Probabilistic Model Checking Modulo Theories

Björn Wachter, Lijun Zhang, Holger Hermanns

Department of Computer Science, Saarland University, Germany
{bwachter,zhang,hermanns}@cs.uni-sb.de

***Abstract*— Probabilistic models are widely used to analyze embedded, networked, and more recently biological systems. Existing numerical analysis techniques are limited to finite-state models and suffer from the state explosion problem. As a consequence, the user often has to manually abstract the intended model to get a tractable one. To this end, we propose the predicate abstraction model checker PASS which automates this process. We leverage recent advances in automatic theorem proving to compute tractable finite-state models. Experiments show the feasibility of our approach. To the best of our knowledge, this is the first time that properties of probabilistic infinite-state models have been verified at this level of automation.**

I. Introduction

Probabilistic models are widely used to analyze and verify systems that exhibit "quantified uncertainty", such as embedded, networked, randomized, and biological systems. The semantic model for these systems are Markov chains or Markov decision processes, depending on whether the systems are sequential or parallel. We here consider homogeneous discrete-time Markov chains (MCs) and Markov decision processes (MDPs). Properties of these systems can be specified by formulas in temporal logics such as PCTL [1], [2]. For instance "the probability to reach a set of bad states is at most 3%" can be expressed in PCTL. Model checking algorithms for these logics have been devised mainly for finite-state MCs [1] and MDPs [2], [3]. Often, modelers need to manually extract a manageable model from a system description because the original model cannot be handled due to state explosion or because it is infinite-state, e.g. due to unbounded arithmetic variables or queues.

Today, predicate abstraction [4] is one of the most widely applied methods for the systematic abstraction of *non-probabilistic* systems. The idea of predicate abstraction is to map concrete states to abstract states according to their valuation under a finite set of predicates ("Boolean expressions"). The predicates induce an 'abstract' model that is submitted to a model checker.

We propose *predicate abstraction* for the analysis of probabilistic models with an infinite or very large finite state space. To this end, we have developed a theoretical framework for predicate abstraction of probabilistic models and we have implemented the predicate abstraction model checker PASS.

PASS supports probabilistic programs in a guarded command language with unbounded integers and unbounded reals. The language is similar to the input language of the finite-state probabilistic model checker PRISM [5]. However, PRISM only supports integer variables *from a finite interval*. Given a program and a set of predicates, PASS computes a finite-state program that approximates the input program. The abstraction preserves the logic *safe PCTL*, a fragment of PCTL [6]. A typical query in safe PCTL is to ask if the probability to reach bad states is below a certain threshold (just like the example formula stated above). One can thus guarantee an upper bound on the probability that something harmful will happen.

When applying predicate abstraction to infinite-state probabilistic models several challenges arise:

- As opposed to abstraction techniques for finite-state probabilistic models [7], [8], [9], [10], computing abstractions of infinite-state models typically requires reasoning at the language level, i.e. solving satisfiability problems in logics in which the program variables live. Richer logics than propositional logic are needed because models may contain integer or real variables. Therefore, predicate abstraction tools employ automatic theorem provers. So far there has been little work on the use of *automatic* theorem proving for probabilistic models.
- Compared to predicate abstraction for non-probabilistic models [4], probabilistic models have a transition structure that makes the abstraction process computationally more complex. In order to reduce the cost of theorem prover calls while computing abstractions, predicate abstraction model checkers employ heuristics [11]. However, probabilistic models require dedicated heuristics.

We show how predicate abstraction for infinite-state probabilistic models can be implemented by employing an automatic theorem prover. Further, we give heuristics that make the abstraction process cheaper. As an automatic theorem prover, we use the SMT solver Yices [12] based on the DPLL(T) paradigm [13].

SMT and DPLL(T). SMT (Satisfiability Modulo Theories) can be seen as an extension of propositional satisfiability with richer background theories such as linear integer arithmetic (LIA). The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a complete, backtracking-based algorithm for deciding the satisfiability of propositional logic formulas in conjunctive normal form, i.e. for solving the CNF-SAT problem. DPLL(T) [13] is a paradigm for completely automatic SMT solvers based on an interplay between a conflict-driven DPLL SAT solver [14] and dedicated solvers for special theories T.

Let us sketch how a DPLL(LIA) solver checks satisfiability

This work is supported by the NWO-DFG bilateral project VOSS and by the DFG as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

of a formula like $x < y - 10 \wedge y > 100 \Rightarrow x < 110$. First it builds a propositional formula that replaces the theory constraints $x < y - 10, y > 100, x < 110$ with Boolean variables $\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3$. This resulting Boolean formula $\mathtt{b}_1 \wedge \mathtt{b}_2 \Rightarrow \mathtt{b}_3$ is the propositional backbone of the original formula. The backbone is submitted to the integrated DPLL SAT solver. This yields a propositional assignment which is checked against the theory of integers. If the propositional assignment is incompatible, a lemma is added that excludes it. This process is continued until a compatible assignment has been found or all possible propositional assignments have been explored.

In PASS, we have integrated the DPLL(T) SMT solver Yices [12] which won all categories of the SMT competition in 2006. Yices, like other SMT solvers, combines different theories by means of a Nelson-Oppen approach [15], such as linear integer arithmetic (LIA), linear real arithmetic (LRA), bit-vectors (BV), and arrays (AR). We anticipate that this admits to extend PASS to a very rich modeling language supporting data structures such as arrays, queues and lists.

A. Contributions.

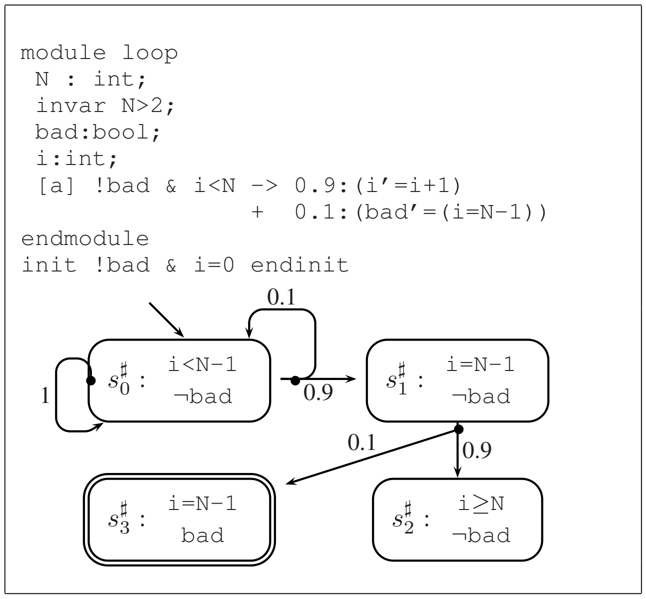
In brief, our contributions are the following

- the *first practical application of predicate abstraction* to (infinite-state) probabilistic models
- an *optimized encoding* of the abstract model construction in terms of SMT and an *implementation* demonstrating its efficiency and precision on practical examples
- the analysis of non-deterministic probabilistic infinite-state models with *unbounded parameters*

We present a predicate abstraction framework for probabilistic programs. A probabilistic program extends discrete-time PRISM models with unbounded integer and real variables. The semantics of a program is an infinite-state probabilistic automaton. Its abstraction, a finite-state probabilistic automaton, preserves the safe fragment of PCTL. We have implemented the predicate abstraction tool PASS and used it to verify a parametrized version of the Bounded Retransmission Protocol (BRP) where the file size and maximal number of retransmissions are integer parameters. The implementation leverages recent advances in automatic theorem proving and features optimizations that speed up the abstraction process by a smart encoding of queries to the SMT solver. Although our method only guarantees *upper bounds* on probabilities in general, probabilities obtained for BRP are *tight* upper bounds for all properties considered in previous case studies.

B. Outline.

Section II illustrates the general idea with a motivating example. In Section III, we introduce background on probabilistic programs, probabilistic automata, and the logic PCTL. We focus on predicate abstraction of probabilistic programs in Section IV. Our implementation is described in Section V. Section VI reports about experiments with PASS, in particular, how we verified the Bounded Retransmission Protocol. We discuss related work in Section VII. Section VIII concludes the paper.



```
module loop
 N : int;
 invar N>2;
 bad:bool;
 i:int;
 [a] !bad & i<N -> 0.9:(i'=i+1)
               +  0.1:(bad'=(i=N-1))
endmodule
init !bad & i=0 endinit
```

Fig. 1. Motivating Example: Loop program and its abstraction with respect to predicates $\mathtt{i} < \mathtt{N} - 1$, $\mathtt{i} = \mathtt{N} - 1$, bad.

II. Motivating Example

To provide some intuition of what we are aiming at, we first consider the example program loop shown in the upper part of Figure 1. We want to obtain an upper bound on the probability to reach a bad state, i.e. a state in which Boolean variable bad is true. The lower part of Figure 1 shows a finite-state probabilistic automaton, actually an abstraction of the behavior of loop. Here, an abstract state is a truth assignment to Boolean variables $\mathtt{b}_0, \mathtt{b}_1, \mathtt{b}_2$ which stand for predicates i<N-1, i=N-1, and bad, respectively. To obtain an upper bound of reaching a bad state in the original program, we compute the probability to reach an abstract state in which $\mathtt{b}_2$ ($\mathtt{b}_2$ corresponds to bad) is true. This probability bound is 0.1, and this turns out to be tight: the probability in the concrete program is also 0.1.

Writing abstract states as bit-vectors - the i-th component stands for $\mathtt{b}_i$, the reachable abstract states are: $s_0^\sharp = (1, 0, 0)$, $s_1^\sharp = (0, 1, 0), s_2^\sharp = (0, 0, 0)$ and $s_3^\sharp = (0, 0, 1)$. The initial state is $s_0^\sharp$ as it represents a concrete initial state.

Probabilistic transitions consist of a state and a probability distribution over successor states. Abstract transitions are 'may'-transitions, i.e. there is a transition between abstract states whenever there exists a corresponding concrete transition. Figure 2 illustrates how transitions of the concrete execution of program loop give rise to abstract may-transitions using an execution path of concrete states s_0, s_1, s_2. The two dashed boxes represent the abstract states $s_0^\sharp$ and $s_1^\sharp$, respectively, as shown in Figure 1. One abstract transition is a self-loop: corresponding transitions remain within $s_0^\sharp$, e.g. τ_1 is such a transition. Since all states in the distribution lead to abstract state $s_0^\sharp$, we sum up 0.9 and 0.1 to 1. The other abstract transition goes to $s_0^\sharp$ with probability 0.1 and to $s_1^\sharp$ with 0.9. A corresponding concrete transition is τ_2.

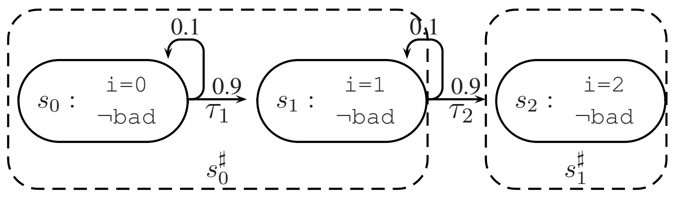


Fig. 2. Example of how abstraction transitions are determined. For the three depicted concrete states s_0, s_1, s_2, we have $\mathtt{N} = 3$. The dashed boxes represent abstract states.

III. Preliminaries

A. Probabilistic Programs

As indicated in the motivating example, we model systems by probabilistic programs in a guarded command language. For conciseness, the language used in our exposition lacks modules, although PASS supports modules in the style of PRISM. Apart from being less convenient, the simpler language is equally expressive[1], retaining all essential aspects such as variables over infinite domains.

We fix a finite set of program variables $\mathtt{X}$ and a finite set of actions Act. Variables are typed in a definition such as `i : int`. In addition to types, one can also specify invariants that constrain variable domains, and thus the state space. Arbitrary Boolean expression over the program variables are supported as invariants. We denote the set of expressions over the set of variables V by $Expr_V$ and we denote the set of Boolean expression over V by $BExpr_V$. In the motivating example, the variable N is used as a parameter that determines the length of the loop. Variable N is only read but not written and it is constrained by the invariant `invar N>2`. Using parameters, one can thus verify (infinite) families of programs in one verification run.

An *assignment* is a total function $E : \mathtt{X} \to Expr_{\mathtt{X}}$ that maps variables $\mathtt{x} \in \mathtt{X}$ to expressions $E(\mathtt{x})$. Given an expression $\mathtt{e} \in Expr_{\mathtt{X}}$ and an assignment E, we denote by $\mathtt{e}[\mathtt{X}/E(\mathtt{X})]$ the expression obtained from $\mathtt{e}$ by substituting each occurrence of a variable $\mathtt{x}$ with $E(\mathtt{x})$.

A *guarded command* $\mathtt{c}$ consists of an action name $a_{\mathtt{c}}$, a guard $\mathtt{g} \in BExpr_{\mathtt{X}}$ and assignments $E_1, ..., E_k$ weighted with probabilities $p_1, ..., p_k$ where $\sum_{i=1}^{k} p_i = 1$. We use the notation $\mathtt{X}' = E$ for the simultaneous assignment E to variables $\mathtt{X}$. Assignments are syntactically separated by a "+":

$$[a_{\mathtt{c}}]\ \mathtt{g} \to \mathtt{p}_1 : \mathtt{X}'{=}E_1 + \ \ldots\ + \mathtt{p}_k : \mathtt{X}'{=}E_k$$

Probability p_i is the probability that assignment E_i will be executed. A *program* $\mathtt{P} = (\mathtt{X}, \mathtt{I}, \mathtt{C})$ consists of a Boolean expression $\mathtt{I} \in BExpr_{\mathtt{X}}$ that defines the set of initial states and a set of guarded commands $\mathtt{C}$.

B. Probabilistic Automata

The semantics of a program is a probabilistic automaton [17]. A *state* over variables $\mathtt{X}$ is a type-consistent mapping of variables in $\mathtt{X}$ to their semantic domains. We denote the set of states by $\Sigma(\mathtt{X})$ or Σ for short and a single state by s. For an expression $\mathtt{e} \in Expr_{\mathtt{X}}$, we denote by $[\![\mathtt{e}]\!]_s$ its valuation in state s. The valuation of a Boolean expression $\mathtt{e}$ is a value $[\![\mathtt{e}]\!]_s \in \{0, 1\}$ (0 for "false", 1 for "true"). For a Boolean expression $\mathtt{e}$ and a state s, we shall write $s \vDash \mathtt{e}$ iff $[\![\mathtt{e}]\!]_s = 1$. Semantic brackets around a Boolean expression $\mathtt{e}$ without a subscript denote the set of states fulfilling $\mathtt{e}$, i.e. $[\![\mathtt{e}]\!] = \{s \in \Sigma \mid s \vDash \mathtt{e}\}$.

A *distribution* $\mu \in Distr_\Sigma$ is a total function $\mu : \Sigma \to [0, 1]$ such that $\mu(\Sigma) = \sum_{s \in \Sigma} \mu(s) = 1$. The set of states $Supp(\mu) = \{s \in \Sigma \mid \mu(s) > 0\}$ is called the support of μ.

Definition 1 (Probabilistic Automaton): A probabilistic automaton $\mathcal{M}$ is a tuple (Σ, I, Act, R) where Σ is a set of states, $I \subseteq \Sigma$ denotes the set of initial states, Act is the set of actions, and $R \subseteq \Sigma \times Act \times Distr_\Sigma$ is the probabilistic transition relation. A probabilistic automaton is called finite if Σ is finite. A path σ is an infinite sequence $(s_0, a_0, \mu_0), (s_1, a_1, \mu_1), \ldots$ starting with an initial state $s_0 \in I$, $(s_i, a_i, \mu_i) \in R$, and $s_{i+1} \in Supp(\mu_i)$ for all $i = 0, 1, \ldots$. Let $Path_\infty$ denote the set of all paths[2].

Labeled transition systems (LTSs), discrete-time Markov chains (MCs) and discrete-time Markov decision processes (MDPs) [18] are special cases of probabilistic automata. An LTS is a probabilistic automaton in which all distributions in the transition relation are trivial, i.e. their support size is one. An MC is a deterministic probabilistic automaton, i.e. for every state s there is at most one transition $(s, a, \mu) \in R$. An MDP is a probabilistic automaton where for each state each action label has at most one outgoing transition, i.e. for every pair $s \in \Sigma$ and $a \in Act$, there exists at most one μ with $(s, a, \mu) \in R$.

Program Semantics. First, we give the semantics of guarded commands. For a guarded command $\mathtt{c}$, fully written out as

$$[a_{\mathtt{c}}]\ \mathtt{g} \to \mathtt{p}_1 : \mathtt{X}'{=}E_1{+}\ldots{+}\mathtt{p}_k : \mathtt{X}'{=}E_k,$$

its semantics is exactly the set of transitions $[\![\mathtt{c}]\!]$ such that $(s, a_{\mathtt{c}}, \mu) \in [\![\mathtt{c}]\!]$ if $s \vDash \mathtt{g}$ and μ satisfies the following dependency, where $\{\!|\ldots|\!\}$ delimits a multiset:

$$\mu(s') = \sum \{\!|\ p_i \mid \forall \mathtt{x} \in \mathtt{X} :\ s'(\mathtt{x}) = [\![E_i(\mathtt{x})]\!]_s\ |\!\}\ .$$

The sum can be explained as follows: For a given state and command, different assignments in the command may lead to the same successor state s', possibly with the same probability. The probabilities are collected in a multiset, and accumulated in the distribution μ.

The semantics of a program $\mathtt{P} = (\mathtt{X}, \mathtt{I}, \mathtt{C})$ is the automaton $\mathcal{M} = (\Sigma, I, Act, R)$ with set of states $\Sigma = \Sigma(\mathtt{X})$, set of initial states $I = [\![\mathtt{I}]\!]$, set of actions $Act = \{a_{\mathtt{c}} \mid \mathtt{c} \in \mathtt{C}\}$, and transitions induced by the guarded commands $R = \bigcup_{\mathtt{c} \in \mathtt{C}} [\![\mathtt{c}]\!]$.

[1]Composition of modules can be reduced to the core language by flattening the module structure [16].

[2]If an automaton has deadlock states, one typically introduces self-loops. For the sake of simplicity, we have intentionally not drawn the self-loops in Figure 1.

C. Properties

We give a short introduction to the safe and live fragments [6] of the logic *probabilistic CTL* (PCTL) [1], [2]. We fix a finite subset AP of the set of Boolean expressions, called atomic propositions. Let $p \in [0,1]$ denote a real constant and $\unlhd \in \{<,>,\leq,\geq\}$ an inequality symbol. We consider the following fragment of PCTL:

$$\Phi := a \mid \neg a \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{\unlhd p}(\phi)$$

where ϕ is a PCTL path formula:

$$\phi := \mathcal{X}\Phi \mid \Phi \,\mathcal{U}\, \Phi \ .$$

$\mathcal{X}$ is the next operator and $\mathcal{U}$ the until operator. To interpret PCTL formulas, one needs a probability measure for path formulas, which requires a resolution of non-determinism. An adversary [19] is a resolution of non-determinism in an automaton that leads to a *deterministic* automaton in which path formulas have a probability measure. One argues about *extremal probability*, formally minimal and maximal probabilities among all adversaries.

In general, an adversary A of an automaton $\mathcal{M}$ is a function from paths to distributions. An adversary A is called *simple* if it only looks at the last state in a path, i.e. if it is a function $A : \Sigma \to Act \times Distr_\Sigma$. However, it suffices to consider *simple* adversaries, as extremal probabilities are already assumed among simple adversaries [2]. Given an automaton $\mathcal{M}$, a simple adversary A induces a MC $\mathcal{M}_A = (\Sigma, I, Act, R_A)$ where $R_A = \{(s,a,\mu) \in R \mid A(s) = (a,\mu)\}$. For a given state $s \in \Sigma$, a unique probability measure $Prob_s^{\mathcal{M}_A}$ can be constructed [3]. For a (measurable) set of paths $P \subseteq Path_\infty$, we denote its measure by $Prob_s^{\mathcal{M}_A}(P)$.

The probabilistic operator $\mathcal{P}$ admits to formulate bounds on the probability measure of a path formula. To simplify the presentation, we don't consider nested probabilistic operators.

If we restrict the probabilistic operator to upper probability bounds $\unlhd \in \{\leq,<\}$ throughout a formula, we arrive at a *safe PCTL* formula. Intuitively, we can ask if the probability to reach a set of bad states does not exceed a certain threshold. Taking $\unlhd \in \{\geq,>\}$ instead yields the *live fragment of PCTL*. Here the intuition is that the probability to reach a set of good states is guaranteed not to fall below a threshold.

Semantics. Let $\mathcal{M}$ be an automaton. The semantics of the atomic propositions, negation, conjunction and disjunction are defined as usual. We give the semantics of the probabilistic operator [2]. For an adversary A and a path formula ϕ, let

$$p_s^{\mathcal{M}_A}(\phi) = Prob_s^{\mathcal{M}_A}(\{\sigma \in Path_\infty \mid \sigma \text{ satisfies } \phi\}).$$

A state s satisfies the PCTL formula $\mathcal{P}_{\unlhd p}(\phi)$, denoted by $s \models \mathcal{P}_{\unlhd p}(\phi)$, iff $p_s^{\mathcal{M}_A}(\phi) \unlhd p$ for *every* adversary A. To check if a state s satisfies a safe PCTL formula $\mathcal{P}_{\leq p}(\phi)$, it suffices to compute

$$p_{max}^s(\phi) = \sup\{p_s^{\mathcal{M}_A}(\phi) \mid A \text{ is an adversary of } \mathcal{M}\}$$

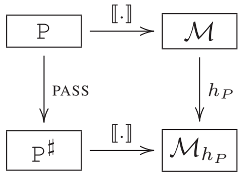
which is the maximum probability over all adversaries for the set of paths starting from s and satisfying the path formula ϕ. Therefore, $s \models \mathcal{P}_{\leq p}(\phi)$ iff $p_{max}^s(\phi) \leq p$ (similar for $<$). Let $p_{max}^{\mathcal{M}}(\phi) = \sup\{p_{max}^s(\phi) \mid s \in I\}$. We say that the automaton $\mathcal{M}$ satisfies $\mathcal{P}_{\leq p}(\phi)$ (similar for $<$), denoted by $\mathcal{M} \models \mathcal{P}_{\leq p}(\phi)$, iff $p_{max}^{\mathcal{M}}(\phi) \leq p$.

IV. Predicate Abstraction

Predicates are Boolean expressions over the program variables. A predicate φ stands for the set of states satisfying it, namely $[\![\varphi]\!]$. During this section, we fix a set of predicates $P = \{\varphi_1, \ldots, \varphi_n\}$. We show how a set of predicates P determines an abstract probabilistic automaton.

Abstract Probabilistic Automaton. The set P induces an equivalence relation over states, a homomorphism and an abstract probabilistic automaton that is the quotient of the concrete probabilistic automaton (all of these objects can be defined in terms of each other). More precisely, two states in Σ are *equivalent* if they satisfy the same set of predicates in P. The equivalence classes partition the states into disjoint sets characterized by which predicates hold and which don't. An equivalence class can therefore be represented by a bit vector of length n. We call such a bit-vector an *abstract state*. We define a function that maps a state s to an abstract state $h_P(s) = ([\![\varphi_1]\!]_s, \ldots, [\![\varphi_n]\!]_s)$. Function h_P induces a quotient automaton $\mathcal{M}_{h_P}$. The transitions of the quotient automaton are chosen such that h_P preserves the transition structure. Therefore, we denominate the function h_P as a homomorphism[3].

The atomic propositions, i.e. the Boolean expressions appearing in the PCTL formula we want to check, are a subset of the predicates, i.e. $AP \subseteq P$. As we will see later, taking the quotient this way preserves safe PCTL. Thus one can submit the quotient automaton to a finite-state model checker, instead of the original, possibly infinite-state model. If the model checker confirms that the property holds for the quotient automaton, we can safely conclude that it holds for the original model as well. The converse, however, is not true in general. Technically PASS transfers the quotient automaton to the finite-state model checker in the form of a *Boolean program* $\mathtt{P}^\sharp$ whose semantics $[\![\mathtt{P}^\sharp]\!]$ is exactly the quotient automaton $\mathcal{M}_{h_P}$ of the original program semantics. Formally, this means that the diagram in Figure 3 commutes. Note that the general idea of relating the original model with its quotient in terms of an adequate notion of simulation is quite common, see e.g. [7], [20], [22], [10]. Section IV-A customizes the quotient automata construction to our specific setting. While quotient automata are quite general, the ensuing Section IV-B is specific to our analysis method and explains a basic algorithm to obtain Boolean programs.



$$\begin{tikzcd}
\boxed{\mathtt{P}} \arrow[r, "{[\![\cdot]\!]}"] \arrow[d, "\text{PASS}"] & \boxed{\mathcal{M}} \arrow[d, "h_P"] \\
\boxed{\mathtt{P}^\sharp} \arrow[r, "{[\![\cdot]\!]}"] & \boxed{\mathcal{M}_{h_P}}
\end{tikzcd}$$

Fig. 3. Predicate Abstraction.

[3]The term "homomorphism" was used by Clarke [20] and Ball [11]. One might call function h_P an "abstraction function". Then, however, there would be a name clash with the terminology of abstract interpretation where the term "abstraction function" is reserved for a different concept [21].

A. Quotient Automaton

A *homomorphism* is a total function that maps concrete states Σ of the infinite-state automaton to a finite set of abstract states $\Sigma^\sharp$. A homomorphism induces an equivalence relation on states in which two states are equivalent if they map to the same abstract state. Therefore, the probabilistic automaton obtained by lumping together equivalent states is called *quotient automaton*. We define a quotient automaton such that *safe PCTL* properties are preserved, i.e. we guarantee that the quotient is a sound abstraction of the original automaton.

Definition 2 (Quotient Automaton): Let $\mathcal{M}$ be an automaton. Further, let $\Sigma^\sharp$ be a finite set of *abstract states* and $h \in \Sigma \to \Sigma^\sharp$ a homomorphism. The homomorphism h induces an automaton $\mathcal{M}_h$, called the quotient of $\mathcal{M}$ (under h), with state set $\Sigma^\sharp$, the same set of actions, initial states $I_h = \{h(s) \mid s \in I\}$, and transitions $R_h = \{(h(s), a, h(\mu)) \mid (s, a, \mu) \in R\}$ where we define the quotient $h(\mu) \in Distr_{\Sigma^\sharp}$ of a distribution μ by $h(\mu)(s^\sharp) = \sum\{\mu(s) \mid s \in \Sigma : \; h(s) = s^\sharp\}$.
Next, we show that the quotient automaton $\mathcal{M}_h$ simulates the automaton $\mathcal{M}$. Thereby, we use the concept of strong simulation and property preservation results that have been described by Segala & Lynch [19]. We need the notion of a weight function to define strong simulation:
Weight Function. Let $\mu_1 \in Distr_{\Sigma_1}, \mu_2 \in Distr_{\Sigma_2}$ distributions. For a relation $H \subseteq \Sigma_1 \times \Sigma_2$ we define the weight function for (μ_1, μ_2) w.r.t. H as a function $\Delta : \Sigma_1 \times \Sigma_2 \to [0, 1]$ such that

- $\Delta(s, s') > 0$ implies $H(s, s')$,
- $\mu_1(s) = \sum_{s' \in \Sigma_2} \Delta(s, s')$ for $s \in \Sigma_1$, and
- $\mu_2(s') = \sum_{s \in \Sigma_1} \Delta(s, s')$ for $s' \in \Sigma_2$.

We write $\mu_1 \sqsubseteq_H \mu_2$ iff there exists a weight function for μ_1 and μ_2 w.r.t. H.
Strong Simulation. Given two probabilistic automata $\mathcal{M}_1 = (\Sigma_1, I_1, Act, R_1)$ and $\mathcal{M}_2 = (\Sigma_2, I_2, Act, R_2)$, we say that automaton $\mathcal{M}_2$ strongly simulates automaton $\mathcal{M}_1$, denoted by $\mathcal{M}_1 \preceq \mathcal{M}_2$, iff there exists a relation $H \subseteq \Sigma_1 \times \Sigma_2$ such that (1) for $s_1 \in I_1$ there exists $s_2 \in I_2$ with $(s_1, s_2) \in H$. (2) for all $(s_1, s_2) \in H$ if there exists $(s_1, a, \mu_1) \in R_1$, there exists a distribution $\mu_2 \in Distr(\Sigma_2)$ such that $(s_2, a, \mu_2) \in R_2$ and $\mu_1 \sqsubseteq_H \mu_2$.

Lemma 1 (Simulation): Let $\mathcal{M}$ be an automaton and its quotient $\mathcal{M}_h$ as defined in Definition 2. Automaton $\mathcal{M}_h$ strongly simulates automaton $\mathcal{M}$, i.e. $\mathcal{M} \preceq \mathcal{M}_h$. The simulation relation is given by $H = \{(s, s^\sharp) \mid h(s) = s^\sharp\}$.
It was shown in [19] that strong simulation preserves safe PCTL. We re-state this result in our framework:

Lemma 2 (Preservation [19]): Let $\mathcal{M}, \mathcal{M}'$ be automata such that $\mathcal{M}'$ strongly simulates $\mathcal{M}$ and let Φ be a safe PCTL formula. Then if $\mathcal{M}'$ satisfies Φ, $\mathcal{M}$ also satisfies Φ.
As a corollary of Lemmas 1 and 2, we finally obtain the following *soundness theorem* that allows us to check a safe PCTL property on the quotient automaton $\mathcal{M}_h$:

Theorem 1 (Soundness): Let $\mathcal{M}$ be an automaton and $\mathcal{M}_h$ its quotient as described in Lemma 1. If $\mathcal{M}_h$ satisfies a safe PCTL formula Φ, $\mathcal{M}$ also satisfies Φ.

B. Boolean Programs

For a program $\mathtt{P} = (\mathtt{X}, \mathtt{I}, \mathtt{C})$, we compute a Boolean program $\mathtt{P}^\sharp = (\mathtt{B}, \mathtt{I}^\sharp, \mathtt{C}^\sharp)$. The variables $\mathtt{B}$ of the Boolean program are the Boolean variables $\mathtt{b}_1, \ldots, \mathtt{b}_n$ corresponding to the predicates $\varphi_1, \ldots, \varphi_n$. The abstract initial condition $\mathtt{I}^\sharp$ and the abstract guarded commands $\mathtt{C}^\sharp$ have to be computed. We abstract a program by abstracting its initial condition and each of its guarded commands.
Literals, Minterms. Let $\mathtt{b}$ be a Boolean variable. A literal is either the expression $\mathtt{b}$ or its negation $\neg\mathtt{b}$. Let $\nu \in \{0, 1\}$. We denote by $\mathtt{b}^\nu$ the literal $\mathtt{b}$ if $\nu = 1$ and its negation $\neg\mathtt{b}$ if $\nu = 0$. A cube is a conjunction of literals. The empty cube corresponds to logical truth. A *minterm* (w.r.t. $\mathtt{B}$) is a complete cube $\bigwedge_{j=1}^{n} \mathtt{b}_j^{\nu_j}$, i.e. each Boolean variable appears exactly once in it, in positive or negated form.
Abstract Interpretation. Both abstract states and sets of abstracts states can be viewed as Boolean expressions over $\mathtt{B}$: an abstract state $s^\sharp$ is a minterm over $\mathtt{B}$, a set of abstract states is a disjunction of minterms (a sum of products). We define a function $\mathcal{E}_P(\mathtt{e}) := \mathtt{e}[\mathtt{b}_i/\varphi_i \mid i \in \{1, ..., n\}]$ that replaces the Boolean variables in a Boolean expression $\mathtt{e} \in BExpr_{\mathtt{B}}$ with the corresponding predicates. It produces a symbolic representation $\mathcal{E}_P(\mathtt{e}) \in BExpr_{\mathtt{X}}$ (in terms of the program variables) of the concrete states represented by $\mathtt{e}$. For example, we have $\mathcal{E}_P(\mathtt{b}_i) = \varphi_i$. The concretization function maps sets of abstract states $S^\sharp$ to corresponding sets of concrete states:

$$\gamma_P : 2^{\Sigma^\sharp} \to 2^\Sigma, S^\sharp \mapsto [\![\mathcal{E}_P(S^\sharp)]\!].$$

Conversely, a set of concrete states S is mapped to a set of abstract states by the abstraction function:

$$\alpha_P : \; 2^\Sigma \to 2^{\Sigma^\sharp}, S \mapsto \{s^\sharp \mid [\![\mathcal{E}_P(s^\sharp)]\!] \cap S \neq \emptyset\}.$$

The pair (α_P, γ_P) forms a Galois connection [21]. Intuitively, this means that the abstraction function yields the most precise over-approximation possible with predicates P.
Weakest Preconditions. The construction of abstract guarded commands is based on weakest preconditions [23]. The weakest precondition $\mathtt{WP}_c(Q) = Q'$ of a Boolean expression Q with respect to a command $\mathtt{c}$ is the weakest Boolean expression (w.r.t. implication order) that guarantees Q to hold after executing command $\mathtt{c}$, typically this is written as a triple $\{Q'\}\mathtt{c}\{Q\}$. For an assignment $\mathtt{X}'{=}E$, the corresponding triple is $\{Q[\mathtt{X}/E(\mathtt{X})]\}\mathtt{X}'{=}E\{Q\}$, i.e. the weakest precondition of expression Q is obtained by substituting within Q the left-hand side variables with the right-hand side expressions yielding $Q[\mathtt{X}/E(\mathtt{X})]$. Therefore, given a set of states characterized by a Boolean expression P, satisfiability of the conjunction $P \wedge Q[\mathtt{X}/E(\mathtt{X})]$ guarantees the existence of a state transition from P to Q. We extend this concept to the case of abstraction of guarded commands with weighted alternatives by making Q a conjunction ranging over the k weighted alternative assignments of the command, as detailed out below. We abbreviate the weakest precondition of an expression e with respect to an assignment E as $\mathtt{WP}_E(e) = e[\mathtt{X}/E(\mathtt{X})]$.
Construction. The following simple construction gives a program $\mathtt{P}^\sharp$ that corresponds to the quotient automaton, in the

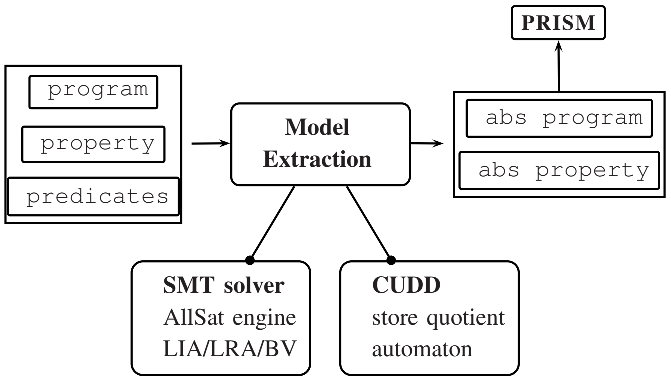


Fig. 4. Tool architecture.

sense that its semantics produces the precisely intended quotient automaton (cf. Lemma 3). The abstract commands are determined as follows: For each guarded command $[a_{\mathrm{c}}]\ \mathtt{g} \rightarrow \mathtt{p}_1 : \mathtt{X}'{=}E_1 + \ldots + \mathtt{p}_k : \mathtt{X}'{=}E_k$ and for all k-tuples of abstract states $({s_0}^\sharp, {s_1}^\sharp, ..., {s_k}^\sharp) \in {\Sigma^\sharp}^k$ such that the expression

$$\mathtt{g} \wedge \mathcal{E}_P({s_0}^\sharp) \wedge \bigwedge_{i=1}^{k} \mathtt{WP}_{E_i}(\mathcal{E}_P({s_i}^\sharp))$$

is satisfiable, introduce the following command:

$$[a_{\mathrm{c}}]\ {s_0}^\sharp \rightarrow \mathtt{p}_1 : \mathtt{B}'{=}s_1^\sharp + \ldots + \mathtt{p}_k : \mathtt{B}'{=}s_k^\sharp\ .$$

The initial condition of $\mathtt{P}^\sharp$ is given by the abstraction of the concrete initial states $\mathtt{I}^\sharp := \alpha_P([\![\mathtt{I}]\!])$. Note that we interpret the set $\alpha_P([\![\mathtt{I}]\!])$ of abstract states as a Boolean expression at this point.

Lemma 3 (Correctness): The semantics $[\![\mathtt{P}^\sharp]\!]$ of the Boolean program $\mathtt{P}^\sharp$, as given in the construction, generates the quotient automaton $\mathcal{M}_{h_P}$.

V. Implementation

This section describes the architecture and implementation of our tool PASS. we first present the overall architecture of the tool, and then focus on important details of the model extraction: its implementation with an SMT solver and several optimizations. These are later evaluated by experiments (cf. Section VI-B).

A. Tool Architecture

The architecture of PASS is depicted in Figure 4. PASS is written in C++, and wraps several other tools. For model extraction, it calls the SMT solver Yices and the CUDD package [24] via API functions. We provide a generic SMT solver interface, so that different SMT solvers can be integrated. In the end, the finite-state model checker PRISM is used to model check Boolean programs. We now describe the different phases in which PASS proceeds in more detail: *Phase A.1: Preprocessing.* Similar to PRISM, PASS supports modules and synchronous composition of commands. Before the model extraction phase, the module structure is flattened yielding a single global module. Asynchronous commands are collected, while commands from different modules that execute synchronously are multiplied out, very much as described in the PRISM manual [16]. *Phase A.2: Predicate Discovery.* Prior to the extraction of the abstract model we need to discover a set of predicates. We have not yet implemented an automatic refinement loop. Therefore, predicates are automatically extracted from the guards of the program and from properties. Furthermore, the user can provide predicates by hand. In order to ensure that there is no redundancy in the predicate set, e.g. such that one predicate is the exact negation of another, a redundancy check is done before adding a predicate to the predicate set. *Phase A.3: Model Extraction.* The model extraction phase takes the program and the predicate set as input and computes the quotient automaton. In subsection V-B, we describe how the model extraction is performed by means of an SMT solver. We also describe optimizations that exploit locality in the structure of the program and the predicates. As mentioned earlier, we seemingly discard structure by flattening modules. However, abstracting modules independently of each other is hardly feasible without loss of precision if predicates relate variables from different modules. The predicates have to be taken into account to perform precise 'local' abstraction. At the level of commands, this is done by the optimizations described in the next subsection.

During extraction, the quotient automaton is stored in BDDs to save memory. The BDDs are then used to synthesize a Boolean program.
Phase A.4: Synthesis of Boolean Program. PASS synthesizes a Boolean program, i.e. it computes an initial condition and a set of updates. The Boolean program is submitted to a finite-state model checker. Currently, we write the Boolean program to a file which is then read by the PRISM model checker. Furthermore, the property has to be translated, i.e. expressions over program variables have to be translated into expressions over the abstract program.

B. Model Extraction

The initial condition of the Boolean program can be generated by enumerating all abstract states (minterms) $s^\sharp = (\mathtt{b}_1, ...\mathtt{b}_n)$ such that the Boolean expression

$$\mathtt{I} \wedge \bigwedge_{j=1}^{n} [\mathtt{b}_j \Leftrightarrow \mathcal{E}_P(\mathtt{b}_j)]$$

is satisfiable. Abstract transitions are obtained in a similar way. As indicated in Lemma 3, the abstract transitions induced by a guarded command are determined by exactly those abstract states ${s_0}^\sharp = (\mathtt{b}_1^0, ...\mathtt{b}_n^0), {s_1}^\sharp = (\mathtt{b}_1^1, ...\mathtt{b}_n^1), ..., {s_k}^\sharp = (\mathtt{b}_1^k, ...\mathtt{b}_n^k)$ such that the following Boolean expression is satisfiable:

$$\mathtt{g} \wedge \bigwedge_{j=1}^{n} [\mathtt{b}_j^0 \Leftrightarrow \mathcal{E}_P(\mathtt{b}_j^0)] \wedge \bigwedge_{i=1}^{k} \bigwedge_{j=1}^{n} [\mathtt{b}_j^i \Leftrightarrow \mathtt{WP}_{E_i}(\mathcal{E}_P(\mathtt{b}_j^i))]\ .$$

AllSAT. We enumerate these abstract states and abstract transitions by using an SMT solver as an *AllSat* engine, i.e. an engine that enumerates all Boolean solutions. The solver runs in an incremental enumeration loop, i.e. facts learned in one run benefit ensuing runs [25]. When the solver finds a model, it is stored in a BDD and a blocking clause is added preventing the solver from enumerating it again. Computing the initial condition takes as many iterations as there are abstract initial states, i.e. at most 2^n. For a guarded command, the number of iterations is equal to the number of induced abstract transitions, i.e. at most $2^{(k+1)n}$. Note that these numbers represent the *worst-case*, the number of enumerations is solely determined by the number of existing solutions. We do *not* per se go through all $2^{(k+1)n}$ possibilities of potential transitions. Nevertheless, it pays off to simplify the Boolean expressions from which we obtain abstractions, in particular, the number of Boolean variables they contain. The initial condition is mostly not critical in practice, since it can be abstracted precisely by adding the predicates contained in it. Therefore, we focus on two optimizations that make computation of abstract commands more efficient.

The optimizations are related to *approximation techniques* for predicate abstraction that make computation of abstract transitions cheaper at the cost of losing precision, e.g. for software [11], and for hardware [26]. Tonetta & Sharygina [27] summarized approximation techniques in a unifying framework: the common idea is to ignore correlations between predicates. Typically, not all predicates have correlations and some correlations may be irrelevant to show a particular property. If existing correlations are ignored, this results in additional transitions and more nondeterminism in the abstract model compared to the quotient automaton.

Additional nondeterminism can put significant strain on a probabilistic finite-state model checker [28]. Therefore, and to obtain results that are more easily verifiable, we *currently* compute precise abstract transitions and defer an in-depth discussion of approximation. Our decomposition optimization uses a probabilistic extension of *predicate partitioning* [26], i.e. it only keeps track of correlations between predicates in the same partition. We statically precompute partitions of predicates between which no correlations exist. Any partition would yield a sound abstraction (in terms of simulation), but the proposed partitioning method guarantees that *full precision* is retained. A soundness proof of our method is given in the long version of the paper.

1) Optimization: Decomposition.: Programs we considered contain parallel assignments to many variables. Often, one can decompose a command into independent sub-commands assigning to different variables, abstract each sub-command separately, and finally combine the results. The overall number of SMT enumerations is the *sum* of the enumerations for the sub-commands rather than the *product*.

The sub-commands are determined by the predicates. For variables $V \subseteq \mathtt{X}$, let Mod_V be the set of predicates containing a variable from V. We partition the set of variables modified by the guarded command into disjoint sets $V_1, ..., V_l$ such that they affect pairwise disjoint sets of predicates Mod_{V_i}. Each V_i gives rise to a sub-command which is abstracted using the optimization described below.

Example 1: Given predicates, $x > 0$, $y > 0$, the following command `x>0->1: (x'=1)&(y'=1)` can be decomposed using the partition $V_1 = \{\mathtt{x}\}$, $V_2 = \{\mathtt{y}\}$ with $Mod_{V_1} = \{\mathtt{x>0}\}$ and $Mod_{V_2} = \{\mathtt{y>0}\}$. This yields two sub-commands: one `x>0->1:(x'=1)` pertaining to V_1 and the other `x>0 -> 1:(y'=1)` to V_2. If we add predicate `x<y`, we get $Mod_{V_1} \cap Mod_{V_2} = \{\mathtt{x<y}\}$ and cannot decompose.

2) Optimization: Relevant Predicates.: Mostly, only a fraction of all predicates is relevant when abstracting a command (or sub-command). Then it suffices to operate on a smaller subset of predicates/Boolean variables. We identify Boolean variables and predicates by indices from $\{1, \ldots, n\}$ where n is the number of predicates. We define (possibly intersecting) sets of indices G_1 and G_2 corresponding to:

G_1: predicates that may help determine (in)validity of another predicate in successor states, i.e. share variables with the guard or right-hand side expression

G_2: predicates whose valuation may be *affected* by a command, i.e. those containing a variable to which the command assigns a new value

Boolean variables whose index is not in G_2 retain their value. Further, the next values of Boolean variables with indices in G_2 are *only* influenced by the present value of Boolean variables with indices in G_1. Therefore, we can obtain the abstract transitions induced by a guarded command from the simplified expression:

$$\mathtt{g} \wedge \bigwedge_{j \in G_1} [\mathtt{b}_j^0 \Leftrightarrow \mathcal{E}_P(\mathtt{b}_j^0)] \wedge \bigwedge_{i=1}^{k} \bigwedge_{j \in G_2} [\mathtt{b}_j^i \Leftrightarrow \mathtt{WP}_{E_i}(\mathcal{E}_P(\mathtt{b}_j^i))] \,.$$

Example 2: Given predicates $\varphi_0 = x > 0, \varphi_1 = y > 0$, we will compute G_1 and G_2 for the command `[a] 1` $\rightarrow$ `1.0: x'=y`. Note that $1 \in G_1$, since validity of $y > 0$ at present determines if $x > 0$ holds afterwards, while $0 \notin G_1$. Further, $0 \in G_2$ as the command assigns to x, whereas $1 \notin G_2$ since the command has no effect on $y > 0$. Therefore, we have groups $G_1 = \{1\}$ and $G_2 = \{0\}$.

VI. Experiments

In this section, we report our experiences with the bounded retransmission protocol case study. Several benchmarks are used to evaluate the effect of the optimizations described above. The experimental results were obtained on a Linux PC (Ubuntu 6.10) with a Pentium 4 processor at 2.6Ghz and 1Gb of RAM, with execution time measured in seconds. For the benchmarks, we split up execution time into different quantities spent in consecutive phases of our method, abbreviated by A, B, and C:

A	abstraction: a Boolean program is extracted from the program by abstraction and written to disk (corresponding to phase A.1–A.4 of Subsection V-A)
B	building: PRISM reads the Boolean program from disk and builds the transition matrix MTBDD
C	checking: PRISM checks the properties

I_1: the sender reports an unsuccessful transmission but the receiver got a complete file
I_2: the sender reports a successful transmission but the receiver didn't get the complete file
1: the sender doesn't report a successful transmission
2: the sender reports an uncertainty on the success of the transmission
3: the sender reports an unsuccessful transmission after transmitting more than 8 chunks
4: the receiver doesn't receive any chunk of a file

Fig. 5. BRP Properties

Steps B and C both happen within PRISM, nevertheless we distinguish building from checking times. In our experiments, we employed the most recent version of PRISM (version 3.1) [16].

Concerning user interaction, most predicates are extracted automatically from the guards, initial condition and properties. In some cases, predicates were added manually (recall that automatic predicate discovery is future work).

A. BRP case study

The *Bounded Retransmission Protocol* (BRP) is a standard benchmark [29], [30], [7], [8], [16], in which a sender and a receiver exchange data via two lossy channels. The file to be transmitted is divided into N chunks, and is sent to the receiver chunk by chunk. BRP is based on the alternating bit protocol except that the number of retransmissions of a chunk is bounded by a number MAX. Therefore, the successful transmission of the whole file is not guaranteed and the protocol may abort the transfer.

Figure 5 recalls the properties checked in previous case studies using the tools RAPTURE [7], [8] and PRISM [16]. All properties are PCTL path formulas. Unlike PRISM and RAPTURE, we can check models for all parameters within very large or even infinite intervals. A comparison of our method and PRISM on finite instances is not very meaningful. Of course, the larger the parameter values the longer PRISM (on the original model) would take.

We first consider properties I_1 and I_2 which have probability zero for any parameter value checked in [7]. This suggests that their probability is zero for *any* parameter value. To obtain an upper bound on the probability for all parameter values, we treat parameters N and MAX as symbolic constants; more precisely, the parameters are integer program variables constrained by $N > 0$ and $MAX > 0$, respectively. We denote the resulting program by $\mathtt{P}[N > 0, MAX > 0]$. When checking them, phase A takes $0.26s$, phase B takes $60s$, and phase C takes $0.033s$[4]. The upper bound computed is zero, and hence I_1 and I_2 have indeed probability zero for *all possible file sizes* and for *any maximal number of retransmissions*.

[4]The bad states that are excluded by the properties are unreachable. PRISM computes reachable states as a part of the model construction.

Now, we deal with properties that are not invariant under all parameter values. For different properties and parameter ranges, the two Tables I and II summarize the results obtained. We group Properties 2 and 4 in Table I, since they exhibit the same monotonicity behavior with respect to the parameters. For the same reason, Properties 1 and 3 are grouped in Table II. Both tables contain respective run times (for phases A, B and C), number of predicates (denoted by $\#P$), obtained probability values. We reuse abstract models to check multiple properties, therefore a row contains a single A and B entry and for each property a C entry.

TABLE I
BRP RESULTS FOR PROPERTIES 2 & 4.

$\mathtt{P}[N > 0, MAX \geq k]$							
				Property 2		Property 4	
k	$\#P$	A (s)	B (s)	prob	C (s)	prob	C (s)
2	53	0.262	191	2.65E-5	1.2	8.00E-6	0.56
3	54	0.266	204	7.89E-7	1.2	1.60E-7	0.58
4	55	0.269	207	2.35E-8	1.5	3.20E-9	1.24
5	56	0.273	209	7.00E-10	1.98	6.40E-11	0.65

As apparent in Table I, the probabilities of Properties 2 and 4 are invariant under N but decrease with MAX. Using PASS, we have computed probabilities for any instance of BRP with an arbitrary file size and $MAX \geq k$. For each k, the obtained probability is a tight upper bound agreeing with the probability obtained with PRISM for file sizes 16, 32, 64 and $MAX = k$. For these properties, phase B turns out to be the run time bottleneck.

TABLE II
BRP RESULTS FOR PROPERTIES 1 & 3 .

				Property 1		Property 3	
k	$\#P$	A (s)	B (s)	prob	C (s)	prob	C (s)
$\mathtt{P}[N = 16, MAX \geq k]$							
2	60	0.3	68	4.23E-4	10	1.85E-4	9
3	61	0.3	76	1.26E-5	10	5.52E-6	11
4	62	0.3	77	3.76E-7	11	1.65E-7	11
5	63	0.3	84	1.12E-8	16	4.90E-9	14
$\mathtt{P}[N = 32, MAX \geq k]$							
2	76	0.4	74	8.46E-4	19	6.08E-4	42
3	77	0.4	85	2.52E-5	23	1.81E-5	45
4	78	0.5	89	7.52E-7	32	5.41E-7	50
5	79	0.5	86	2.24E-8	36	1.61E-8	55
$\mathtt{P}[N = 64, MAX \geq k]$							
2	108	1.0	84	1.69E-3	68	1.45E-3	197
3	109	1.0	92	5.05E-5	55	4.34E-5	119
4	110	1.0	106	1.50E-6	69	1.29E-6	149
5	111	1.1	98	4.48E-8	87	3.85E-8	172

The probabilities of Properties 1 and 3 increase with N and decrease with MAX. If N is represented symbolically, these two properties have probability 1 in the abstract program, i.e., the upper bound for unknown file sizes is 1. If instead we fix size N to 16 (and to 32, 64), and consider the program $\mathtt{P}[N = 16, MAX \geq k]$ with $k = 2, 3, \ldots$, we get the exact probabilities corresponding to the result of PRISM for $MAX = k$ (see Table II). Note that in some cases, the bottleneck is in phase C, not B.

Altogether, we have successfully verified all the properties from Table 5 for all possible file sizes (Properties A, B, 2 and 4) and/or retransmission number (all of them). Further, we obtained *tight upper bounds* on the probabilities in all cases.

B. Effect of Optimizations.

To study the effect of the optimizations in the model extraction phase (A.3) discussed in Section V-B, we experimented with several different probabilistic programs. Some programs are novel (such as the `brp` models just discussed) and smaller test cases `roulette`, and `readers_writers`. Some are PRISM case studies, such as `ipv4`. Due to space constraints, we only give a brief insight into the time savings due to decomposition, while we do not report the effect of the other more obvious optimization. The resulting models are the same for the original and the optimized method. As shown in Table III significant savings in running time can be noted on larger models.

TABLE III
EFFECT OF DECOMPOSITION ON RUNNING TIME

program	# preds	A w/o opt (s)	A opt (s)
`roulette`	3	0.032	**0.028**
`readers_writers`	4	0.032	**0.028**
`brp_N_2`	53	0.356	**0.262**
`brp_N_5`	56	0.476	**0.273**
`brp_64_2`	108	2.088	**1.005**
`brp_64_5`	111	2.964	**1.107**
`ipv4`	61	45.162	**1.196**

VII. Related Work.

Model checking of probabilistic automata or MDPs, has been studied during the last decade [31], [1], [2], [32]. Huth gives a comprehensive survey [33] of abstraction techniques for both finite and infinite-state MDPs.

Several abstraction techniques addressing finite-state models have been introduced. D'Argenio *et al.* [7], [8] presented abstraction and refinement techniques for a particular class of reachability properties. Given a partition of the state space, they construct an abstraction based on the notion of quotient automata. Such abstractions tend to be much smaller than the original model. If the abstraction is too coarse, it is refined until either the original model is found to violate the property or the property can be established. Refinement is based on simulation criteria. Their method produces a lower bound for the minimum reachability probability and an upper bound for the maximum reachability probability.

Chatterjee [9] describes a fully automatic abstraction framework for probabilistic two-player games based on counterexample-guided abstraction refinement [34]. MDPs can be handled as a special case by their technique. It is demonstrated how the abstraction and refinement process can be implemented symbolically by means of MTBDD operations.

Recently, Kwiatkowska [10] introduced game-based abstraction techniques for MDPs. Their abstraction is also based on a partition of the state space. Applying results from stochastic games, they arrive at both upper and lower bound for the maximum and minimum reachability probability. Technically, they separate the non-determinism in the original MDP from the non-determinism introduced through the abstraction, and then represent each type as a player in the game. We believe that our method can be combined with this approach in a fruitful way.

The aforementioned abstraction techniques avoid a reachability analysis on the original model, however they still unfold its state space to construct the abstraction. This can be very costly for large state spaces and cannot be directly applied to infinite state spaces. In contrast, our abstraction method is based on reasoning at the language level. Hence, as far as the methods are comparable, our method scales better and admits infinite-state models. So far, however, we do not support automatic model refinement.

Rather loosely related are the following contributions: Monniaux [35] describes an extension of the framework of abstract interpretation [21] to infinite-state MDPs. The framework considers linear-time properties. The abstraction techniques just mentioned and the one presented here address branching-time properties. Remke *et al.* [36] presented model checking algorithms for a special class of infinite-state *continuous-time* Markov chains exploiting so-called quasi birth-death structure in the state space. Their method admits both transient and steady-state analysis. Abdulla *et al.* [37] studied eager Markov chains, which is a special class of infinite-state discrete-time Markov chains with reward structure. They presented algorithms to approximate the expected reward with arbitrary accuracy. Model checking techniques for probabilistic push-down automata [38] and recursive Markov chains [39] are orthogonal to our work: an interprocedural extension of our technique could abstract programs to probabilistic push-down automata or recursive Markov chains.

State-level aggregation techniques give complimentary means to alleviate the state explosion problem, but generally do not address infinite-state models. Techniques for probabilistic models include bisimulation reduction [40], [41], [42], partial order reduction [43], [44], [45], symmetry reduction [46], and many others.

VIII. Conclusion and Future Work

We have presented a novel predicate abstraction technique that enables us to check infinite-state probabilistic models against safe PCTL formulas. Our method is based on reasoning at the language level and does not unfold the state space.

The BRP example has shown that the core contribution of this paper, the generation of an abstract program is remarkably efficient. However, the way Boolean programs are currently submitted to PRISM is inefficient. We anticipate that a tighter integration of phase A and B (for which currently the model is dumped to disk) will speed up the postprocessing (B and C) of our models significantly. We therefore strive for a direct integration of PRISM and PASS in the near future, such that the abstract model structure can be better exploited in subsequent steps.

Initially, the relevant predicates are extracted from the guards and properties to be checked. However, if the abstract model needs to be refined, predicates are inserted manually. The obvious next step on our research agenda is to investigate automatic abstraction refinement to discover predicates automatically by extending existing refinement techniques such as [47], [48] to infinite-state probabilistic models. In the non-probabilistic setting, a counterexample is usually a path leading to a bad state. If the counterexample found in the abstraction happens to be spurious, the models must be refined to eliminate the counterexample. Recently, a theory of counterexamples of Markov chains has been developed [49], where a counterexample is a set of path whose measure refutes the safe bound specified by the PCTL formula.

Other interesting directions for further work are extensions of our technique to live PCTL following ideas from [10]. On the model side, we plan to admit Markovian continuous time (yielding continuous-time MDPs), and clocked continuous time (yielding probabilistic timed automata [50]).

An extended version of this paper including additional proofs and an executable of PASS are available at the URL:

`http://depend.cs.uni-sb.de/pass.html`

References

[1] H. Hansson and B. Jonsson, "A Logic for Reasoning about Time and Reliability," *Formal Asp. Comput.*, vol. 6, no. 5, pp. 512–535, 1994.

[2] A. Bianco and L. de Alfaro, "Model checking of probabilistic and nondeterministic systems," in *FSTTCS*. Springer, 1995, pp. 499–513.

[3] C. Baier, "On Algorithmic Verification Methods for Probabilistic Systems," 1998, Habilitationsschrift, Universität Mannheim.

[4] S. Graf and H. Saídi, "Construction of Abstract State Graphs with PVS," in *CAV*. London, UK: Springer-Verlag, 1997, pp. 72–83.

[5] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A Tool for Automatic Verification of Probabilistic Systems," in *TACAS*, 2006, pp. 441–444.

[6] C. Baier, J.-P. Katoen, H. Hermanns, and V. Wolf, "Comparative branching-time semantics for markov chains," *Inf. Comput.*, vol. 200, no. 2, pp. 149–214, 2005.

[7] P. R. D'Argenio, B. Jeannet, H. E. Jensen, and K. G. Larsen, "Reachability Analysis of Probabilistic Systems by Successive Refinements," in *PAPM-PROBMIV*, 2001, pp. 39–56.

[8] ——, "Reduction and Refinement Strategies for Probabilistic Analysis," in *PAPM-PROBMIV*, 2002, pp. 57–76.

[9] R. J. Krishnendu Chatterjee, Thomas A. Henzinger and R. Majumdar, "Counterexample-Guided Planning," in *UAI*, July 2005.

[10] M. Kwiatkowska, G. Norman, and D. Parker, "Game-based Abstraction for Markov Decision Processes," in *QEST*, 2006, pp. 157–166.

[11] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and Cartesian Abstraction for Model Checking C Programs." in *TACAS*, 2001, pp. 268–283.

[12] B. Dutertre and L. M. de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," in *CAV*, 2006, pp. 81–94.

[13] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast Decision Procedures," in *CAV*, 2004, pp. 175–188.

[14] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers." in *CADE*, 2002, pp. 295–313.

[15] G. Nelson and D. C. Oppen, "Simplification by Cooperating Decision Procedures." *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, 1979.

[16] D. Parker, "PRISM web site," www.cs.bham.ac.uk/˜dxp/prism.

[17] A. Parma and R. Segala, "Logical Characterizations of Bisimulations for Discrete Probabilistic Systems," in *FoSSaCS*, 2007, pp. 287–301.

[18] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[19] R. Segala and N. A. Lynch, "Probabilistic simulations for probabilistic processes," *Nord. J. Comput.*, vol. 2, no. 2, pp. 250–273, 1995.

[20] E. Clarke, O. Grumberg, and D. Long, "Model Checking and Abstraction," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1512–1542, 1994.

[21] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *POPL*, 1977, pp. 238–252.

[22] S. Bensalem, S. Graf, and Y. Lakhnech, "Abstraction as the Key for Invariant Verification," in *Verification: Theory and Practice*, 2003, pp. 67–99.

[23] E. W. Dijkstra, *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[24] F. Somenzi, "CUDD: CU Decision Diagram Package," http://vlsi.colorado.edu/˜fabio/CUDD/, 2005.

[25] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras, "SMT Techniques for Fast Predicate Abstraction," in *CAV*, 2006, pp. 424–437.

[26] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke, "Word level predicate abstraction and refinement for verifying RTL verilog." in *DAC*, 2005, pp. 445–450.

[27] S. Tonetta and N. Sharygina, "A uniform framework for predicate abstraction approximation," in *orkshop on Software Verification and Validation (SVV 2006)*, 2006.

[28] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, 2002.

[29] J. F. Groote and J. van de Pol, "A Bounded Retransmission Protocol for Large Data Packets." in *AMAST*, 1996, pp. 536–550.

[30] P. R. D'Argenio, J.-P. Katoen, T. C. Ruys, and J. Tretmans, "The Bounded Retransmission Protocol Must Be on Time!" in *TACAS*, 1997, pp. 416–431.

[31] M. Y. Vardi, "Automatic verification of probabilistic concurrent finite-state programs," in *FOCS*, 1985, pp. 327–338.

[32] C. Courcoubetis and M. Yannakakis, "The complexity of probabilistic verification." *J. ACM*, vol. 42, no. 4, pp. 857–907, 1995.

[33] M. Huth, "An abstraction framework for mixed non-deterministic and probabilistic systems." in *Validation of Stochastic Systems*, 2004, pp. 419–444.

[34] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement." in *CAV*, 2000, pp. 154–169.

[35] D. Monniaux, "Analyse de programmes probabilistes par interprétation abstraite," Thèse de doctorat, Université Paris IX Dauphine, 2001.

[36] A. Remke, B. R. Haverkort, and L. Cloth, "Model checking infinite-state markov chains." in *TACAS*, 2005, pp. 237–252.

[37] P. A. Abdulla, N. B. Henda, R. Mayr, and S. Sandberg, "Eager Markov Chains," in *ATVA*, 2006, pp. 24–38.

[38] J. Esparza, A. Kučera, and R. Mayr, "Model Checking Probabilistic Pushdown Automata," *Logical Methods in Computer Science*, 2006.

[39] K. Etessami and M. Yannakakis, "Algorithmic Verification of Recursive Probabilistic State Machines." in *TACAS*, 2005, pp. 253–270.

[40] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Inf. Comput.*, vol. 94, no. 1, pp. 1–28, 1991.

[41] J.-P. Katoen, T. Kemna, I. Zapreev, and D. N. Jansen, "Bisimulation minimisation mostly speeds up probabilistic model checking," in *TACAS*, ser. LNCS, no. 4424, 2007, pp. 76–92.

[42] S. Derisavi, "A symbolic algorithm for optimal Markov chain lumping," in *TACAS*, ser. LNCS, no. 4424, 2007, pp. 139–154.

[43] C. Baier, M. Größer, and F. Ciesinski, "Partial order reduction for probabilistic systems." in *QEST*, 2004, pp. 230–239.

[44] P. R. D'Argenio and P. Niebert, "Partial order reduction on concurrent probabilistic programs." in *QEST*, 2004, pp. 240–249.

[45] C. Baier, P. R. D'Argenio, and M. Größer, "Partial order reduction for probabilistic branching time." *Electr. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 97–116, 2006.

[46] M. Z. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking." in *CAV*, 2006, pp. 234–248.

[47] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction." in *POPL*, 2002, pp. 58–70.

[48] K. L. McMillan, "Lazy abstraction with interpolants." in *CAV*, 2006, pp. 123–136.

[49] T. Han and J.-P. Katoen, "Counterexamples in probabilistic model checking," in *TACAS*, ser. LNCS, no. 4424, 2007, pp. 60–75.

[50] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, "Automatic verification of real-time systems with discrete probability distributions." in *Theor. Comput. Sci.*, vol. 282, no. 1, 2002, pp. 101–150.