An Abstract Monte-Carlo Method for the Analysis of Probabilistic Programs*

David Monniaux
École Normale Supérieure
Laboratoire d'Informatique
45, rue d'Ulm
75230 Paris cedex 5
France
David.Monniaux@ens.fr

ABSTRACT

We introduce a new method, combination of random testing and abstract interpretation, for the analysis of programs featuring both probabilistic and non-probabilistic nondeterminism. After introducing "ordinary" testing, we show how to combine testing and abstract interpretation and give formulas linking the precision of the results to the number of iterations. We then discuss complexity and optimization issues and end with some experimental results.

1 INTRODUCTION

We introduce a generic method that lifts an ordinary abstract interpretation scheme to an analyzer yielding upper bounds on the probability of certain outcomes, taking into account both randomness and ordinary nondeterminism.

1.1 Motivations

It is sometimes desirable to estimate the probability of certain outcomes of a randomized computation process, such as a randomized algorithm or an embedded systems whose environment (users, mechanical and electrical parts ...) is modelized by known random distributions. In this latter case, it is particularly important to obtain upper bounds on the probability of failure.

Let us take an example. A copy machine has a computerized control system that interacts with the user through some control panel, drives (servo)motors and receives information from sensors. In some circumstances, the sensors can give bad information; for instance, some loose scrap of paper might prevent some optical sensor from working correctly. It is nevertheless desired that the probability that the machine will stop in an undesired state (without having returned the original, for instance) is very low given some realistic rates of failure from the sensors. To make the system more reliable, some sensors are redundant and the controlling algorithm tries to act coherently. Since adding sensors to the design costs space and hardware, it is interesting to evaluate the probabilities of failure even before building a prototype. A similar case can be made of industrial systems such as nuclear power plants were sensors have a limited life time and cannot be expected to be reliable. Sound analysis methods are especially needed for that kind of systems as safety guidelines are often formulated in terms of maximal probabilities of failures [8].

Treating the above problem in an entirely probabilistic fashion is not entirely satisfactory. While it is possible to modelize the user by properties such as "the probability that the user will hit the C key during the transfer of double-sided documents is less than 1%", this can prevent detecting some failures. For instance, if pressing some "unlikely" key combination during a certain phase of copying has a good chance of preventing correct accounting of the number of copies made, certain users might use it to get free copies. This is certainly a bug in the system. To account for the behavior of inputs that cannot be reliably modelized by random distributions (for instance, malicious attacks) we must incorporate nondeterminism.

1.2 Comparison to other works

An important literature has been published on software *testing* [11, 15, ...]; the purpose of testing techniques is to discover bugs and even to assert some sort of reliability criterion by testing the program on a certain number of cases. Such cases are either chosen randomly *(random testing)* or according to some *ad hoc* criteria, such as program statement or branch coverage *(partition testing)*. Partition-based methods can be enhanced by sampling randomly inside the partition elements. Often, since the actual distribution in production use is unknown, an uniform distribution is assumed.

*This work was partially funded by Commissariat à l'Énergie Atomique under contract 27234/VSF.

In our case, all the results our method gives are relative to some fixed, known, distributions driving some inputs. On the other hand, we will not have to assume some known distribution on the other inputs: they will be treated as nondeterministic. We thus avoid all problems pertaining to arbitrary choices of partitions or random distributions; our method, contrary to most testing methods, is fully mathematically sound.

There exists a domain called *probabilistic software engineering* [12] also aiming at estimating the safety of software. It is based on statistical studies on syntactic aspects of source code, or software engineering practices (programming language used, organization of the development teams ...), trying to estimate number of bugs in software according to recorded engineering experience. Our method does not use such considerations and bases itself on the actual software only.

Our analysis is based on a semantics equivalent to those proposed by Kozen [6, 7, 2nd semantics] and Monniaux [9]. We proposed a definition of abstract interpretation on probabilistic programs, using sets of measures, and gave a generic construction for abstract domains for the construction of analyzers. Nevertheless, this construction is rather "algebraic" and, contrary to the one explained here, does not make use of the well-studied properties of probabilities.

Several schemes of guarded logic commands [3] or refinement [10] have been introduced. While these systems are based on semantics broadly equivalent to ours, they are not analysis systems: they require considerable human input and are rather formal systems in which to construct derivations of properties of programs.

1.3 Contribution

We introduce for the first time a method combining statistical and static analyses. This method is proven to be mathematically sound. While some other methods have been recently proposed to statically derive properties of probabilistic programs in a general purpose programming language [9], ours is to our knowledge the first that makes use of statistical convergences.

1.4 Structure of the paper

We shall begin by an explanation of ordinary testing and its mathematical justification, then explain our "abstract Monte-Carlo" method. We shall then give the precise concrete semantics that an abstract interpreter must use to implement our method. We shall finish with some early results from our implementation.

We shall take as an example a simple imperative language. Our method is by no means limited to imperative programming, but we found this choice to be both close to the most common programming uses and relatively simple to explain our method on.

2 ABSTRACT MONTE-CARLO: THE IDEA

In this section, we shall explain, in a mathematical fashion, how our method works.

2.1 The Ordinary Monte-Carlo Testing Method

Let us consider a deterministic program c whose input x lies in X and whose output lies in Z. We shall note $[\![c]\!] : X \mapsto Z$ the semantics of c (so that $[\![\mathbf{c}]\!](x)$ is the result of the computation of c on the input x). We shall take X and Z two measurable spaces and constrain $[\![c]\!]$ to be measurable. These measurability conditions are technical and do not actually restrict the scope of programs to consider [9]. For the sake of simplicity, we shall suppose in this sub-section that c always terminates.

Let us consider $W \subseteq Z$ a measurable set of final states whose probability we wish to measure when x is a random variable whose probability measure is μ. The probability of W is therefore $\mu([\![c]\!]^{-1}(W))$. Noting

$$t_W(x) = \begin{cases} 1 & \text{if } [\![c]\!](x) \in W \\ 0 & \text{otherwise,} \end{cases}$$

this probability is the expectation $\mathbf{E}t_W$. The law of large numbers says that if we independently choose inputs x_k, with distribution μ, and compute the experimental average $\bar{t}_W^{(n)} = \frac{1}{n}\sum_{k=1}^{n} t_W(x_k)$, then $\lim_{n\to\infty} \bar{t}_W^{(n)} = \mathbf{E}t_W$. We can even evaluate the probability of underestimating the probability by more than t using the Chernoff-Hoeffding [4] [14, inequality A.4.4] bounds:

$$\Pr\left(\bar{X}^{(n)} - \mathbf{E}X \geq t\right) \leq e^{-2nt^2} \quad (1)$$

Taking $X = 1 - t_W$, it follows that

$$\Pr\left(\mathbf{E}t_W \geq \bar{t}_W^{(n)} + t\right) \leq e^{-2nt^2} \quad (2)$$

This method suffers from two drawbacks that make it unsuitable in certain cases:

- It supposes that all inputs to the program are either constant or driven according to a known probability distribution. In general, this is not the case: some inputs might well be only specified by intervals of possible values, without any probability measure. In such cases, it is common [11] to assume some kind of distribution on the inputs, such as an uniform one for numeric inputs. This might work in some cases, but grossly fail in others, since this is mathematically unsound.

- It supposes that the program terminates every time within an acceptable delay.

We propose a method that overcomes both of these problems.

2.2 Abstract Monte-Carlo

We shall now consider the case where the inputs of the program are divided in two: those, in X, that follow a random distribution μ and those that simply lie in some set Y. Now $[\![c]\!] : X \times Y \to Z$. The probability we are now trying to quantify is $\mu\{x \in X \mid \exists y \in Y\ [\![c]\!]\langle x, y\rangle \in W\}$. Some technical conditions must be met so that this probability is well-defined; namely, the spaces X and Y must be standard Borel spaces [5, Def. 12.5].[1] Since countable sets, $\mathbb{R}$, products of sequences of standard Borel spaces are standard Borel [5, §12.B], this restriction does not concern most semantics.

Noting

$$t_W(x) = \begin{cases} 1 & \text{if } \exists y \in Y \ [\![c]\!]\langle x, y\rangle \in W \\ 0 & \text{otherwise,} \end{cases}$$

this probability is the expectation $\mathbf{E} t_W$.

While it would be tempting, we cannot use a straightforward Monte-Carlo method since, in general, t_W is not computable.[2]

Let us first recall the mathematical foundations of abstract interpretation [2, 1]. Let us now consider two preordered sets $A^\sharp$ and $Z^\sharp$ so that there exist monotone functions $\gamma_A : A^\sharp \to \mathcal{P}(A)$, where $A = X \times Y$, and $\gamma_W : Z^\sharp \to \mathcal{P}(Z)$, where $\mathcal{P}(Z)$ is the set of parts of set Z, ordered by inclusion. The elements in $A^\sharp$ and $Z^\sharp$ represent some properties; for instance, if $X = \mathbb{Z}^m$ and $Y = \mathbb{Z}^n$, $A^\sharp$ could be the set of descriptions of polyhedra in $\mathbb{Z}^{m+n}$ and γ_A the function mapping the description to the set of points inside the polyhedron [2]. We then define an **abstract interpretation** of program c to be a monotone function $[\![c]\!]^\sharp : A^\sharp \to Z^\sharp$ so that

$$\forall a^\sharp \in A^\sharp,\ \forall a \in A\ a \in \gamma_A(A^\sharp) \Rightarrow [\![c]\!](a) \in \gamma_B \circ [\![c]\!]^\sharp(a^\sharp).$$

Let us also suppose that we can compute the following functions:

- $I : X \to A^\sharp$ so that $\forall x \in X\ \gamma_A \circ I(x) \supseteq \{x\} \times Y$;
- $\tau_W : Z^\sharp \to \{0, 1\}$ so that for all $z^\sharp \in Z^\sharp$, $\tau_W(z^\sharp) = 0 \Rightarrow \gamma_Z(z^\sharp) \cap W = \emptyset$.

It is then possible to compute a function T_W suitable for our needs: $T_W = \tau_W \circ [\![c]\!]^\sharp \circ I$.

We shall see in the following section how to build abstract interpreters with a view to using them for this Monte-Carlo method.

[1] Let us suppose X and Y are standard Borel spaces [5, §12.B]. $X \times Y$ is thus a Polish space [5, §3.A] so that the first projection π_1 is continuous. Let $A = \{x \in X \mid \exists y \in Y\ [\![c]\!]\langle x, y\rangle \in W\}$; then $A = \pi_1([\![c]\!]^{-1}(W))$. Since $[\![c]\!]$ is a measurable function and W is a measurable set, $[\![c]\!]^{-1}(W)$ is a Borel subset in the Polish space $X \times Y$. A is therefore analytic [5, Def. 14.1]; from Lusin's theorem [5, Th. 21.10], it is universally measurable. In particular, it is μ-measurable [5, §17.A]. $\mu(A)$ is thus well-defined.

[2] Let us take a Turing machine (or program in a Turing-complete language) F. There exists an algorithmic translation taking F as input and outputting the Turing machine $\tilde{F}$ computing the total function $\varphi_{\tilde{F}}$ so that

$$\varphi_{\tilde{F}}\langle x, y\rangle = \begin{cases} 1 & \text{if } F \text{ terminates in } y \text{ or less steps on input } x \\ 0 & \text{otherwise.} \end{cases}$$

Let us take $X = Y = \mathbb{N}$ and $Z = \{0, 1\}$ and the program $\tilde{F}$, and define $t_{\{1\}}$ as before. $t_{\{1\}}(x) = 1$ if and only if F terminates on input x. It is a classical fact of computability theory that the $t_{\{1\}}$ function is not computable for all F [13].

3 A CONCRETE SEMANTICS SUITABLE FOR ANALYSIS

From the previous section, it would seem that it is easy to use any abstract interpreter in a Monte-Carlo method. Alas, we shall now see that special precautions must be taken in the presence of calls to random generators inside loops.

3.1 Concrete Semantics

We have for now spoken of deterministic programs taking one input x chosen according to some random distribution and one input y in some domain. Calls to random generators (such as the POSIX `drand48()` function) are usually modelized by a sequence of independent random variables. If a bounded number of calls ($\leq N$) to such generators is used in the program, we can consider them as input values: x is then a tuple $\langle x_1, \ldots, x_N, v\rangle$ where $x_1, \ldots, x_n$ are the values for the generator and v is the input of the program. If an unbounded number of calls can be made, it is tempting to consider as an input a countable sequence of values $(x_n)_{n\in\mathbb{N}}$ where x_1 is the result of the first call to the generator, x_2 the result of the second call … ; a formal description of such a semantics has been made by Kozen [6, 7].

Such a semantics is not very suitable for program analysis. Intuitively, analyzing such a semantics implies tracking the number of calls made to number generators. The problem is that such simple constructs as:

```
if (...)  { random(); } else {}
```

are difficult to handle: the countings are not synchronized in both branches.

We shall now propose another semantics, identifying occurrences of random generators by their program location and loop indices. The Backus-Naur form of the programming language we shall consider is:

instruction ::= *elementary*
  | *instruction* `;` *instruction*
  | `if` *boolean_expr*
    `then` *instruction*
    `else` *instruction*
    `endif`
  | `while` *boolean_expr*
    `do` *instruction*
    `done`

We leave the subcomponents largely unspecified, as they are not relevant to our method. *elementary* instructions are deterministic, terminating basic program blocks like assignments and simple expression evaluations. *boolean_expr* boolean expressions, such as comparisons, have semantics as sets of acceptable environments. For instance, a *boolean_expr* expression can be `x < y + 4`; its semantics is the set of execution environments where variables `x` and `y` verify the above comparison. If we restrict ourselves to a finite number n of integer variables, an environment is just a n-tuple of integers.

The denotational semantics of a code fragment c is a mapping from the set X of possible execution environments before the instruction into the set Y of possible environments after the instruction. Let us take an example. If we take environments as elements of $\mathbb{Z}^3$, representing the values of three integer variables x, y and z, then $[\![\texttt{x:=y+z}]\!]$ is the strict function $\langle x,y,z\rangle \mapsto \langle y+z,y,z\rangle$. Semantics of basic constructs (assignments, arithmetic operators) can be easily dealt with this forward semantics; we shall now see how to deal with flow control.

The semantics of a sequence is expressed by simple composition

$$[\![e_1;\ e_2]\!] = [\![e_2]\!] \circ [\![e_1]\!] \tag{3}$$

Tests get expressed easily, using as the semantics $[\![c]\!]$ of a boolean expression c the set of environments it matches:

$$[\![\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!](x) = \\ \text{if } x \in [\![c]\!] \text{ then } [\![e_1]\!](x) \text{ else } [\![e_2]\!](x) \tag{4}$$

and loops get the usual least-fixpoint semantics (considering the point-wise extension of the Scott flat ordering on partial functions)

$$[\![\texttt{while}\ c\ \texttt{do}\ f]\!] = \text{lfp}(\lambda\phi.\lambda x. \\ \text{if } x \in [\![c]\!] \text{ then } \phi \circ [\![f]\!](x) \text{ else } x). \tag{5}$$

Non-termination shall be noted by $\perp$.

As for expressions, the only constructs whose semantics we shall precise are the random generators. We shall consider a finite set G of different generators. Each generator g outputs a random variable r_g with distribution μ_g; each call is independent from the precedent calls. Let us also consider the set P of program points and the set $\mathbb{N}^*$ of finite sequences of positive integers. The set $C = P \times \mathbb{N}^*$ shall denote the possible times in an execution where a call to a random generator is made: $\langle p, n_1 n_2 ... n_l\rangle$ notes the execution of program point p at the n_1-th execution of the outermost program loop, ... , n_l-th execution of the innermost loop at that point. C is countable. We shall suppose that inside the inputs of the program there is for each generator g in G a family $(\hat{g}_{\langle p,w\rangle})_{\langle p,w\rangle \in C}$ of random choices.

The semantics of the language then become:

$$[\![e_1;\ e_2]\!] = [\![e_2]\!] \circ [\![e_1]\!] \tag{6}$$

Tests get expressed easily, using as the semantics $[\![c]\!]$ of a boolean expression c the set of environments it matches:

$$[\![\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!].\langle w,x\rangle = \\ \text{if } x \in [\![c]\!] \text{ then } [\![e_1]\!].\langle w,x\rangle \text{ else } [\![e_2]\!].\langle w,x\rangle \tag{7}$$

Loops get the usual least-fixpoint semantics (considering the point-wise extension of the Scott flat ordering on partial functions):

$$[\![\texttt{while}\ c\ \texttt{do}\ f]\!].\langle w_0,x_0\rangle = \\ \text{lfp}\,(\lambda\phi.\lambda\langle w,x\rangle.\text{if } x \in [\![c]\!] \text{ then } \phi \circ S \circ [\![f]\!]\langle w,x\rangle) \text{ else } x).\langle 1.w_0, x_0\rangle \tag{8}$$

where $S.\langle c.w,x\rangle = \langle (c+1).w,x\rangle$. The only change is that we keep track of the iterations of the loop.

As for random expressions,

$$[\![p : \texttt{random}_g]\!].\langle w,x\rangle = \hat{g}_{\langle p,w\rangle} \tag{9}$$

where p is the program point.

This semantics is equivalent to the denotational semantics proposed by Kozen [6, 7, 2nd semantics] and Monniaux [9], the semantic of a program being a continuous linear operator mapping an input measure to the corresponding output. The key point of this equivalence is that two invocations of random generators in the same execution have different indices, which implies that a fresh output of a random generator is randomly independent of the environment coming to that program point.

3.2 Analysis

Our analysis algorithm is a randomized version of an ordinary abstract interpreter. Informally, we treat calls to random generators are treated as follows:

- calls occurring outside fixpoint convergence iterations are interpreted as constants chosen randomly by the interpreter;
- calls occurring inside fixpoint convergence iterations are interpreted as upper approximations of the whole domain of values the random generator yield.

For instance, in the following C program:

```
int x;
x = coin_flip(); /* coin_flip() returns 0 or 1 */
                 /* each with probability 0.5 */
for(i=0; i<5; i++)
{
  x = x + coin_flip();
}
```

the first occurrence of `coin_flip()` will be treated as a random value, while the second occurrence will be treated as the least upper bound of $\{0\}$ and $\{1\}$.

This holds for "naive" abstract interpreters; more advanced ones might perform "dynamic loop unrolling" or other semantic transformations corresponding to a refinement of the abstract domain to handle execution traces:

$$[\![\texttt{while}\ c\ \texttt{do}\ e]\!](x) = \\ \left(\left(\bigcup_{k<N_1+N_2} \psi^k(x)\right) \cup \psi^{N_2}\left(\text{lfp}\left(\lambda l.\psi^{N_1}(x) \cup \psi(l)\right)\right)\right) \cap [\![c]\!]^C \tag{10}$$

where $\psi(x) = [\![e]\!](x \cap [\![c]\!])$ and N_1 and N_2 are possibly decided at run-time, depending on the computed values. In this case, the interpreter uses a random generator for the occurrences of $\texttt{random}_g$ operations outside lfp computations and abstract values for the operations inside lfp's. Its execution defines the finite set K of $\langle p, n_1 \ldots n_l\rangle$ tags uniquely identifying the random values chosen for $\hat{g}_{\langle p,n_1 \ldots n_l\rangle}$, as well as the values $(\check{g}_c)_{c\in K}$ that have been chosen. This yields

$$\forall (\hat{g}_c)_{g\in G, c\in C}\ \forall y \in Y\ (\forall c \in K\ \hat{g}_c = \check{g}_c) \Rightarrow \\ [\![c]\!]\langle (\hat{g}_c)_{g\in G,c\in C}, y\rangle \in \gamma_Z(z^\sharp) \tag{11}$$

which means that

$$\forall (\hat{g}_c)_{g\in G, c\in C}\ (\forall c \in K\ \hat{g}_c = \check{g}_c) \Rightarrow \\ t_W((\hat{g}_c)_{g\in G,c\in C}) \leq \tau_W(z^\sharp) \tag{12}$$

If we virtually choose randomly some $\breve{g}_c$ for $c \notin K$, we know that $t_W((\breve{g}_c)_{g\in G, c\in C}) \le \tau_W(z^\sharp)$. Furthermore, $(\breve{g}_c)$ follows the product random distribution $\mu_g^{\otimes C}$ (each $\breve{g}_c$ has been chosen independently of the others according to measure μ_g).

Let us summarize: we wish to generate upper bounds of experimental averages of a Bernoulli random variable $t_W : X \to \{0,1\}$ whose domain has the product probability measure $\mu_I \otimes \bigotimes_{g\in G} \mu_g^{\otimes C}$ where μ_I is the input measure and the μ_g's are the measures for the random number generators. The problem is that the domain of this random variable is made of countable sequences; thus we cannot generate its input strictly speaking. We instead effectively choose at random a finite number of coordinates for the countable sequences, and compute a common upper bound for t_W for all inputs identical to our chosen values on this finite number of coordinates. This is identical to virtually choosing a random countable sequence x and getting an upper bound of its image by t_W.

Implementing such an analysis inside an ordinary abstract interpreter is easy. The calls to random generators are interpreted as either a random generation, or as the least upper bound over the range of the generator, depending on a "randomize" flag. This flag is adjusted depending on whether the interpreter is computing a fixpoint. The interpreter does not track the indices of the random variables: these are only needed for the proof of correctness. The analyzer does a certain number n of trials and outputs the experimental average $\bar{t}_W^{(n)}$. As a convenience, our implementation also outputs the $\bar{t}_W^{(n)} + t$ upper bound so that there is at least a probability $1-\varepsilon$ that this upper bound is safe according to inequation (2). This is the value that is reported in the experiments of section 5.

While our explanations referred to a forward semantics, the abstract interpreter can of course combine forward and backward analysis [1, section 6], provided the chosen random values are recorded so that subsequent passes of analysis can reuse them. Another related improvement, explained in the next section, uses a preliminary backward analysis prior to random generation.

4 COMPLEXITY

The complexity of our method is the product of two independent factors:

- the complexity of one ordinary static analysis of the program; strictly speaking, this complexity depends not only on the program but on the random choices made, but we can take a rough "average" estimate that depends only on the program being analyzed;

- the number of iterations, that depends only on the requested safety margins; the minimal number of iterations to reach a certain safety criterion can be derived from inequalities [14, appendix A] such as inequation (1) and does not depend on the actual program being analyzed.

We shall now focus on the latter factor, as the former depends on the particular case of analysis being implemented.

Let us recall inequation (2): $\Pr\left(\mathbf{E}t_W \ge \bar{t}_W^{(n)} + t\right) \le e^{-2nt^2}$. *It means that to get with $1-\varepsilon$ probability an ap-*

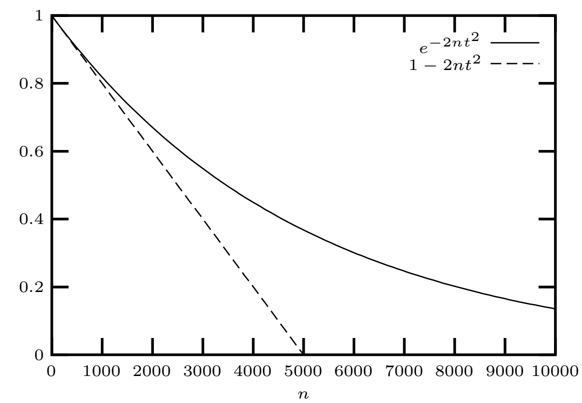

Figure 1: Upper bound on the probability that the computed probability exceeds the real value by more than t, for $t = 0.01$.

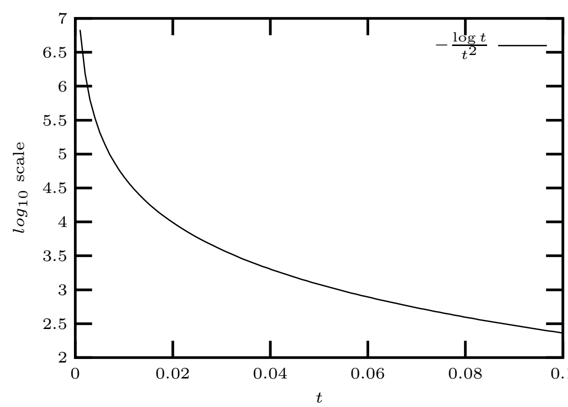

Figure 2: Numbers of iterations necessary to achieve a probability of false report on the same order of magnitude as the error margin.

proximation of the requested probability μ, it is sufficient to compute an experimental average over $\left\lceil -\frac{\log \varepsilon}{2t^2} \right\rceil$ trials.

This exponential improvement in quality (Fig. 1) is nevertheless not that interesting. Indeed, in practice, we might want ε and t of the same order of magnitude as μ. Let us take $\varepsilon = \alpha t$ where α is fixed. We then have $n \sim -\frac{\log t}{t^2}$, which indicates prohibitive computation times for low probability events (Fig. 2). *This high cost of computation for low-probability events is not specific to our method; it is true of any Monte-Carlo method*, since it is inherent in the speed of convergence of averages of identically distributed random variables; this relates to the speed of convergence in the central limit theorem [14, ch 1]. It can nevertheless be circumvented by tricks aimed at estimating the desired low probability by computing some other, bigger, probability from which the desired result can be computed.

Fortunately, such an improvement is possible in our method. If we know that $\pi_1(\llbracket c \rrbracket^{-1}(W)) \subseteq R$, with a measurable R, then we can replace the random variable t_W by its restriction to R: $t_{W|R}$; then $\mathbf{E}t_W = \Pr(R).\mathbf{E}t_{W|R}$. If $\Pr(R)$ and $\mathbf{E}t_W$ are on the same order of magnitude, this means that $\mathbf{E}t_{W|R}$ will be large and thus that the number of required iterations will be low. *Such a restricting R can be obtained by static analysis, using ordinary backwards abstract interpretation.*

A salient point of our method is that our Monte-Carlo computations are **highly parallelizable**, with linear speedups: n iterations on 1 machine can be replaced by n/m iterations on m machines, with very little communication. Our method thus seems especially adapted for clusters of low-cost PC with off-the-shelf communication hardware, or even more distributed forms of computing. Another improvement can be to compute bounds for several W sets simultaneously, doing common computations only once.

5 PRACTICAL IMPLEMENTATION AND EXPERIMENTS

We have a prototype implementation of our method, implemented on top of an ordinary abstract interpreter doing forward analysis using integer and real intervals. Figures 3 to 5 show various examples for which the probability could be computed exactly by symbolic integration. Figure 6 shows a simple program whose probability of outcome is difficult to figure out by hand. Of course, more complex programs can be handled, but the current lack of support of user-defined functions and mixed use of reals and integers prevents us from supplying real-life examples. We hope to overcome these limitations soon as implementation progresses.

6 CONCLUSIONS

We have proposed a generic method that combines the well-known techniques of abstract interpretation and Monte-Carlo program testing into an analysis scheme for probabilistic and nondeterministic programs, including reactive programs whose inputs are modelized by both random and nondeterministic inputs. This method is mathematically proven correct, and uses no assumption apart from the distributions and nondeterminism domains supplied by the user. It yields

```
int x, i;
know (x>=0 && x<=2);
i=0;
while (i < 5)
{
  x += coin_flip();
  i++;
}
know (x<3);
```

Figure 3: **Discrete probabilities.** The analyzer establishes that, with **99% safety**, the probability p of the outcome $(x < 3)$ is less than **0.509** given worst-case nondeterministic choices of the precondition $(x \geq 0 \land x \leq 2)$. The analyzer used $n = 10000$ random trials. Formally, p is $\Pr\left(\texttt{coin\_flip} \in \{0,1\}^5 \mid \exists x \in [0,2] \cap \mathbb{Z}\ \llbracket P \rrbracket(\texttt{coin\_flip}, x) < 3\right)$. Each `coin_flip` is chosen randomly in $\{0,1\}$ with an uniform distribution. The exact value is **0.5**.

```
double x;
know (x>=0. && x<=1.);
x+=uniform()+uniform()+uniform();
know (x<2.);
```

Figure 4: **Continuous probabilities.** The analyzer establishes that, with **99%** safety, the probability p of the outcome $(x < 2)$ is less than **0.848** given worst-case nondeterministic choices of the precondition $(x \geq 0 \land x \leq 1)$. The analyzer used $n = 10000$ random trials. Formally, p is $\Pr\left(\texttt{uniform} \in [0,1]^3 \mid \exists x \in [0,1]\ \llbracket P \rrbracket(\texttt{uniform}, x) < 2\right)$. Each `uniform` is chosen randomly in $[0,1]$ with the Lebesgue uniform distribution.
The exact value is $5/6 \approx \mathbf{0.833}$.

```
double x, i;
know(x<0.0 && x>0.0-1.0);
i=0.;

while (i < 3.0)
{
  x += uniform();
  i += 1.0;
}
know (x<1.0);
```

Figure 5: **Loops.** The analyzer establishes that, with **99% safety**, the probability p of the outcome $(x < 1)$ is less than 0.859 given worst-case nondeterministic choices of the precondition $(x < 0 \land x > -1)$. The analyzer used $n = 10000$ random trials. Formally, p is $\Pr\left(\texttt{uniform} \in [0,1]^3 \mid \exists x \in [0,1]\ \llbracket P \rrbracket(\texttt{uniform}, x) < 1\right)$. Each `uniform` is chosen randomly in $[0,1]$ with the Lebesgue uniform distribution. The exact value is $5/6 \approx \mathbf{0.833}$.

```
{
  double x, y, z;
  know (x>=0. && x<=0.1);
  z=uniform(); z+=z;
  if (x+z<2.)
  {
    x += uniform();
  } else
  {
    x -= uniform();
  }
  know (x>0.9 && x<1.1);
}
```

Figure 6: The analyzer establishes that, with **99% safety**, the probability p of the outcome $(x > 0.9 \wedge x < 1.1)$ is less than **0.225** given worst-case nondeterministic choices of the precondition $(x \geq 0 \wedge x \leq 0.1)$. Formally, p is $\Pr\left(\texttt{uniform} \in [0,1]^2 \mid \exists x \in [0, 0.1]\ [\![P]\!](\texttt{uniform}, x) \in [0.9, 1.1]\right)$. Each `uniform` is chosen randomly in $[0, 1]$ with the Lebesgue uniform distribution.

upper bounds on the probability of outcomes of the program, according to the supplied random distributions, with worse-case behavior according to the nondeterminism; whether or not this bounds are sound is probabilistic, and a lower-bound of the soundness of those bounds is supplied. While our explanations are given using a simple imperative language as an example, the method is by no means restricted to imperative programming.

The number of trials, and thus the complexity of the computation, depends on the desired precision. The method is parallelizable with linear speed-ups. The complexity of the analysis, or at least its part dealing with probabilities, increases if the probability to be evaluated is low. However, static analysis can come to help to reduce this complexity.

We have implemented the method on top of a simple static analyzer and conducted experiments showing interesting results on small programs written in an imperative language. As implementation progresses, we expect to have results on complex programs akin to those used in embedded systems.

REFERENCES

[1] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13):103–179, 1992.

[2] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Conference on Principles of Programming Languages*. ACM Press, 1978.

[3] Jifeng He, K. Seidel, and A. McIver. Probabilistic models for the guarded command language. *Science of Computer Programming*, 28(2–3):171–192, April 1997. Formal specifications: foundations, methods, tools and applications (Konstancin, 1995).

[4] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.*, 58(301):13–30, 1963.

[5] Alexander S. Kechris. *Classical descriptive set theory.* Graduate Texts in Mathematics. Springer-Verlag, New York, 1995.

[6] D. Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science*, pages 101–114, Long Beach, Ca., USA, October 1979. IEEE Computer Society Press.

[7] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[8] N. G. Leveson. Software safety: Why, what, and how. *Computing Surveys*, 18(2):125–163, June 1986.

[9] David Monniaux. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)*, number 1824 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[10] Carroll Morgan, Annabelle McIver, Karen Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Formal Aspects of Computing*, 8(6):617–647, 1996.

[11] Simeon Ntafos. On random and partition testing. In Michal Young, editor, *ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 42–48, 1998.

[12] Panel on Statistical Methods in Software Engineering. *Statistical Software Engineering.* National Academy of Sciences, 1996.

[13] H. Rogers. *Theory of recursive and effective computability.* MGH, 1967.

[14] Galen R. Shorack and Jon A. Wellner. *Empirical Processes with Applications to Statistics.* Wiley series in probability and mathematical statistics. John Wiley & Sons, 1986.

[15] P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical software testing pages 99-109. In *Proceedings of the 1993 international symposium on Software testing and analysis*, pages 99–109. Association for Computer Machinery, June 1993.