x)$ = distribution produced by SIS.

SMC produces two results – the estimate $\hat{p}$ of the probability p of τ and a measure e of precision of $\hat{p}$. 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., τ is rare). Importance Sampling [9] (IS) has been shown to address this challenge. Suppose the random input x to $\mathcal{M}$ has distribution $f(x)$. In IS, we first perform SMC under a different input distribution $g(x)$ that makes τ more likely (i.e., increases p), and then adjust the result back to $f(x)$.

Traditionally, importance Sampling is implemented by "tiliting" the parameters of the input distributions to increase the likelihood of τ. However, tilting is less effective if the set of inputs that cause τ, i.e., the input-event region denoted x_τ, is disjoint. For instance, this happens when analysing a program where τ 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 distrbution $f(x)$ is uniform in the range $[0, 10]$, and $x_\tau = [2.99, 3.01] \cup [6.99, 7.01]$. Figure 1(b) shows a titled distribution $g(x)$ uniform in the range $[2, 10]$. While $g(x)$ make τ more likely than $f(x)$, it still assigns positive weight to large parts – e.g., $(3.01, 6.99)$ – of the input space that do not belong to x_τ.

In this paper, we address this challenge, and make three specific contributions. First, we develop a new technique to construct more precise input distrbutions for IS – such as $g^*(x)$ 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 $f(x)$, and recursively computes a precise "over-approximation" of x_τ in the form of an abstract indicator function (AIF). In each step of the recursion, SIS constructs a verification condition using $\mathcal{M}$ and $f(x)$ and checks its satisfiability with a SMT solver to eliminate parts of the input space that are not in x_τ. The algorithm outputs an AIF represented by a set of "input cubes", i.e., a disjunction of intervals [**?**] 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 succesfully 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. In this way, SIS builds new bridges between these 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 over two orders of magnitude. 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`.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents background definitions and concepts. Section 4 presents SIS, and Section 5 presents our tool OSMOSIS. In Section 6, we present our experiments and results, and in Section 7, we conclude.

2 Related Work

SC: This needs a lot of work. Like statistical model checking, probabilistic model checking [10] is also an automated, algorithmic approach for computing numerical properties of stochastic systems. However, in this approach, 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) [5]. For example, the system could be a DTMC modeling the repeated throwing of a biased coin that comes up "heads" with probability 0.55 and "tails" with probability 0.45. The property could be the "probability of seeing 3 heads followed by 4 tails followed by 5 heads". Probabilistic model checking then involves an exhaustive exploration of the DTMC's statespace to compute the property's value. Typically, this involves the following steps: (i) compute the set of reachable discrete states S of an automaton constructed by composing the system with the property; (ii) construct a set of equations Q whose solution corresponds to the steady-state probabilities of S; and (iii) solve Q numerically and extract the value of the property from the solution. Probabilistic model checking is an active area of research, involving both theoretical advancements [3] and practical tool development [7]. It has been used to verify systems ranging from pacemakers [1], root contention protocols [8] and biological pathways [6]. For our approach, we used statistical model checking, since the systems we want to verify are too complex to be modeled as Markov chains.

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

$$p = \int I_{\mathcal{M}\models\Phi}(x)f(x)dx \qquad \hat{p} = \sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) \qquad RE(\hat{p}) = \frac{\sqrt{Var(\hat{p})}}{E[\hat{p}]}$$

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

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

Fig. 2. Background definitions. Var = variance; E = expected value; $W(x) = \frac{f(x)}{g(x)}$.

is to estimate the probability that $\mathcal{M}$ satisfies a property Φ, denoted $\mathcal{M} \models \Phi$, given a joint probability distribution $f(x)$ 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 $x \sim f(x)$ to mean x is distributed by $f(x)$. 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, a random vector $x \sim f(x)$ is generated. The system $\mathcal{M}$ is simulated with input x to generate a trace σ. The trial's outcome is 0 if Φ holds on σ, and 0 otherwise.

Define an indicator function $I_{\mathcal{M}\models\Phi}(x)$ that returns 1 if $\mathcal{M} \models \Phi$ under input x, and 0 otherwise. Then the probability p can be calculated as in Figure 2. It can be estimated as $\hat{p}$ shown in Figure 2, where N is the number of trials, and $x_i \sim f(x)$. The precision of $\hat{p}$ is quantified by its "relative error" $RE(\hat{p})$, defined in Figure 2. The second row in Figure 2 shows a known relationship [2] beween $RE(\hat{p})$, p and N, and its rearrangement with N on the left of the equality.

Importance Sampling. Figure 2 shows that the number of simulations needed to achieve a fixed precision with SMC increases rapidly as the target event becomes rarer. Importance Sampling [9] (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 $g(x)$ with lower variance, and then mathematically adjust the result back to the original distribution $f(x)$. The third row of Figure 2 shows the definition of p using $g(x)$ and the "weight function" $W(x) = \frac{f(x)}{g(x)}$. The estimator for this form is:

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

where the $x_i \sim g(x)$. The biggest challenge in applying IS effectively is choosing a "good" $g(x)$ that will reduce $Var(\hat{p})$. Typically this is done by "tilting" $f(x)$ by changing its distribution parameters (mean, variance etc.) However, as discussed, tilting is not effective if Φ is disjoint in the input space. Our main contribution,

SIS, is a technique for constructing a good $g(x)$ even in such cases. We describe this technique 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:

$$g^{\diamond}(x) = \frac{I_{\mathcal{M}\models\Phi}(x)f(x)}{p} \tag{2}$$

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

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

$$g^*(x) = \frac{I^*_{\mathcal{M}\models\Phi}(x)f(x)}{p^*} \tag{3}$$

$$W^*(x) = \frac{f(x)}{g^*(x)} = \frac{f(x)p^*}{I^*_{\mathcal{M}\models\Phi}(x)f(x)} = \frac{p^*}{I^*_{\mathcal{M}\models\Phi}(x)} \tag{4}$$

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

Probability Estimation and Relative Error in SIS. Substituting $W^*(x)$ from (4) into (1), we get the SIS estimator:

$$\hat{p} = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i)W^*(x_i) = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i)\frac{p^*}{I^*_{\mathcal{M}\models\Phi}(x_i)} \tag{5}$$

where $x_i \sim g^*(x)$. Note that, from (3), if $x_i \sim g^*(x)$, then $I^*_{\mathcal{M}\models\Phi}(x_i) = 1$. Also, p^* is a constant. Hence, (5) simplifies to:

$$\hat{p} = \frac{p^*}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) = p^* \times \hat{p}_{\text{raw}}, \text{ where} \tag{6}$$

$$\hat{p}_{\text{raw}} = \frac{1}{N}\sum_{i=1}^{N} I_{\mathcal{M}\models\Phi}(x_i) \tag{7}$$

is the estimated probability that for an input $x_i \sim g^*(x)$, $I_{\mathcal{M}\models\Phi}(x_i) = 1$, i.e., x_i is an input where $\mathcal{M} \models \Phi$ holds. Since $\hat{p}_{\text{raw}}$ is an unweighted average of Bernoulli random variables, its relative error can be estimated [2] as:

$$RE(\hat{p}_{\text{raw}}) \approx \frac{1}{\sqrt{p_{\text{raw}} N}} \tag{8}$$

Furthermore, from (6), 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. Use iterative abstraction-refinement to construct the AIF $I^*_{\mathcal{M}\models\Phi}(x)$.
2. Calcuate p^* and $g^*(x)$.
3. Use SMC to estimate $\hat{p}_{\text{raw}}$ with desired $RE(\hat{p}) = RE(\hat{p}_{\text{raw}})$, using $g^*(x)$ to draw random inputs. 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 abstraction-refinement 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_i in x, $F_i(x)$ is the Cumulative Distribution Function (CDF), $F_i^{-1}(u)$ is the inverse CDF (or quantile function), and $u_i = F^{-1}(x_i)$ is the quantile domain variable. Let ξ be a M-dimensional axis-aligned input domain hypercube defining an interval $[l_i, h_i]$ on each input variable x_i for $1 \leq i \leq M$. We also define the quantile domain hypercube c defined by the ranges $[F_i(l_i), F_i(h_i)]$ for each dimension. We use the notation $c = F(\xi)$ and $\xi = F^{-1}(c)$ to transform cubes between input and quantile space. We will use the terms *input cube* and *quantile cube* to refer to cubes in the input and quantile spaces, respectively. When the term *cube* is used without qualification we will assume quantile cubes. We can now represent the AIF in terms of a qualtile cube set C^* as:

$$I^*_{\mathcal{M}\models\Phi}(x) = \begin{cases} 1 \text{ if } \exists c \in C^* \mid F(x) \in c \\ 0 \text{ otherwise} \end{cases} \tag{9}$$

where $\forall c \in C^*, I_{\mathcal{M}\models\Phi}(x) = 1 \Rightarrow x \in c$.

Cube Splitting. Let ξ_U be the input cube defining the support of the input distribution function $f(x)$. The corresponding qunatile 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/i to mean the cube formed by spliting the interval on u_i in c in half, and retaining only the upper half. Similarly, $c/\bar{i}$ is the result of a similar operation where the lower half of the interval is retained. Note that we can split on the

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

Fig. 3. Basic AIF Generation Algorithm; G=variable grouping factor, M=number of input variables, $L_{\max}$=recusion depth limit, `Solve` = satisfiability check via SMT solver.

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 a input drawn from $f(x)$ falls in the result is halved. Thus, the probability of an input drawn from $f(x)$ falling in a level-k cube is $\frac{1}{2^k}$.

Iterative Abstraction-Refinement. Generation of the AIF $I^*_{\mathcal{M}\models\Phi}(x)$ is performed recursively through the hierarchical use of an SMT solver. The basic algorithm `aifGen` is shown in Figure 3. It takes as input the SMT representation of the indicator function I, and the input cube c over which to generate an abstaction. Constant $L_{\max}$ is the maximum recursion depth. It returns the subset of level-$L_{\max}$ cubes in C^* within cube c. C^* representing the AIF as defined in (9) can then be deterimined by calling `aifGen`, 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 I 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 containting 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 chosing the spliting 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_k forming the cubes c_0, and c_1 for the lower and upper half of the CDF interval on variable u_k in c. We then recufrsively call the generation algorithm on those two sub-cubes and return the union of the cube sets returned by each call.

Calculation of p^*. All cubes in the set C^* returned by `aifGen` are level-$L_{\max}$ and non-overlapping. As a result, p^* is calculated from the number of cubes in the set, and the recursion depth limit as:

$$p^* = \frac{|C^*|}{2^{L_{\max}}} \tag{10}$$

which represents the probability that an input from the original distribution $f(x)$ will fall in the abstract indicator function $I^*_{\mathcal{M}\models\Phi}(x)$.

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 3. 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 recurisve call on c_1, otherwise we make the recusrive 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 space 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 resursive 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 supress use of this optimization for the child invocations.

Optimized AIF Generation Algorithm. Figure 4 shows the fully optimized abstract indicactor function incorporating both of the optmizations 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

[2] It could be UKNOWN if result from cube c is UKNOWN, 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 k = (level(c)/G) % M;
(9)     Cube c_0 = c/k̄; Cube c_1 = c/k;
(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. 4. Optimized Abstract Indicator Function (AIF) Generation Algorithm; G=variable grouping factor, M=number of input variables, $L_{\max}$=recusion depth limit.

in which the counter-example cube is returned. We assume that the empty set is returned if the result is not SAT. At Line 5 we convert the input space cube ξ_d to a quantile space 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 $I^*_{\mathcal{M}\models\Phi}(x)$, and computing p^* and $g^*(x)$, the last step in SIS is the actual SMC. As peviously mentioned, we draw samples from the distribution $g^*(x)$ as defined in (3), then use (7) to estimate the raw probability $\hat{p}_{raw}$ and scale this by the value p^* calculated above.

No. Of Samples. The number of samples N^* needed to estimate $\hat{p}_{raw}$ is:

$$N^* = \frac{1 - p_{raw}}{p_{raw} RE^2(\hat{p}_{raw})} = \frac{1 - p/p^*}{p/p^* RE^2(\hat{p}_{raw})} \tag{11}$$

From (6), 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:

$$\frac{N}{N^*} = \frac{\frac{1-p}{pRE^2(\hat{p})}}{\frac{1-p/p^*}{p/p^* RE^2(\hat{p}_{raw})}} = \frac{1-p}{p^* - p} \approx \frac{1}{p^*} \tag{12}$$

Random Input Generation. To generate a random input from $g^*(x)$, we recognize that this is the equivalent of generating an input from $f(x)$ and accepting only those that fall within $I^*_{\mathcal{M}\models\Phi}(x)$. 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 $f(x)$. We then choose a uniform vector $u \in c$ and use the inverse CDF to generate the input vector as $x = F^{-1}(u)$.

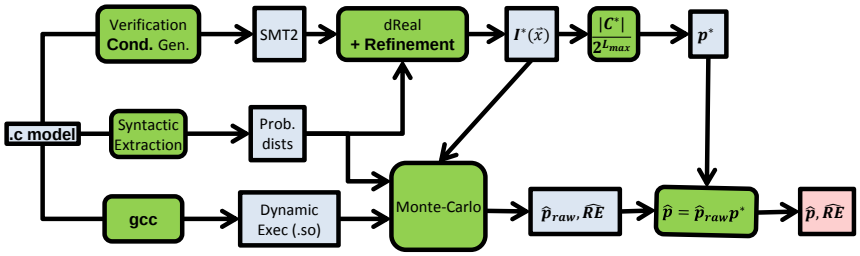

Fig. 5. Architecture of OSMOSIS Tool.

5 Osmosis

We implemented SIS in a tool called OSMOSIS. The input to OSMOSIS is a description of $\mathcal{M}$ in an anotated version of C, with the target property Φ defined as `ASSERT()` statements. OSMOSIS calculates the probability of an `ASSERT()` failure via SIS, using dReal[4] as the backend SMT solver.

Osmosis Architecture. Figure 5 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 3 or Figure 4) is used to build the AIF $I^*_{\mathcal{M}\models\Phi}(x)$. This AIF is used to calculate p^* and $g^*(x)$, and in conjunction with the dynamically loaded executable for $\mathcal{M}$ to calculate $\hat{p}_{\text{raw}}$ and $RE(\hat{p}_{\text{raw}})$. Finally, $\hat{p}$ is calculated using p^* and $\hat{p}_{\text{raw}}$.

Osmosis Input Format. Figure 6(a) shows an example OSMOSIS input model. The anotations at Lines 5 and 6 indicate the input to the model. Line 5 defines a random input named "a" with a uniform distribution between 0 and 5. Line 6 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 approrptiate, we refer to the model input collectively as he 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 6(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 1, then finally an assertion is made on Line 13. The include on Line 1, allows the model include the special 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 $I_{\mathcal{M}\models\Phi}(x)$, 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.

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

The model is also converted to single-static-assignment form so that each local variable is assigned once. Finally, `ASSERT()` conditions are negated since we are interested in testing if there are any inputs that can result in an assertion failure.

Assignments become equalities. Conditional (`if`) statements are translated by generating a variable for the condition, then translating both branches as consequences of implications of the 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 6(c) generates the SMT assertions in Figure 6(d). Loop (`while` and `for`) statements are unrolled and must include an annotation to indicate the maximim loop count. Note that the construction of φ is effective and linear in the size of the model.

Figure 6(b) shows the φ generated from the $\mathcal{M}$ given in Figure 6(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 the Monte-Carlo simulation to estimate $\hat{p}_{\text{raw}}$ using (7). 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 $x \sim g^*(x)$ is generated as described in the previous section and inititalized. A global flag

Name	In	$L_{\max}/G$	p^*	$1/p^*$	dReal Calls none	1	2	1+2	Time 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

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

variable indicating success/failure is also cleared. The function `INPUT_D()` indexes and returns a value from that vector. The `ASSERT()` statement tests the condition, 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 are inputs where $I_{\mathcal{M}\models\Phi}(x) = 1$, and those where the `ASSERT()` does not fail are inputs where $I_{\mathcal{M}\models\Phi}(x) = 0$.

6 Results

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

simple The example problem from Figure 6a.

hockey A hockey puck gets a random initial impulse from a random direction. Failure means that it stops on a point 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 peformed under Linux Ubuntu 12.04 on a 2.2GHz Intel Core i7 machine with 16 Gb of RAM.

Table 1 shows the results for AIF generation on each example. For each problem 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 leads to better performance for models with larger numbers of inputs. Recall from (12) that $1/p^*$ is an estimate for $\frac{N}{N^*}$. Note that the value of p^* value can be used to dynamically limit the recursion depth when generating the AIF, terminating when we have

Name	RE	$L_{\max}/G$	$\hat{p}$	N	N/N^*	Time (sec.) SMC	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}	–	2593	2593
		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. SMC Results; $RE = RE(\hat{p})$=target relative error; G=group factor.

acheived 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 peforms better, reducing the number of calls as well as total SMT solving time by half in most cases. Also, while there is some benift 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 $I^*_{\mathcal{M}\models\Phi}(x)$ 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 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 Crude Monte-Carlo (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 in closer agreement.

The column labeled N shows the number of samples needed to acheive the target relative error for each experiment, and the column labeled N/N^* shows the improvement of SIS over CMC for the SIS experiments. 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 recusion 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 that the predictions for the "simple" example slightly underestimate the measured improvements because the p^* values are close the measured probabilities while the predicition assumes $p^* \gg p$ and will tend to underestimate as p^* approaches p.

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 but one, often by multiple orders of magnitude. 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. Chen, T., Diciolla, M., Kwiatkowska, M.Z., Mereacre, A.: Quantitative Verification of Implantable Cardiac Pacemakers. In: Proceedings of the 33rd Real-Time Systems

Symposium (RTSS '12). pp. 263–272. IEEE Computer Society, San Juan, PR, USA (December 2012)
2. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.A. (eds.) ATVA. Lecture Notes in Computer Science, vol. 6996, pp. 1–12. Springer (2011), http://dblp.uni-trier.de/db/conf/atva/atva2011.html#ClarkeZ11
3. Feng, L., Kwiatkowska, M.Z., Parker, D.: Automated Learning of Probabilistic Assumptions for Compositional Reasoning. In: Giannakopoulou, D., Orejas, F. (eds.) Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (FASE '11). Lecture Notes in Computer Science, vol. 6603, pp. 2–17. Springer-Verlag, Saarbrücken, Germany (March–April 2011)
4. Gao, S., Kong, S., Clarke, E.: dreal: An smt solver for nonlinear theories over the reals. In: Bonacina, M. (ed.) Automated Deduction CADE-24, Lecture Notes in Computer Science, vol. 7898, pp. 208–214. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-38574-2_14
5. Hansson, H., Jonsson, B.: A Logic for Reasoning about Time and Reliability. Formal Aspects of Computing (FACJ) 6(5), 512–535 (December 1994)
6. Heath, J., Kwiatkowska, M.Z., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. Theoretical Computer Science (TCS) 391(3), 239–257 (February 2008)
7. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 585–591. Springer-Verlag, Snowbird, UT, July 14 - July 20, 2011. New York, NY (July 2011)
8. Kwiatkowska, M.Z., Norman, G., Sproston, J.: Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol. Formal Aspects of Computing (FACJ) 14(3), 295–318 (April 2003)
9. Srinivasan, R.: Importance Sampling: Applications in Communications and Detection. Engineering online library, Springer (2002), http://books.google.com/books?id=ZYytEwpH5BQC
10. Stoelinga, M.: Alea jacta est: verification of probabilistic, real-time and parametric systems. Ph.D. thesis, University of Nijmegen, the Netherlands (2002), available via http://www.soe.ucsc.edu/~marielle
11. Younes, H.L.S.: Verification and planning for stochastic processes with asynchronous events. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2004), http://portal.acm.org/citation.cfm?id=1087528