Abstraction Refinement for Probabilistic Software

Mark Kattenbelt, Marta Kwiatkowska, Gethin Norman, and David Parker

Oxford University Computing Laboratory, Parks Road, Oxford, OX1 3QD

Abstract. We present a methodology and implementation for verifying ANSI-C programs that exhibit probabilistic behaviour, such as failures or randomisation. We use abstraction-refinement techniques that represent probabilistic programs as Markov decision processes and their abstractions as stochastic two-player games. Our techniques target quantitative properties of software such as "the maximum probability of file-transfer failure" or "the minimum expected number of loop iterations" and the abstractions we construct yield lower and upper bounds on these properties, which then guide the refinement process. We build upon state-of-the-art techniques and tools, using SAT-based predicate abstraction, symbolic implementations of probabilistic model checking and components from GOTO-CC, SATABS and PRISM. Experimental results show that our approach performs very well in practice, successfully verifying actual networking software whose complexity is significantly beyond the scope of existing probabilistic verification tools.

1 Introduction

Software model checking techniques have become increasingly sophisticated in recent years. Witness for example the success of the SLAM project, used to identify bugs in Windows device drivers. This technology is based on *predicate abstraction* [1] and *counterexample-guided abstraction-refinement* (CEGAR) [2], which are used to construct increasingly precise finite-state abstractions of programs to either demonstrate the violation of a safety property (e.g. a buffer overflow) or guarantee the absence of such faults.

In this paper, we present novel techniques for verification of software that exhibits *probabilistic* behaviour, for example due to interaction with components prone to failures or due to the use of randomisation. We target ANSI-C programs, extending the language with two probabilistic functions: `coin`(p), which returns 1 with probability p and 0 with probability $1-p$; and `prob`(n), which returns an integer between 0 and $n-1$ uniformly at random. These provide a natural way of modelling both failures (e.g. a function call to open a network connection which fails with probability p) and randomisation (e.g. selecting a random pivot to sort a list of n items). We also provide functions to model nondeterministic behaviour, such as calls to underspecified procedures or program input.

Our approach is based on *probabilistic model checking*, a generalisation of model checking for systems that exhibit stochastic behaviour. It is applied to

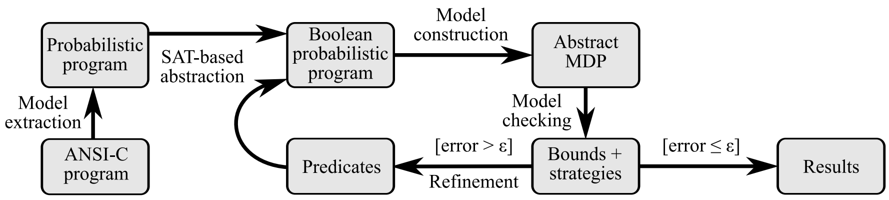


Fig. 1. Abstraction-refinement loop for probabilistic programs.

state transition systems augmented with probabilistic information, such as Markov decision processes (MDPs). The properties to be verified are *quantitative* in nature and, since MDPs model both probability and nondeterminism, relate to best- or worst-case bounds on behaviour, e.g. "the maximum probability of file-transfer failure" or "the minimum expected number of loop iterations". Various tools are available for probabilistic model checking, such as PRISM, MRMC and LiQuor, and the techniques have been successfully applied to a wide range of applications from security to biological modelling. They have yet, however, to be used in the context of real programming languages.

We present a quantitative analogue of the well-known CEGAR loop (see Figure 1), which successively refines an abstraction of a concrete model until it is sufficiently precise. The combination of probabilistic and nondeterministic behaviour is naturally modelled with MDPs. The underlying theory is based on representing abstractions of MDPs as two-player stochastic games [3], which use the two players to distinguish the nondeterminism of the concrete system and that introduced during abstraction. We use SAT-based predicate abstraction [4] to construct, from a concrete probabilistic program, an abstraction in the form of a *Boolean probabilistic program*, whose semantics is the stochastic game abstraction. Model checking the game [3] yields lower and upper bounds on a quantitative property of the original MDP (such as "the minimum probability that the program terminates successfully" or "the maximum expected number of function calls during program execution") and strategies (for the two players) that achieve the bounds. Although the analysis does not yield counterexamples (in the sense of a trace to an error state), the bounds and strategies provide both a quantitative measure of the precision of the abstraction and, when necessary, provide a means of refining the abstraction.

Related work. The closest work is [5], which proposes a CEGAR framework for predicate abstraction of MDPs described in a simple guarded-command language. This verifies or refutes properties of the form "the maximum probability of error is at most p" for a probability threshold p. In [5], abstractions are also MDPs (as proposed in [6]), and only upper bounds on maximum probabilities are computed. So, to refute a property, probabilistic counterexamples [7] (comprising multiple paths whose combined probability exceeds p) are generated. If these paths are spurious, they are used to generate further predicates using interpolation. Perhaps the most important distinguishing feature of our work is the use of two-player games as abstractions. These provide *lower and upper bounds*, which avoids enumerating a set of paths, which can be large or even infinite (resulting in non-termination). The bounds also enable a *quantitative* approach: we target properties without thresholds such as "what is the maximum probability of error?". A second important difference is that we use real programming languages, rather than guarded-commands. Lastly, we also consider rewards.

Other abstraction-refinement techniques for probabilistic systems have been proposed. In [6], abstractions of MDPs are refined by state-space partitioning but, like [5], this yields only one-sided bounds. Magnifying lens abstraction [8] partitions an MDP and analyses each one separately but, since it relies on building the full concrete model, even a symbolic implementation [9] is unlikely to scale to real software. In [10], a counterexample-based abstraction-refinement technique for planning problems is proposed; however, there is no implementation to test this on practical examples.

In [11], a framework is described for analysing probabilistic programs based on expectation transformers, but this is not applied to real software and does not include an automated step for refining abstractions. Another approach to abstracting probabilistic models is to label transitions with intervals, e.g. [12], but this has only been applied to models without nondeterminism. In earlier work [13], we applied the game-based abstraction of [3] to predicate abstraction, but not for source code and without refinement.

Another important direction for the verification of probabilistic software is the extension of abstract interpretation to the probabilistic setting [14–16], although these approaches have yet to be combined with refinement. Other approaches to the probabilistic verification of imperative languages include APEX [17], which performs equivalence checking for a simple procedural language, and the tool LiQuor [18], whose modelling language Probmela includes imperative language style constructs. Neither approach uses abstraction.

Contributions. The precise contributions of this paper are the following:

- we present the first abstraction-refinement techniques for probabilistic systems that are specifically targeted at real programming languages;
- we describe a complete implementation of these techniques, built using state-of-the-art tools and techniques, and demonstrate its applicability on several real software case studies that cannot be verified with existing tools;
- we improve upon existing approaches by using game-based abstraction to obtain both lower and upper bounds on quantitative properties.

2 Background

A probability distribution over a set S is a function $\lambda : S \to [0,1]$ satisfying $\sum_{s\in S}\lambda(s){=}1$. Let $\mathsf{dist}(S)$ denote the set of all distributions over S. We use the notation $p_1\!:\!s_1\!+\!\cdots\!+\!p_n\!:\!s_n$ for the distribution $\lambda \in \mathsf{dist}(S)$ such that $\lambda(s_i) = p_i$. For a set X, $x \in X$, $\lambda \in \mathsf{dist}(S)$ and $\Lambda \in \mathcal{P}(\mathsf{dist}(S))$, let $x \boxtimes \lambda \in \mathsf{dist}(X \times S)$ denote the distribution where $(x\boxtimes\lambda)(x,s){=}\lambda(s)$ and $x\boxtimes\Lambda$ denote the set $\{x\boxtimes\lambda \mid \lambda \in \Lambda\}$.

Definition 1 (Markov decision process). *A* Markov decision process *(MDP) is a tuple* $\mathsf{M} = \langle S, S_i, \delta\rangle$*, where* S *is a set of* states*,* $S_i \subseteq S$ *are* initial states *and*

$\delta : S \to \mathcal{P}(\mathsf{dist}(S))$ *is a* probabilistic transition function*, which maps each state to a finite, non-empty set of probability distributions over states.*

An MDP's behaviour is both probabilistic and nondeterministic. A transition $s \xrightarrow{\lambda} s'$ from state s is made by first nondeterministically selecting a distribution $\lambda \in \delta(s)$, and then selecting a successor state s' with probability $\lambda(s')$. A *path* is a sequence of transitions. A state is *reachable* if there is a path to it from an initial state. Under an *adversary*, which resolves all nondeterminism, we can define a probability measure over paths [19]. For a target set $\mathcal{F} \subseteq S$, we then define the *minimum* and *maximum probability*, under any adversary, of reaching $\mathcal{F}$ from state s, denoted $\mathbf{p}^-_s(\mathcal{F})$ and $\mathbf{p}^+_s(\mathcal{F})$ respectively. By also associating rewards (non-negative real values) with transitions, we can also define the *minimum* and *maximum expected reward* of reaching $\mathcal{F}$.

Definition 2 (Abstract MDP). *An* abstract MDP *is a tuple* $\hat{\mathsf{M}} = \langle \hat{S}, \hat{S}_i, \hat{\delta} \rangle$*, where* $\hat{S}$ *is a set of* abstract states*,* $\hat{S}_i \subseteq \hat{S}$ *are* initial abstract states *and* $\hat{\delta} : \hat{S} \to \mathcal{P}(\mathcal{P}(\mathsf{dist}(\hat{S})))$ *is an* abstract probabilistic transition function*, which maps each state to a set of sets of distributions over states.*

The underlying semantics of an abstract MDP is a two-player stochastic game [20]. A transition $\hat{s} \xrightarrow{\langle \Lambda, \lambda \rangle} \hat{s}'$ includes two successive nondeterministic choices: first, a set of distributions $\Lambda \in \hat{\delta}(\hat{s})$ is chosen by player 1; then, an element $\lambda \in \Lambda$ is selected by player 2. The successor $\hat{s}'$ is chosen with probability $\lambda(\hat{s}')$. Similarly to MDPs, under *strategies* for players 1 and 2, which resolve all nondeterminism, we can define a probability measure over paths. For target $\hat{\mathcal{F}} \subseteq \hat{S}$, we define both the probability and expected reward of reaching $\hat{\mathcal{F}}$ from a state $\hat{s}$ or choice Λ when both players minimise, player 1 minimises and 2 maximises, player 1 maximises and 2 minimises and both maximise. For reachability probabilities, we denote these $\mathbf{p}^{--}_{\hat{s}}(\hat{\mathcal{F}})$, $\mathbf{p}^{-+}_{\hat{s}}(\hat{\mathcal{F}})$, $\mathbf{p}^{+-}_{\hat{s}}(\hat{\mathcal{F}})$ and $\mathbf{p}^{++}_{\hat{s}}(\hat{\mathcal{F}})$ respectively.

As proposed in [3], abstract MDPs are used to represent abstractions of MDPs. The key idea is to separate the two forms of nondeterminism: the first choice in a transition (player 1) represents nondeterminism caused by abstraction; the second choice (player 2) corresponds to the nondeterminism of the original MDP. For an MDP M, the construction of its abstraction is based on an *abstraction function* $\alpha : S \to \hat{S}$ from concrete to abstract states. We lift α to distributions and sets and in the obvious way, e.g. $\alpha(\lambda)(\hat{s}) = \sum_{\alpha(s)=\hat{s}} \lambda(s)$.

Definition 3 (Abstraction of MDPs [3]). *Given an MDP* $\mathsf{M} = \langle S, S_i, \delta \rangle$ *and abstraction function* $\alpha : S \to \hat{S}$*, the* abstraction of M under α *is the abstract MDP* $\alpha(\mathsf{M}) = \langle \hat{S}, \alpha(S_i), \hat{\delta} \rangle$ *where for any* $\hat{s} \in \hat{S}$ *we have that* $\Lambda \in \hat{\delta}(\hat{s})$ *if and only if there exists* $s \in S$ *such that* $\alpha(s)=\hat{s}$ *and* $\Lambda=\alpha(\delta(s))$*.*

The abstraction $\alpha(\mathsf{M})$ of M yields lower and upper bounds on probabilities and expected rewards of M [3]. For example, for any $s \in S$ and $\mathcal{F} \subseteq S$:

$$\begin{array}{ccccc} \mathbf{p}^{--}_{\alpha(s)}(\alpha(\mathcal{F})) & \leqslant & \mathbf{p}^-_s(\mathcal{F}) & \leqslant & \mathbf{p}^{+-}_{\alpha(s)}(\alpha(\mathcal{F})) \\ \mathbf{p}^{-+}_{\alpha(s)}(\alpha(\mathcal{F})) & \leqslant & \mathbf{p}^+_s(\mathcal{F}) & \leqslant & \mathbf{p}^{++}_{\alpha(s)}(\alpha(\mathcal{F})) \end{array}$$

Algorithms for computing these measures for MDPs and abstract MDPs can be found in e.g. [21] and [22], respectively.

3 Probabilistic programs

In this section, we define *probabilistic programs*. Since these are both probabilistic and nondeterministic in nature, their semantics are given in terms of MDPs.

Let $\mathcal{U}$ denote a *data universe*, that is, the set of all possible *data valuations*. Given an expression $\mathtt{E}$ over $\mathcal{U}$ and a valuation $u \in \mathcal{U}$, $\mathtt{E}(u)$ denotes the evaluation of $\mathtt{E}$ on u. For an l-value $\mathtt{x}$ and an expression $\mathtt{E}$, $u[\mathtt{x}{\mapsto}\mathtt{E}]$ denotes the valuation derived from u by setting $\mathtt{x}$ to $\mathtt{E}(u)$ and $\mathrm{Type}(\mathtt{x})$ the set of all values of the same type as $\mathtt{x}$. The set of *commands* $\mathcal{C}_\mathcal{U}$ over $\mathcal{U}$ consists of: conditional statements $[\mathtt{B}]$, deterministic assignments $\mathtt{x}{=}\mathtt{E}$, probabilistic assignments $\mathtt{i}{=}\mathtt{coin}(p)$ and $\mathtt{i}{=}\mathtt{prob}(n)$, nondeterministic assignments $\mathtt{i}{=}\mathtt{ndet}(n)$ and $\mathtt{i}{=}\mathtt{ndet}()$, where $\mathtt{B}$ is a Boolean expression over $\mathcal{U}$, $\mathtt{E}$ is an expression over $\mathcal{U}$, $\mathtt{x}$ is an l-value, $\mathtt{i}$ is an integer l-value, $p \in (0,1)$ and $n \in \mathbb{N}$. We use GOTO-CC [23] to transform programs such that all expressions are side-effect free and all assignments are type-consistent. The l-values in deterministic assignments can be of any valid type, including pointers, structures and arrays.

Definition 4 (Probabilistic program). *A* probabilistic program P *is a tuple* $\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *where* $\mathcal{U}$ *is a data universe,* $\langle V, E\rangle$ *is a finite directed (control-flow) graph with initial vertex* v_i *and* $\mathcal{L} : E \to \mathcal{C}_\mathcal{U}$ *labels edges with commands.*

We assume that if an outgoing edge from a vertex v is labelled with a conditional, then so are all other outgoing edges from v and, for each $u \in \mathcal{U}$, precisely one of these conditions holds. Any other vertex has only a single outgoing edge. Therefore, each vertex is associated with a single type of command.

During program extraction function calls are inlined; thus we do not support unbounded recursion. We also do not consider dynamic memory allocation or floating point arithmetic. We assume a conventional model checker guarantees the absence of any undefined behaviour, e.g. a null-pointer dereference, during the evaluation of expressions and l-values. We deal with pointers through static points-to analysis augmented with (dynamic) information using predicates [4]. The semantics of non-probabilistic commands are captured with transitions that occur with probability 1.

Definition 5 (Probabilistic program semantics). *Let* $\mathsf{P}{=}\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *be a probabilistic program. The semantics of* P *is the MDP* $[\![\mathsf{P}]\!]{=}\langle V{\times}\mathcal{U}, \{v_i\}{\times}\mathcal{U}, \delta\rangle$ *where for any* $\langle v, u\rangle \in V{\times}\mathcal{U}$*:*

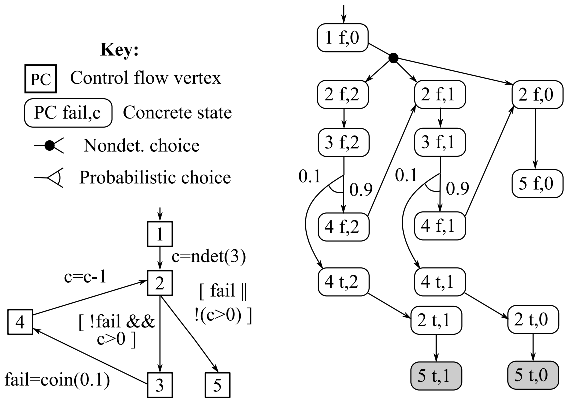
$$\delta(v,u) = \left\{ v'\boxtimes\lambda \;\middle|\; \langle v, v'\rangle \in E,\ \lambda \in [\![\mathcal{L}\langle v, v'\rangle]\!](u) \right\}$$

and $[\![\mathtt{cmd}]\!] : \mathcal{U}{\to}\mathcal{P}(\mathsf{dist}(\mathcal{U}))$ *is the semantics of command* $\mathtt{cmd}$ *such that for* $u \in \mathcal{U}$*:*

$$\begin{aligned}
[\![[\mathtt{B}]]\!](u) &= \{1 : u\} \textit{ if } \mathtt{B}(u) \textit{ and } \emptyset \textit{ otherwise} \\
[\![\mathtt{x}{=}\mathtt{E}]\!](u) &= \{1 : u[\,\mathtt{x}{\mapsto}\mathtt{E}(u)\,]\} \\
[\![\mathtt{i}{=}\mathtt{coin}(p)]\!](u) &= \{(1{-}p) : u[\mathtt{i}{\mapsto}\mathtt{0}] + p : u[\mathtt{i}{\mapsto}\mathtt{1}]\} \\
[\![\mathtt{i}{=}\mathtt{prob}(n)]\!](u) &= \{\tfrac{1}{n} : u[\mathtt{i}{\mapsto}\mathtt{0}] + \cdots + \tfrac{1}{n} : u[\mathtt{i}{\mapsto}n{-}\mathtt{1}]\} \\
[\![\mathtt{i}{=}\mathtt{ndet}(n)]\!](u) &= \{1 : u[\mathtt{i}{\mapsto}\mathtt{0}], \ldots, 1 : u[\mathtt{i}{\mapsto}n{-}\mathtt{1}]\} \\
[\![\mathtt{i}{=}\mathtt{ndet}()]\!](u) &= \{1 : u[\mathtt{x}{\mapsto}\mathtt{val}] \mid \mathtt{val} \in \mathrm{Type}(\mathtt{x})\}\,.
\end{aligned}$$

```
   bool fail = false;
   int c = 0;

   int main() {
1:   c = num_to_send();
2:   while (!fail && c>0) {
3:      fail = send_msg();
4:      c--;
     }
5:   assert(!fail);
   }
   int num_to_send() {
     return ndet(3);
   }
   bool send_msg() {
     return (coin(0.1)==1);
   }
```



(a) (b) (c)

Fig. 2. Simple example: (a) C code; (b) probabilistic program; (c) MDP semantics.

Example 1. Figure 2 shows a fragment of C code, corresponding probabilistic program and MDP semantics. The code comprises a loop which tries to send `c` messages, `c` being obtained by calling `num_to_send()`, which nondeterministically returns 0, 1 or 2. A message is sent by calling `send_msg()`, which fails with probability 0.1. Once a transmission fails, the loop terminates. The maximum probability of any transmission failing (i.e. of reaching control-flow location 5 with `fail` equal to true) is 0.19 and occurs when `c` is set to 2.

4 Abstraction of probabilistic programs

In practice, constructing the concrete semantics of all but the simplest programs is intractable. Hence, in order to verify real programs, it becomes essential to consider *abstraction*. We adopt the approach of [24] and use *Boolean probabilistic programs*, which retain the same control-flow structure as their concrete counterpart but abstract the concrete data universe $\mathcal{U}$ to a finite Boolean abstraction induced by set of *predicates* (Boolean expressions) over $\mathcal{U}$ [1].

Definition 6 (Boolean probabilistic program). *A* Boolean probabilistic program B *is a tuple* $\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *where* Φ *is a set of* n *(quantifier-free) predicates,* $\langle V, E\rangle$ *is a directed (control-flow) graph with initial vertex* v_i *and* $\mathcal{T} : E \rightarrow (\mathbb{B}^n \rightarrow \mathcal{P}(\mathcal{P}(\mathsf{dist}(\mathbb{B}^n))))$ *is an* abstract probabilistic transition function.

The semantics of a concrete probabilistic program is an MDP; the semantics of its abstraction, a Boolean probabilistic program, is an abstract MDP. Conventional (non-probabilistic) Boolean programs are typically used to represent existential abstractions [25] where both concrete and abstract semantic models are labelled transition systems. In this case, a Boolean program can be seen as a special instance of a (concrete) program. In our setting this does not hold since the semantic models of concrete and abstract programs differ. Hence, we define

Boolean probabilistic programs directly in terms of a mapping $\mathcal{T}$ rather than through commands (as we did with $\mathcal{L}$ for probabilistic programs).

Definition 7 (Boolean probabilistic program semantics). *The semantics of a Boolean probabilistic program* $\mathsf{B}=\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *is the abstract MDP* $[\![\mathsf{B}]\!]=\langle V\times\mathbb{B}^n, \{v_i\}\times\mathbb{B}^n, \hat{\delta}\rangle$ *where* $n=|\Phi|$ *and for any* $\langle v,a\rangle \in V\times\mathbb{B}^n$:

$$\hat{\delta}(v,a) = \left\{ v'\boxtimes\Lambda \,\middle|\, \langle v,v'\rangle \in E,\ \Lambda \in \mathcal{T}(v,v')(a) \right\}.$$

Given a probabilistic program $\mathsf{P}=\langle \mathcal{U}, \langle V, E\rangle, v_{\mathrm{i}}, \mathcal{L}\rangle$ and predicates $\Phi=\{\phi_1,\ldots,\phi_n\}$ over the data universe $\mathcal{U}$, we now show how to construct the corresponding abstract Boolean probabilistic program. The abstraction function $\alpha : \mathcal{U} \to \mathbb{B}^n$ is given by $\alpha(u)=\langle \phi_1(u),\ldots,\phi_n(u)\rangle$; we lift α to distributions and sets and to $V\times\mathcal{U}$ by letting $\alpha\langle v,u\rangle=\langle v,\alpha(u)\rangle$. For any $a=\langle b_1,\ldots,b_n\rangle \in \mathbb{B}^n$, let $a[i]=b_i$.

Definition 8 (Abstraction of probabilistic programs). *Given a probabilistic program* $\mathsf{P}=\langle \mathcal{U}, \langle V, E\rangle, v_i, \mathcal{L}\rangle$ *and set of predicates* Φ *with abstraction function* α*, the* abstraction of P under Φ *is given by the Boolean probabilistic program* $\alpha(\mathsf{P})=\langle \Phi, \langle V, E\rangle, v_i, \mathcal{T}\rangle$ *where for any* $e\in E$ *and* $a \in \mathbb{B}^n$*:* $\Lambda \in \mathcal{T}(e)(a)$ *if and only if there exists* $u\in\mathcal{U}$ *such that* $\alpha(u)=a$ *and* $\Lambda=\alpha([\![\mathcal{L}(e)]\!](u))\neq\emptyset$.

Applying MDP abstraction (Definition 3) to the semantics of a concrete program (Definition 5) yields the same abstraction as the Boolean program (Definitions 8 and 7). This is because, although Definition 8 applies the abstraction per control-flow edge, there is no nondeterminism between edges in the concrete program.

Example 2. Figure 3(a) shows a representation of the Boolean probabilistic program obtained by abstracting the program from Example 1 using predicates `fail` and (`c==0`). We use $\phi{=}*_1$ and $\phi{=}*_2$ to describe the abstract probabilistic transition function in which the value of the predicate ϕ is determined by player 1 or player 2, respectively. For example, if $a{=}\langle b_1,b_2\rangle \in \mathbb{B}^2$ and $\lambda_{\mathrm{f}},\lambda_{\mathrm{t}}$ be the distributions $1{:}(b_1,\mathrm{f})$ and $1{:}(b_1,\mathrm{t})$, then (`c==0`)$=*_1$ on edge e indicates that $\mathcal{T}(e)(a)=\{\{\lambda_{\mathrm{f}}\},\{\lambda_{\mathrm{t}}\}\}$, whereas (`c==0`)$=*_2$ means that $\mathcal{T}(e)(a)=\{\{\lambda_{\mathrm{f}},\lambda_{\mathrm{t}}\}\}$. Figure 3(b) shows the abstract MDP semantics of the Boolean probabilistic program. Each abstract state is labelled with lower/upper bounds on the maximum probability of reaching control-flow location 5 with variable `fail` equal to true.

SAT-based abstraction. In order to construct the abstraction of a probabilistic program, we adopt the SAT-based techniques of [4], in which the basic idea is to construct the abstract transition relation for each edge of a program's control-flow graph by formulating it as a Boolean satisfiability problem. Each satisfiable instance corresponds to an element of the transition relation; all such instances can be enumerated efficiently by a SAT-solver. An important advantage of this approach is that it allows a detailed bit-level semantics of the source code.

Our setting is slightly different: our abstractions are Boolean probabilistic programs and so we construct abstract probabilistic transition functions, rather than abstract transition relations. Despite this, fundamental similarities remain: the use of Boolean programs means that the abstraction for each command can

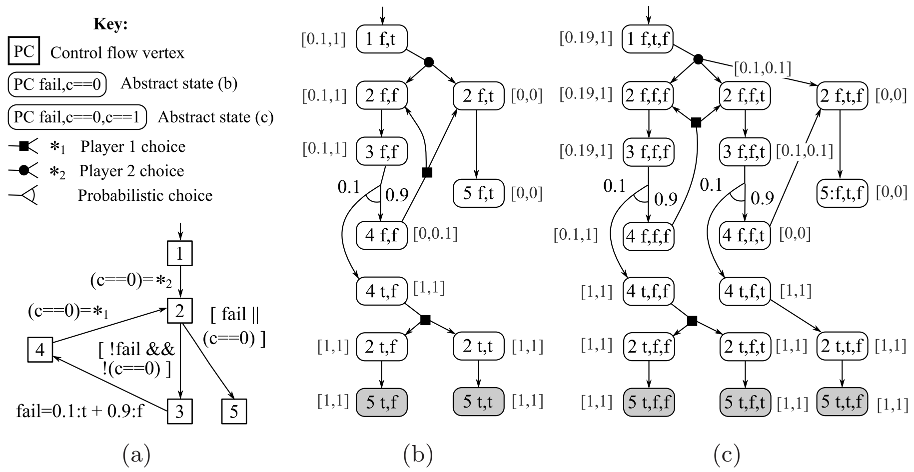


Fig. 3. Abstractions for Example 1: (a) & (b) Boolean probabilistic program and abstract MDP for initial abstraction; (c) abstract MDP for refined abstraction.

be built in isolation; and the definition of abstraction can be phrased as an existential satisfiability problem.

Consider a probabilistic program $\mathsf{P}{=}\langle\mathcal{U}, \langle V, E\rangle, v_{\mathrm{i}}, \mathcal{L}\rangle$ and set of n predicates Φ. To construct the abstraction we need only construct the abstract probabilistic transition function $\mathcal{T}(e)$ for each edge $e \in E$. Recall from Definition 8 that, for each $a \in \mathbb{B}^n$, $\mathcal{T}(e)(a)$ returns a set of probability distributions over $\mathbb{B}^n$ where $\Lambda \in \mathcal{T}(e)(a)$ if and only if there exists $u \in \mathcal{U}$ such that $\alpha(u){=}a$ and $\Lambda{=}\alpha(\llbracket\mathcal{L}(e)\rrbracket(u)){\neq}\emptyset$. We now formulate $\mathcal{T}(e)$ as a satisfiability problem whose structure is dependent on the command labelling e.

Conditionals. If e is labelled $[\mathtt{B}]$, then $\llbracket\mathcal{L}(e)\rrbracket(u)$ equals $\{1{:}u\}$ if $\mathtt{B}(u)$ and $\emptyset$ otherwise, and hence $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1:a\}$ and:

$$\exists\, u \in \mathcal{U}\,.\,\big(\,\alpha(u){=}a \wedge \mathtt{B}(u)\,\big)$$

Deterministic Assignments. If e is labelled $\mathtt{x{=}E}$, then $\llbracket\mathcal{L}(e)\rrbracket(u){=}\{1{:}u[\mathtt{x}{\mapsto}\mathtt{E}(u)]\}$, and hence $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1:a'\}$ for some $a' \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U}\,.\,\big(\,\alpha(u){=}a \wedge \alpha(u[\,\mathtt{x}{\mapsto}\mathtt{E}(u)\,]){=}a'\,\big)$$

Probabilistic assignments. If e is labelled $\mathtt{i{=}coin}(p)$, then $\llbracket\mathcal{L}(e)\rrbracket(u){=}\{(1{-}p):u[\mathtt{i}{\mapsto}\mathtt{0}] + p : u[\mathtt{i}{\mapsto}\mathtt{1}]\}$, and $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{(1{-}p){:}a_{\mathtt{0}} + p{:}a_{\mathtt{1}}\}$ for some $a_{\mathtt{0}}, a_{\mathtt{1}} \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U}\,.\,\big(\,\alpha(u){=}a \wedge \alpha(u[\mathtt{i}{\mapsto}\mathtt{0}]){=}a_{\mathtt{0}} \wedge \alpha(u[\mathtt{i}{\mapsto}\mathtt{1}]){=}a_{\mathtt{1}}\,\big)$$

The case when e is labelled $\mathtt{i{=}prob}(n)$ follows similarly.

Nondeterministic assignments. If e is labelled $\mathtt{i{=}ndet}(n)$, then $\llbracket\mathcal{L}(e)\rrbracket(u) = \{1{:}u[\mathtt{i}{\mapsto}\mathtt{0}], \ldots, 1{:}u[\mathtt{i}{\mapsto}n{-}1]\}$, and therefore $\Lambda \in \mathcal{T}(e)(a)$ if and only if $\Lambda{=}\{1 : a_0, \ldots, 1 : a_{n-1}\}$ for some $a_0, \ldots, a_{n-1} \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U}\,.\,\big(\,\alpha(u){=}a \;\wedge\; \alpha(u[\mathtt{i}{\mapsto}\mathtt{0}]){=}a_0 \;\wedge \ldots \wedge\; \alpha(u[\mathtt{i}{\mapsto}\mathtt{n{-}1}]){=}a_{n-1}\,\big)$$

Computing $\mathcal{T}(e)$ in this way for $\texttt{i}{=}\texttt{ndet}(n)$ with large n can result in intractable SAT formulas. The same is true if we adopt a similar approach for assignments $\texttt{x}{=}\texttt{ndet}()$. So, for such commands, we make the assumption that $\mathcal{T}(e)(a)$ contains a single set, Λ say. Since $[\![\mathcal{L}(e)]\!](u) = \{1 : u[\texttt{x}{\mapsto}\texttt{val}] \mid \texttt{val} \in \text{Type}(\texttt{x})\}$, we have $\lambda \in \Lambda$ if and only if $\lambda{=}1 : a'$ for some $a' \in \mathbb{B}^n$ such that:

$$\exists\, u \in \mathcal{U} \,.\, \exists\, \texttt{val} \in \text{Type}(\texttt{x}) \,.\, \big(\, \alpha(u){=}a \wedge \alpha(u[\,\texttt{x}{\mapsto}\texttt{val}\,]){=}a' \,\big)$$

The above assumption holds if Φ can be partitioned into $\{\Phi_{\texttt{x}}, \Phi\backslash\Phi_{\texttt{x}}\}$, where predicates in $\Phi_{\texttt{x}}$ refer only to $\texttt{x}$, and those in $\Phi\backslash\Phi_{\texttt{x}}$ are not influenced by $\texttt{x}$. Fortunately, this case turns out to be sufficient in practice. Assignments of the form $\texttt{x}{=}\texttt{ndet}()$ typically model operating system calls that nondeterministically succeed or fail and the actual values being assigned are irrelevant to the property under consideration. The same assumption is used for $\texttt{i}{=}\texttt{ndet}(n)$ with large n.

5 Abstraction refinement

In order to make our abstraction techniques practically applicable, it is essential to develop *refinement* techniques that can automatically construct an abstraction which is sufficiently precise for verification but also small enough for efficient analysis. In conventional CEGAR approaches, this is based on the generation of counterexamples (paths to an error state), which are either *feasible* (i.e. a concretisation exists and the safety property is refuted) or *spurious* (in which case, the counterexample is used to generate additional predicates for refinement).

The crucial difference in our setting is that model checking is quantitative: our aim is not just to establish the absence/existence of a path to a target, but rather to compute quantitative properties, e.g. the minimum probability or maximum reward of reaching a target. The abstraction-refinement loop we propose (as illustrated earlier in Figure 1) is based on iterative refinement of an abstraction (an abstract MDP) until the lower and upper bounds for the property of interest differ by less than some threshold ε. In this section, we describe how the abstract MDP yields predicates that can be used to refine the abstraction.

Predicate discovery. We describe the case for maximum probabilities (the process for minimum probabilities and expected rewards is identical). Therefore suppose we have a probabilistic program $\mathsf{P} = \langle \mathcal{U}, \langle V, E\rangle, v_{\mathrm{i}}, \mathcal{L}\rangle$, target $\mathcal{F} \subseteq V \times \mathcal{U}$ and predicates Φ with abstraction function α such that $\alpha(\mathsf{P})$ is not sufficiently precise for some initial state, i.e. there exists $a \in \mathbb{B}^n$ such that:

$$\mathbf{p}^{++}_{\langle v_{\mathrm{i}},a\rangle}(\alpha(\mathcal{F})) - \mathbf{p}^{-+}_{\langle v_{\mathrm{i}},a\rangle}(\alpha(\mathcal{F})) > \varepsilon \,. \tag{1}$$

Recall that model checking the abstraction $\alpha(\mathsf{P})$ also yields strategies that achieve the lower and upper bounds. A strategy tells us how nondeterminism is resolved, i.e. it gives for each abstract state $\langle v, a\rangle$ an element Λ of $\hat{\delta}\langle v, a\rangle$ in the abstract MDP. Each such choice Λ encodes a subset of the concrete states represented by $\langle v, a\rangle$, namely the set $\{\langle v, u\rangle \in V \times \mathcal{U} \mid \alpha(u){=}a,\ \alpha(\delta\langle v, u\rangle){=}\Lambda\}$.

Based on results from [3, 26], the inequality in (1) guarantees that there exists an abstract state $\langle s^\star, a^\star\rangle$ and *distinct* choices Λ^- and Λ^+ made by lower and upper bound strategies in $\langle s^\star, a^\star\rangle$ such that either $\mathbf{p}^{-+}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{-+}_{\Lambda^+}(\alpha(\mathcal{F}))$ or $\mathbf{p}^{++}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{++}_{\Lambda^+}(\alpha(\mathcal{F}))$. We call $\langle s^\star, a^\star\rangle$ a *refinable state*. Our aim is to eliminate the choice between Λ^- and Λ^+, through a predicate that separates the concrete states corresponding to these two choices. For example, if $\mathbf{p}^{-+}_{\Lambda^-}(\alpha(\mathcal{F})) < \mathbf{p}^{-+}_{\Lambda^+}(\alpha(\mathcal{F}))$, then choosing Λ^+ makes the lower bound higher. Hence we can improve the lower bound of the states encoded by Λ^+ by eliminating the choice.

Below, we describe how to generate a new predicate, based on the command associated with the control location of the refinable state $\langle v^\star, a^\star\rangle$. By construction, $v^\star$ either has one or more outgoing edges labelled with conditionals or a single outgoing edge $\langle v^\star, v\rangle$ labelled with an assignment.

Conditionals. If the outgoing edges of $v^\star$ are conditionals, then $\Lambda^-{=}\{1{:}\langle v^-, a^\star\rangle\}$ and $\Lambda^+{=}\{1{:}\langle v^+, a^\star\rangle\}$ for some distinct v^- and v^+ such that $\langle v^\star, v^-\rangle$ and $\langle v^\star, v^+\rangle$ are labelled with conditionals ($[\mathtt{B}^-]$ and $[\mathtt{B}^+]$ say). We add $\mathtt{B}^-$ to Φ. By assumption on probabilistic programs, at most one of $\mathtt{B}^-$ and $\mathtt{B}^+$ is satisfiable in the concretisations of an abstract state, eliminating the choice between Λ^- and Λ^+.

Deterministic Assignments. If $\langle v^\star, v\rangle$ is labelled $\mathtt{x}{=}\mathtt{E}$, then $\Lambda^-{=}\{1{:}\langle v, a^-\rangle\}$ and $\Lambda^+{=}\{1{:}\langle v, a^+\rangle\}$ for some $a^-, a^+ \in \mathbb{B}^n$. Since Λ^- and Λ^+ are distinct there exists a predicate $\phi_i \in \Phi$ such that $a^-[i]{\neq}a^+[i]$. We add the predicate $\mathrm{WP}(\phi_i, \mathtt{x}{=}\mathtt{E})$. By definition, this predicate is satisfied if, after executing the assignment $\mathtt{x}{=}\mathtt{E}$, ϕ_i holds, i.e. $\mathrm{WP}(\phi_i, \mathtt{x}{=}\mathtt{E})(u)$ if and only if $\phi_i(u[\,\mathtt{x}{\mapsto}\mathtt{E}\,])$. If $\langle v^\star, u^-\rangle$ is encoded by Λ^-, then $\alpha(u^-[\,\mathtt{x}{\mapsto}\mathtt{E}\,]){=}\Lambda^-$, and hence $\phi_i(u^-[\,\mathtt{x}{\mapsto}\mathtt{E}\,])$ if and only if $a^-[i]$. A similar argument holds for states encoded by Λ^+. As $a^-[i]{\neq}a^+[i]$, the new predicate is either satisfied by all states encoded by Λ^- and none by Λ^+ or vice versa, and therefore $\mathrm{WP}(\phi_i, \mathtt{x}{=}\mathtt{E})$ eliminates the choice between Λ^- and Λ^+.

Probabilistic assignments. If $\langle v^\star, v\rangle$ is labelled $\mathtt{i}{=}\mathtt{coin}(p)$, then, Λ^- and Λ^+ are of the form $\{(1{-}p){:}\langle v, a_0^-\rangle + p{:}\langle v, a_1^-\rangle\}$ and $\{(1{-}p){:}\langle v, a_0^+\rangle + p{:}\langle v, a_1^+\rangle\}$. Since $\Lambda^-{\neq}\Lambda^+$, there exists $\phi_i \in \Phi$ such that $a_j^-[i]{\neq}a_j^+[i]$ for some $0{\leqslant}j{\leqslant}1$ and we add $\mathrm{WP}(\phi_i, \mathtt{x}{=}\mathtt{j})$ to Φ which, by similar arguments to above, removes the choice between Λ^- and Λ^+. The case when $\langle v^\star, v\rangle$ is labelled $\mathtt{i}{=}\mathtt{prob}(n)$ follows similarly.

Nondeterministic assignments. By construction an assignment $\mathtt{x}{=}\mathtt{ndet}()$ consists of a single choice, hence $\langle v^\star, v\rangle$ cannot be labelled $\mathtt{x}{=}\mathtt{ndet}()$. If $\langle v^\star, v\rangle$ is labelled $\mathtt{i}{=}\mathtt{ndet}(n)$, then Λ^- and Λ^+ are of the form $\{1{:}\langle v, a_0^-\rangle, \ldots, 1{:}\langle v, a_{n-1}^-\rangle\}$ and $\{1{:}\langle v, a_0^+\rangle, \ldots, 1{:}\langle v, a_{n-1}^+\rangle\}$ respectively. Since $\Lambda^-{\neq}\Lambda^+$, there exists $\phi_i \in \Phi$ such that $a_j^-[i]{\neq}a_j^+[i]$ for some $0{\leqslant}j{\leqslant}n{-}1$ and we add the predicate $\mathrm{WP}(\phi_i, \mathtt{x}{=}\mathtt{j})$.

As in conventional CEGAR, our method is incomplete due to the use of WP-based abstraction refinement [27]. However, as we will show later, our approach successfully finds suitable abstractions in practice.

Example 3. Consider the program of Example 1 and the abstraction from Example 2 (Figures 3(a) and 3(b)). The abstract state (4 f,f) is the only one with both a player 1 choice and differing bounds (0 if branching right to (2 f,t) and 0.1 if branching left to (2 f,f)), i.e. it is the only possible refinable state.

The command for control-flow vertex 4 is the deterministic assignment `c=c−1` and the predicate (`c==0`) differs between (2 f,t) and (2 f,f) so our new predicate is WP((`c==0`), `c=c−1`), i.e. (`c==1`). The abstraction under the predicates `fail`, (`c==0`), (`c==1`) is shown in Figure 3(c). We see that the bounds on the maximum probability for the initial state have tightened from $[0.1, 1]$ to $[0.19, 1]$. A further refinement (on the same control-flow vertex) would result in an abstraction equivalent to the original MDP, yielding exact bounds $[0.19, 0.19]$.

Extensions. We investigate several extensions to the refinement loop.

Refinable state selection. Our refinement scheme can be applied to any refinable state $\langle v^\star, a^\star \rangle$. Hence, we consider two heuristics for choosing a refinable state: "maximum error" (pick a state with the greatest difference in lower and upper bounds, aiming to refine the abstraction where it is least precise); "nearest" (pick a state closest to the initial states). In addition, since model checking an abstract MDP (which determines the refinable states) is relatively expensive, we consider refining *multiple states* within a single iteration of the refinement loop.

Avoiding unreachable states. Although a refinable state $\langle v^\star, a^\star \rangle$ is always reachable in the abstract MDP, there is no guarantee that any concretisation of $\langle v^\star, a^\star \rangle$ is reachable in the concrete MDP. Hence, refining $\langle v^\star, a^\star \rangle$ could add unnecessary complexity to the abstraction. To avoiding this, we employ *spurious path removal*: we find an abstract path to $\langle v^\star, a^\star \rangle$ and use SAT-based symbolic simulation to check if a concretisation of the path exists. If not, in addition we use conventional weakest precondition-based refinement to eliminate the path.[1]

Predicate initialisation. Conventional (non-probabilistic) CEGAR can be used to check the existence of a path to the target. The predicates generated during this process are likely to form a subset of those found by our refinement approach and can potentially be discovered more efficiently in this fashion. Hence, we consider employing existing efficient CEGAR tools to generate an initial set of predicates.

Predicate localisation. It is well known that successful implementations of predicate abstraction compute abstractions efficiently because they keep the number of relevant predicates small, e.g. by exploiting locality [28]. We apply similar ideas to our approach, only adding discovered predicates to locations where they are required, based on a backwards control-flow traversal from the refinable state. This also allows us to take an incremental approach to building the abstraction, reusing the previous abstraction for locations with no new predicates.

6 Implementation and Results

We have built a complete implementation of the techniques described. Model extraction from C code is done using an extension of GOTO-CC [23]. Predicate abstraction was implemented using components from the SATABS tool [29] and

[1] This approach cannot guarantee to detect if $\langle v^\star, a^\star \rangle$ is unreachable, since doing so amounts to fully verifying a safety property with conventional CEGAR tools.

ping	A	"max. probability of not receiving a reply to an echo request"
	B	"max. prob. of establishing connectivity with packet loss following two requests"
	C	"max. expected number of echo requests required to establish connectivity"
tftp	A	"max. probability of establishing a write request"
	B	"max. probability of successfully transferring some file data"
	C	"max. expected amount of data that is sent before timeout"
her	A	"min. probability of terminating in a stable state"
	B	"max. expected number of rounds before termination"
zcnf	A	"min. probability of configuring with a fresh IP"
	B	"max. expected number of probes"
brp	A	"max. probability of the receiving nothing while a chunk was sent"
	B	"max. probability of the sender reporting uncertainty"

Fig. 4. List of properties verified.

MiniSAT SAT solver. Model checking of stochastic games is done with extensions of the symbolic engines of the probabilistic model checker PRISM [30].

We illustrate the practicality of our approach by studying its performance on several case studies. We consider two networking utilities: an ICMP ping client and a TFTP client.[2] Both are approximately 1KLOC in size and feature complex programming constructs such as arrays, pointers and function pointers. Low-level kernel and networking functions are replaced with stubs whose behaviour is either probabilistic (e.g. opening a socket) or nondeterministic (e.g. user input). We also consider ANSI-C versions of several protocols used as probabilistic model checking benchmarks: Herman's self-stabilisation (from APEX [17]), Zeroconf and the Bounded Retransmission Protocol. All programs are available[3] and the properties verified for each are listed in Figure 4.

We ran experiments for several different configurations of the options described in the previous section ("maximum error" and "nearest" refinable state selection; with and without spurious path removal and predicate initialisation). Table 1 presents detailed statistics for the fastest verification run on each example; Table 2 compares the different configurations. All experiments were run on an Intel Core Duo 2 (T7200) with 2GB RAM. The CEGAR loop terminated when the (relative) error was below ε=10^{-4}. All timings are in seconds.

Overall performance. The results demonstrate that our method verifies a wide range of programs and quantitative properties in an efficient and fully automatic manner. This is particularly impressive for the more complex ping and TFTP utilities. The tables show that the number of refinement iterations and predicates are relatively low. The difference between the total and average numbers of predicates indicates that the use of predicate localisation is essential. With regards to timings (Table 1), we see that abstraction and refinement are in most cases efficient, whereas the model checking phase is the most expensive. One reason for this is that the numerical solution process is relatively expensive (compared to the model checking required in conventional, non-probabilistic CEGAR) and is performed twice (for the lower and upper bound). Also, due to predicate localisation, abstractions can be constructed incrementally.

---

[2] Based on based on GNU Inetutils 1.5 and TFTP-HPA 0.48, respectively.

[3] All programs are available at `www.prismmodelchecker.org/files/vmcai09/`.

		refinement iterations	predicates total (avg.)	timing breakdown init.	abstr.	check	refine	total time
ping	A	4	33 (7.82)	56%	15%	27%	2%	**15.5**
	B	31	45 (9.27)	-	48%	45%	7%	**87.2**
	C	12	16 (2.73)	-	17%	68%	15%	**5.37**
tftp	A	11	39 (10.7)	32%	15%	48%	5%	**57.8**
	B	28	51 (12.0)	-	46%	45%	9%	**96.5**
	C	22	35 (9.22)	-	17%	75%	8%	**64.4**
her	A	18	24 (7.08)	-	17%	82%	1%	**33.5**
	B	2	40 (11.1)	7%	2%	91%	<1%	**259**
zcnf	A	9	11 (3.94)	-	9%	85%	6%	**1.97**
	B	9	10 (3.81)	4%	10%	77%	9%	**1.43**
brp	A	5	6 (6.00)	-	50%	31%	19%	**0.71**
	B	7	7 (7.00)	-	44%	44%	12%	**1.34**

Table 1. Experimental results: detailed results for fastest verification run.

Extensions. Table 2 shows the performance of the configurable options of our implementation. [4] For *refinable state selection*, although neither policy for choosing a refinable state is consistently better, "nearest" seems the sensible default as there are several cases where it is significantly faster. In particular, the "maximum error" policy for property B of the ping utility takes over an hour due to repeatedly choosing refinable states with no reachable concretisations. This problem is successfully resolved using *spurious path removal*. However we see that, in several cases, this produces a significant slow-down. This is because, although symbolic simulation itself is relatively fast, the predicates it adds result in slower model checking times. The situation is worse for the "maximum error" policy as the generated paths are longer and give more predicates. Employing *predicate initialisation* (through SATABS), increases efficiency on several examples. Often in cases where it performs best (e.g. property A of the ping and TFTP utilities), there are relatively few paths to the target state, so refining based on a single path is productive. On examples with a large number of such paths, using the game-based refinement alone performs better. In particular, for property B of BRP, predicate initialisation is very slow because the target is only reachable through very long paths but our method only needs to consider a small number of loop iterations for sufficiently tight bounds.

Related tools. Finally, we briefly compare our implementation with some related tools. Although a direct comparison with [5] is not possible (due to the difference in input language), we note that property A of BRP (called p4 in [5]) takes under a second here and about 5 seconds in [5] on a comparable machine. Also, we obtain sufficiently tight bounds discovering 6 predicates, compared to 25 in [5]. In [17], the Herman case study is tested on the APEX tool but the focus is different (contextual equivalence) and no times are given. PRISM [30] can be applied to the last three case studies and is faster, but only by using manually constructed abstractions. The more complex programs, the ping and TFTP utilities, are significantly beyond the scope of PRISM.

[4] Since refining multiple refinable states did not yield significant improvements in performance, we have omitted the results for this extension.

		default				spurious path removal				predicate initialisation			
		max error		nearest		max error		nearest		max error		nearest	
		pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time	pred. (avg.)	total time
ping	A	33 (6.9)	18.4	33 (6.3)	16.6	34 (6.4)	41.2	33 (6.3)	28.8	33 (7.8)	**15.5**	33 (7.8)	15.5
	B	-	-	45 (9.3)	**87.2**	54 (12)	803	56 (12)	627	54 (13)	305	53 (13)	301
	C	16 (2.7)	**5.37**	20 (3.2)	7.97	17 (3.1)	6.92	22 (3.6)	15.1	19 (3.3)	6.59	19 (3.3)	7.39
tftp	A	41 (11)	95.4	41 (11)	111	52 (11)	262	52 (12)	142	39 (11)	64.3	39 (11)	**57.8**
	B	51 (12)	173	51 (12)	**96.5**	62 (13)	1,680	62 (13)	227	56 (15)	194	56 (15)	161
	C	35 (8.4)	69.0	35 (9.2)	**64.4**	46 (8.9)	888	42 (8.8)	81.7	37 (9.2)	85.1	37 (10)	86.3
her	A	25 (6.9)	49.9	24 (7.1)	**33.5**	32 (7.6)	215	26 (6.6)	66.1	40 (11)	36.6	40 (11)	36.3
	B	34 (8.9)	520	27 (7.8)	751	39 (10)	974	32 (8.4)	619	40 (11)	**259**	40 (11)	263
zcnf	A	11 (3.9)	**1.97**	11 (3.9)	1.98	11 (3.9)	2.07	11 (3.9)	2.12	11 (4.1)	2.10	11 (4.1)	2.17
	B	10 (3.8)	1.44	10 (3.8)	1.45	10 (3.8)	1.51	10 (3.8)	1.54	10 (3.8)	**1.43**	10 (3.8)	1.54
brp	A	7 (7.0)	0.92	6 (6.0)	0.72	8 (8.0)	0.91	6 (6.0)	**0.71**	15 (15)	3.53	15 (15)	3.53
	B	7 (7.0)	**1.34**	10 (10)	2.12	27 (27)	6.98	12 (12)	2.29	-	-	-	-

Table 2. Experimental results: comparison of techniques. Timeout '-' is 1 hour.

7 Conclusions

We have presented a novel abstraction-refinement method for verification of software with probabilistic behaviour. Our approach uses two-player stochastic games, SAT-based predicate abstraction and probabilistic model checking. The use of game-based abstractions allows us to compute lower and upper bounds on quantitative properties, which form the basis of our abstraction-refinement loop. We have demonstrated the applicability of our approach by successfully verifying a selection of case studies, including several complex programs well beyond the reach of state-of-the-art probabilistic model checkers such as PRISM.

We plan to extend this work in several directions. These include investigating the use of imprecise abstractions, an approach frequently take in conventional software verification to improve efficiency, and developing techniques to handle probabilistic choices over larger ranges. We also hope to improve the way in which loops are dealt with. Currently, the abstractions we construct include an explicit representation of the loop, which is required to compute, for example, the probability of the loop terminating. We plan to investigate the use of existing techniques such as ranking functions to improve efficiency in this area.

Acknowledgments. The authors are supported in part by EPSRC grants EP/D07956X and EP/D076625. We would also like to thank Daniel Kroening for his advice and support regarding SATABS.

References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Proc. CAV'97. LNCS 1254, Springer (1997)
2. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proc. CAV'00. Volume 1855., Springer (2000)
3. Kwiatkowska, M., Norman, G., Parker, D.: Game-based abstraction for Markov decision processes. In: Proc. QEST'06, IEEE (2006)
4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C programs using SAT. FMSD **25**(2-3) (2004) 105–127

5. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Proc. CAV'08. LNCS 5123, Springer (2008)
6. D'Argenio, P., Jeannet, B., Jensen, H., Larsen, K.: Reachability analysis of probabilistic systems by successive refinements. In: Proc. PAPM/PROBMIV'01. LNCS 2165, Springer (2001)
7. Han, T., Katoen, J.P.: Counterexamples in probabilistic model checking. In: Proc. TACAS'07. LNCS 4424, Springer (2007)
8. de Alfaro, L., Roy, P.: Magnifying-lens abstraction for Markov decision processes. In: Proc. CAV'07. LNCS 4590, Springer (2007)
9. Roy, P., Parker, D., Norman, G., de Alfaro, L.: Symbolic magnifying lens abstraction in Markov decision processes. In: Proc. QEST'08, IEEE (2008)
10. Chatterjee, K., Henzinger, T., Jhala, R., Majumdar, R.: Counterexample-guided planning. In: Proc. UAI'05. (2005)
11. McIver, A., Morgan, C.: Abstraction, refinement and proof for probabilistic systems. Springer (2004)
12. Huth, M.: On finite-state approximants for probabilistic computation tree logic. Theoretical Computer Science **346**(1) (2005) 113–134
13. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: Game-based probabilistic predicate abstraction in PRISM. In: Proc. QAPL'08. (2008)
14. Pierro, A.D., Wiklicky, H.: Concurrent constraint programming: Towards probabilistic abstract interpretation. In: Proc. PPDP'00, ACM Press (2000)
15. Monniaux, D.: Abstract interpretation of programs as Markov decision processes. Science of Computer Programming **58**(1-2) (2005) 179–205
16. Smith, M.: Probabilistic abstract interpretation of imperative programs using truncated normal distributions. In: Proc. QAPL'08. (2008)
17. Legay, A., Murawski, A., Ouaknine, J., Worrell, J.: On automated verification of probabilistic programs. In: Proc. TACAS'08. LNCS 4963, Springer (2008)
18. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. QEST'06, IEEE (2006)
19. Kemeny, J., Snell, J., Knapp, A.: Denumerable Markov Chains. Springer (1976)
20. Shapley, L.: Stochastic games. Proc. Nat. Acad. Science **39** (1953) 1095–1100
21. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis (1997)
22. Condon, A.: On algorithms for simple stochastic games. Advances in computational complexity theory **13** (1993) 51–73
23. GOTO-CC. http://www.verify.ethz.ch/goto-cc/
24. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. PLDI'01. (2001)
25. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5) (1994) 1512–1542
26. Kattenbelt, M., Kwiatkowska, M., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. Technical Report RR-08-06, Oxford University Computing Laboratory (2008)
27. Jhala, R., McMillan, K.: A practical and complete approach to predicate refinement. In: Proc. TACAS'06. LNCS 3920, Springer (2006)
28. Henzinger, T., Jhala, R., Majumdar, R., McMillan, K.: Abstractions from proofs. In: Proc. POPL'04, ACM Press (2004)
29. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Proc. TACAS'05. LNCS 3440, Springer (2005)
30. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Proc. TACAS'06. LNCS 3920, Springer (2006)