SMT-Based Bisimulation Minimisation of Markov Models

Christian Dehnert[1], Joost-Pieter Katoen[1], David Parker[2]

[1] RWTH Aachen University, Germany
[2] University of Birmingham, United Kingdom

Abstract. Probabilistic model checking is an increasingly widely used formal verification technique. However, its dependence on computationally expensive numerical operations makes it particularly susceptible to the state-space explosion problem. Among other abstraction techniques, bisimulation minimisation has proven to shorten computation times significantly, but, usually, the full state space needs to be built prior to minimisation. We present a novel approach that leverages satisfiability solvers to extract the minimised system from a high-level description directly. A prototypical implementation in the framework of the probabilistic model checker Prism provides encouraging experimental results.

1 Introduction

Markov chains are omnipresent. They are used in reliability analysis, randomised algorithms and performance evaluation. In the last decade, probabilistic model checking has emerged as a viable and efficient alternative to classical analysis techniques for Markov chains, which typically focus on transient and long-run probabilities. This growing popularity is mainly due to the availability of ever improving software tools such as Prism [15] and Mrmc [11]. Like traditional model checkers, these tools suffer from the curse of dimensionality—the state space grows exponentially in the number of system components and variables. As numerical computations are at the heart of verifying Markov chains, several numerical values need to be stored for each (reachable) state in addition to the model itself, making the state space explosion problem even more pressing.

A variety of techniques has been recently developed to reduce the state space of probabilistic models prior to verification. These include symmetry reduction [14], bisimulation minimisation [10] and abstraction, e.g., in the abstraction-refinement paradigm [13,9]. Bisimulation [16] (also known as ordinary lumping [3]) is of particular interest as it preserves widely used logics such as PCTL*, PCTL and probabilistic LTL [1], and its coarsest quotient can be efficiently computed [5,18]. In contrast to traditional model checking [6], it has proven to significantly shorten computation times [10]. The main drawback of bisimulation quotienting algorithms, however, is that they typically require the entire reachable state space. Techniques to alleviate this effect include the use of data structures based on binary decision diagrams (BDDs) to reduce storage costs [21,20,17], and compositional minimisation techniques [4]. This paper takes a radically different approach: we extract the bisimulation quotient directly from a high-level description using Smt (satisfiability modulo theories) solvers.

The starting-point of our approach is to take a probabilistic program representing a Markov chain, described in the Prism modelling language. This probabilistic program consists of guarded commands, each of which includes a probabilistic choice over a set of assignments that manipulate program variables. The main idea is to apply a partition-refinement bisimulation quotienting in a truly symbolic way. Each block in the partition is represented by a Boolean expression and weakest preconditions of guarded commands yield the refinement of blocks. All these computation steps can be dispatched as standard queries to an Smt solver. The quotient distribution, over blocks, for a state is obtained by summing up the probabilities (given by a command) attached to assignments that lead into the respective blocks. This is determined using an AllSat enumeration query by the Smt solver. Whereas similar techniques have been used for obtaining over-approximations of Markov models [19,12] —yielding sound abstractions for a "safe" fragment of PCTL properties— this is (to the best of our knowledge) the first bisimulation quotienting approach based on satisfiability solvers. This paper focuses on bisimulation for discrete-time Markov chains (DTMCs), but the techniques are also directly applicable to continuous-time Markov chains (CTMCs). In addition, in Section 4, we discuss how to extend them to models that incorporate nondeterminism, such as MDPs or CTMDPs.

Experiments on probabilistic verification benchmarks show encouraging results: speed-ups in total verification time by up to two orders of magnitude over Prism on one example, and, on another, scalability to models that go far beyond the capacity of current verification engines such as Prism and Mrmc. A comparison with the symbolic bisimulation engine Sigref [21] also yields favourable results. Although —like for BDD-based quotienting— there are examples with smaller improvements or where this approach is inferior, we believe that Smt-based quotienting offers clear potential. Other advantages of our approach are that it is applicable to infinite probabilistic programs whose bisimulation quotient is finite and that it is directly applicable to parametric Markov chains [8].

Related Work. For non-probabilistic models, Graf and Saïdi [7] pioneered *predicate abstraction*, which uses Boolean predicates and weakest precondition operations to compute abstractions. This technique is rather en vogue in software model checking, as is the use of SAT/SMT solvers to build abstractions [2]. Predicate abstraction has been adapted to the probabilistic setting [19,12] and used to obtain abstractions of probabilistic models, e.g., through CEGAR [9] or game-based abstraction refinement [13]. However, these methods usually only focus on a specific subset of PCTL* and do not compute precise abstractions but over-approximations. While the latter can be beneficial in some cases, in general it requires a separate abstraction for each property to be verified. In contrast, bisimulation minimisation is a precise abstraction, which coincides with PCTL* equivalence [1], so all properties that can be formulated in that logic are preserved. Another important difference is that probabilistic abstraction-refinement techniques [9,13] usually require an expensive numerical solution phase to perform refinement, whereas partition refinement for bisimulation is comparatively cheap. Omitting numerical analysis also makes our approach easily extendable to systems with continuous time and parameters. Wimmer et al. [21] use BDDs to symbolically compute the bisimulation quotient. However, this requires the construction of a BDD representation of the state space of the full, unreduced model prior to minimisation, which we deliberately try to avoid.

2 Preliminaries

We begin with some brief background material on discrete-time Markov chains, bisimulation minimisation and the Prism modelling language.

Markov models. Markov models are similar to transition systems in that they comprise states and transitions between these states. In discrete-time Markov chains, each state is equipped with a discrete probability distribution over successor states. Let $Dist_S$ be the set of discrete probability distributions over S.

Definition 1 (Discrete-time Markov chain (DTMC)). *A DTMC is a tuple $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$ where (i) S is a countable, non-empty set of states, (ii) $\mathbf{P} : S \times S \to [0,1]$ is the transition probability function that assigns to each pair (s, s') of states the probability $\mathbf{P}(s, s')$ of moving from state s to s' in one step such that $\mathbf{P}(s, \cdot) \in Dist_S$, (iii) $s_{init} \in S$ is the initial state, (iv) AP is a set of atomic propositions, and (v) $L : S \to 2^{AP}$ is the labelling function that assigns a (possibly empty) set of atomic propositions $L(s)$ to a state $s \in S$.*

For quantitative reasoning over DTMCs, we focus on the logic PCTL*, a probabilistic extension of CTL* which subsumes other common logics such as PCTL and probabilistic LTL. Instead of the path quantifiers (A and E) of CTL*, it features the $\mathcal{P}_{\bowtie p}(\varphi)$ operator, which asserts that the probability mass of paths satisfying φ satisfies $\bowtie p$, where $\bowtie \in \{<, \leq, >, \geq\}$. For example, the property “the probability that the system eventually fails lies below 70%” can be expressed as $\mathcal{P}_{<0.7}(\Diamond fail)$. Let $s \models_{\mathcal{D}} \Phi$ denote that state s of a DTMC $\mathcal{D}$ satisfies a formula Φ. If $s_{init} \models_{\mathcal{D}} \Phi$, the DTMC $\mathcal{D}$ satisfies Φ and we denote this by $\mathcal{D} \models \Phi$.

Bisimulation minimisation. Given a DTMC $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$ and an equivalence relation R on S, the set of equivalence classes of S under R is denoted S/R and $[s]_R$ is the equivalence class of $s \in S$ under R.

Definition 2 (Quotient distribution). *For equivalence relation R and distribution $\mu \in Dist_S$, the* quotient distribution *of μ with respect to R, denoted $[\mu]_R \in Dist_{S/R}$, is defined by $[\mu]_R(C) = \sum_{s \in C} \mu(s)$ for all $C \in S/R$.*

Definition 3 ((Strong) probabilistic bisimulation). *An equivalence relation R on S is a* (strong) bisimulation *on $\mathcal{D}$ if, for all $(s_1, s_2) \in R$:*

$$L(s_1) = L(s_2) \text{ and } \mathbf{P}(s_1, C) = \mathbf{P}(s_2, C) \text{ for all } C \in S/R$$

where $\mathbf{P}(s, C)$ denotes the sum $\sum_{s' \in C} \mathbf{P}(s, s')$.

States $s_1, s_2 \in S$ are (strongly) bisimilar, denoted $s_1 \sim s_2$, if there exists a bisimulation on $\mathcal{D}$ that relates s_1 and s_2. Note that $\sim$ is the coarsest bisimulation on $\mathcal{D}$. Intuitively, bisimilar states can stepwise simulate each other, meaning that they can be merged while preserving important properties.

Definition 4 (Bisimulation quotient). *For a DTMC $\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$, the* bisimulation quotient *is defined as $\mathcal{D}/{\sim} = (S/{\sim}, \mathbf{P}', [s_{init}]_\sim, AP, L')$, where*

$$\mathbf{P}'([s]_\sim, [t]_\sim) = \mathbf{P}(s, [t]_\sim) \text{ and } L'([s]_\sim) = L(s).$$

Note that $\mathcal{D}/{\sim}$ is well-defined. The preservation theorem of Aziz et al. [1] states that strong bisimulation is sound and complete with respect to PCTL*:

Proposition 1. $s \models_{\mathcal{D}} \Phi \iff [s]_\sim \models_{\mathcal{D}/\sim} \Phi$ *for all PCTL* formulae* Φ*.*

This implies $\mathcal{D} \models \Phi$ if and only if $\mathcal{D}/\sim \models \Phi$ for all PCTL* formulae Φ. Put differently, a given formula may be verified on the (possibly much smaller) bisimulation quotient while preserving the verification result of the original model.

The PRISM modelling language. PRISM is a widely used, state-of-the-art probabilistic model checker. It features a high-level modelling language, adopted by several other tools. We focus on a subset of this language whose semantics corresponds to DTMCs. Let Var be a finite set of variables, each of which is typed as either Boolean or bounded integer, and $\Sigma(\mathit{Var})$ be the set of all valuations of variables in Var that respect these types. Furthermore, let $\mathit{Expr}_{\mathit{Var}}$ and $\mathit{BExpr}_{\mathit{Var}}$ denote the set of all expressions over Var and the Boolean expressions thereof, respectively. For $e \in \mathit{Expr}_{\mathit{Var}}$ and $s \in \Sigma(\mathit{Var})$, let $[\![e]\!]_s$ be the value of e in s, i.e. the value of e when all occurring variables are replaced by their values in s. Let $s \models b$ for $b \in \mathit{BExpr}_{\mathit{Var}}$ iff $[\![b]\!]_s = 1$ and let $[\![b]\!]$ be the set of all valuations that assign the truth value 1 to b, i.e., $[\![b]\!] = \{s \in \Sigma(\mathit{Var}) \mid s \models b\}$.

Definition 5 (Assignment). *An* assignment *over a set of variables Var is a function $E : \mathit{Var} \to \mathit{Expr}_{\mathit{Var}}$ that complies with the respective types.*

For a valuation $s \in \Sigma(\mathit{Var})$ and an assignment E over Var, we write $s \xrightarrow{E} s'$ if and only if for all $v \in \mathit{Var}$ we have $[\![v]\!]_{s'} = [\![E(v)]\!]_s$ with the intuition that s is transformed into s' by updating the values of all variables according to E. If $s \xrightarrow{E} s'$, we will say that the execution of E in s leads to state s'.

Definition 6 (Guarded command). *A* guarded command $c = (a, g, (p_1, E_1), \ldots, (p_n, E_n))$ *over Var and a set Act of actions consists of (i) an action $a \in \mathit{Act}$, (ii) a guard $g = \mathit{guard}(c) \in \mathit{BExpr}_{\mathit{Var}}$ and (iii) probabilities $p_i \in [0, 1]$ associated with assignments E_i over Var for $1 \leq i \leq n$ such that $\sum_{1 \leq i \leq n} p_i = 1$.*

Syntactically, we write c as $[a]\ g \longrightarrow p_1 : (\mathit{Var}'{=}E_1) + \ldots + p_n : (\mathit{Var}'{=}E_n)$, where $\mathit{Var}'{=}E_i$ is short for a list of entities of the form $v'{=}e$ for $v \in \mathit{Var}$ and $e = E_i(v)$. Intuitively, a guarded command can be executed in every state that satisfies its guard. If it is executed, the ith assignment is executed with probability p_i. A probabilistic program comprises a set of guarded commands.

```
pc: int[1,4] init 1;    h, f, r: bool init false;
[coin] pc=1  → 0.5 : (pc′=pc + 1)&(h′=¬h) + 0.5 : (pc′=pc + 1)&(h′=h);
[proc] pc=2  → 0.2 : (pc′=pc + 1)&(f′=¬f) + 0.8 : (pc′=pc + 1)&(f′=f);
[rtn1] pc=3 ∧ h ∧ ¬f  → 0.2 : (pc′=pc + 1)&(r′=0)&(f′=1) + 0.8 : (pc′=pc + 1)&(r′=1);
[rtn2] pc=3 ∧ ¬h ∧ ¬f  → 0.5 : (pc′=pc + 1)&(r′=0)&(f′=1) + 0.5 : (pc′=pc + 1)&(r′=1);
[rest]  pc=3 ∧ f  → 0.99 : (pc′=1)&(h′=0)&(f′=0)&(r′=0) + 0.01 : (pc′=pc + 1);
[done] pc=4  → 1 : (pc′=pc);
```

Fig. 1. Running example: The probabilistic program P_{Ex}.

Definition 7 (Probabilistic program). *A* probabilistic program $P = (Var, s_{init}, Act, Comm)$ *consists of (i) a finite set Var of variables, (ii) an initial state $s_{init} \in \Sigma(Var)$, (iii) a finite set of actions Act and (iv) a finite set of guarded commands $Comm$ over Var and Act. (v) Additionally, for each $s \in \Sigma(Var)$, there must exist exactly one $c \in Comm$ with $s \models guard(c)$.*

Example 1. Fig. 1 shows our running example: a probabilistic program P_{Ex} over variables $Var_{Ex} = \{pc, h, f, r\}$, modelling a randomised algorithm with 4 phases (indicated by pc). The algorithm starts by throwing a coin (h) before a processing step that has certain probability (0.2) to fail (f). In case of a failure, the algorithm restarts with high probability (0.99) and terminates in error otherwise. If there is no failure, it returns a result r that is either true or false with a certain probability that depends on the coin flip. As false is the (supposedly) incorrect result, the fail flag is set in this case. Note that all variables v that are not assigned any value in an assignment keep their previous value, i.e. have $v' = v$. For the sake of clarity, we sometimes include these superfluous assignments. In further examples in this paper, we refer to the command with name a by c_a (for instance, the coin flip command will be referred to as c_{coin}) and we denote the ith assignment of command c_a by $E_{a,i}$.

The PRISM modelling language also supports parallel composition of modules, where some commands are executed synchronously. We deal with such models by flattening them into one module, using a symbolic composition of the commands that need to synchronise. While this may increase the number of commands in the program, this is always possible in a totally automatic way and is thus no restriction on the expressivity of the probabilistic programs considered.

The semantics of a probabilistic program P is a DTMC $[\![P]\!]$ whose state space is $\Sigma(Var)$ and whose transitions are defined by the guarded commands. The additional constraint (see Def. 7 (v)) assures that a guarded command induces a probability distribution over the successor states and that there are neither deadlock states nor states that have multiple guarded commands enabled. Note that this is no restriction. If there exists a state that satisfies no guard, the state has no outgoing transition and is thus equipped with a self-loop for model checking purposes. This can already be done at the language level by introducing a loop command for states that do not have any outgoing transition. On the other hand, if there is a state that satisfies multiple guards, this corresponds to a nondeterministic choice in that state. Hence, the semantics of the program would be

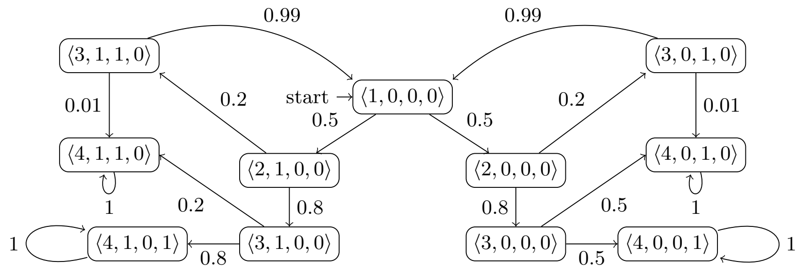

Fig. 2. The (reachable) fragment of $[\![P_{Ex}]\!]$, with states of the form $\langle pc, h, f, r\rangle$.

an MDP instead of a DTMC. The extension of our approach to non-deterministic models is possible (see Section 4), but not the main concern of this paper.

Example 2. The reachable part of the state space of the probabilistic program P_{Ex} (see Fig. 1) is depicted in Fig. 2, where the states are of the form $\langle pc, h, f, r\rangle$.

3 SMT-based Bisimulation Minimisation

We now give our Smt-based approach to bisimulation minimisation. We first summarise how to perform minimisation using partition refinement, and then describe its symbolic implementation using weakest preconditions and Smt.

Partition refinement. The standard approach to deriving a bisimulation quotient algorithmically prior to verification is to use partition refinement [5,18]. This technique starts with an initially coarse partition of the state space S and successively splits (refines) blocks containing states that have different behaviour with respect to the current partition until no more refinement is necessary. If the initial partition is chosen based on the atomic propositions, this results in the coarsest partition of S that induces a bisimulation, i.e., $S/\sim$.

In the probabilistic setting, a block B of partition $\mathit{\Pi}$ needs to be split if it contains two states that possess different quotient distributions with respect to the current partition, i.e. if there exist $s_1, s_2 \in B$ such that $[\mathbf{P}(s_1, \cdot)]_{R(\mathit{\Pi})} \neq [\mathbf{P}(s_2, \cdot)]_{R(\mathit{\Pi})}$ where $R(\mathit{\Pi})$ is the equivalence relation induced by the partition $\mathit{\Pi}$. In other words, in order to implement a partition refinement approach, we need to: (i) determine if B contains states with different quotient distributions and (ii) if so, identify the subsets of B which agree on the quotient distribution, because these form the blocks into which B is split.

In our work, a partition $\mathit{\Pi} = \{B_1, \ldots, B_k\}$ of the state space $S = \mathit{\Sigma}(Var)$ is represented *symbolically* by corresponding Boolean expressions $\pi = \{b_1, \ldots, b_k\}$ such that $B_i = [\![b_i]\!]$ for each $1 \leq i \leq k$. In the sections below, we describe how to reason symbolically, using weakest preconditions, about: (i) whether a state's successors are contained in a particular block; and (ii) the quotient distribution of a state. Once the partition refinement algorithm terminates (i.e., there are no further blocks in the current partition to split), the quotient DTMC is constructed as follows: its state space is taken as the set $\mathit{\Pi}$ of blocks in the final partition; and the transition probabilities for each block $B \in \Pi$ are then given by the (unique) corresponding quotient distribution for that block.

Weakest preconditions. To reason symbolically about the effect of an assignment E in a command of a probabilistic program, we use the weakest precondition operation. The weakest precondition of $b_i \in \mathit{BExpr}_{\mathit{Var}}$ with respect to E, denoted $\mathit{wp}(b_i, E)$, characterizes exactly the valuations s of $\Sigma(\mathit{Var})$ for which the successor valuation after assignment E satisfies b_i:

$$s \models \mathit{wp}(b_i, E) \quad \Longleftrightarrow \quad s \xrightarrow{E} s' \text{ with } s' \models b_i .$$

We can determine $\mathit{wp}(b_i, E)$ through a purely syntactic modification of b_i by simultaneously replacing each occurrence of each variable $v \in \mathit{Var}$ in b_i by $E(v)$.

Example 3. Consider the first assignment of the command c_{coin} in Example 1:

$$E_{coin,1}(pc) = pc + 1,\ E_{coin,1}(h) = \neg h,\ E_{coin,1}(f) = f,\ E_{coin,1}(r) = r$$

and let $b_1{=}\neg h$. Then $\mathit{wp}(b_1, E_{coin,1}) = \neg\neg h \equiv h$. This reflects the fact that exactly the states s with $s \models h$ are transformed into a state s' by $E_{coin,1}$ such that $s' \models \neg h$. Intuitively, this is because $E_{coin,1}$ flips the truth value of h. □

We fix, from now on, a command $c = [a]\ g \longrightarrow p_1 : (\mathit{Var}'{=}E_1) + \ldots + p_n : (\mathit{Var}'{=}E_n)$ with n assignments. Given n Boolean expressions $b_{i_1}, \ldots, b_{i_n}$ for indices $i_1, \ldots, i_n \in \{1, \ldots, k\}$, observe that, for $s \in \Sigma(Var)$:

$$s \models \bigwedge\nolimits_{j=1}^{n} \mathit{wp}(b_{i_j}, E_j) \quad \Longleftrightarrow \quad \text{for all } 1 \leq j \leq n\ .\ s \xrightarrow{E_j} s_j \text{ such that } s_j \models b_{i_j} .$$

Note, however, that the command c is not necessarily enabled in all the states s since some might fail to satisfy the guard g of c. If, in addition, such a state satisfies g, we know that in the semantics of the probabilistic program, state s has an outgoing probability distribution that goes with probability p_j to a state s_j satisfying b_{i_j}. We say that the jth assignment of c will lead into block B_{i_j}, which we write as $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. If, on the other hand, $\varphi = g \wedge \bigwedge_{j=1}^{n} \mathit{wp}(b_{i_j}, E_j)$ is unsatisfiable, i.e. there is no $s \in \Sigma(Var)$ such that $s \models \varphi$, then there exists no state s for which $s \xrightarrow{E_j} s_j$ such that $s_j \models b_{i_j}$ for all $1 \leq j \leq n$.

Example 4. Consider the command c_{coin} and its two assignments $E_{coin,1}$ and $E_{coin,2}$ (see Fig. 1) and let $b_1 = \neg h$ and $b_2 = h$. Then:

$$\bigwedge\nolimits_{i=1}^{2} \mathit{wp}(b_i, E_{coin,i}) = \neg\neg h \wedge h \equiv h\,.$$

So, for a state s where $s \models \mathit{guard}(c_{coin})$, i.e. $s \models (pc = 1)$, and $s \models h$, we conclude that command c is enabled, assignment $E_{coin,1}$ will lead into a state satisfying $\neg h$ (b_1) and assignment $E_{coin,2}$ will lead into one satisfying h (b_2). We denote this by $s \xrightarrow{c_{coin}} (B_1, B_2)$. □

Quotient distributions. Reasoning in a similar way, we can also determine the quotient distribution for a state s with respect to the current partition. From above, if $s \in \Sigma(Var)$ with $s \models guard(c)$ and $s \models \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$, we have that $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. Since command c has n assignments, each leading into a block, we have an n-tuple $(B_{i_1}, \ldots, B_{i_n})$ of target blocks per state. These blocks B_{i_j} are not necessarily distinct. Accordingly, to determine the quotient probability distribution of s with respect to the partition Π, we sum up the probabilities that lead into the same blocks.

Note, however, that the probabilities do not appear in the formulas. Depending on whether or not a state satisfies the conjunction of weakest preconditions for certain blocks, we know whether the corresponding assignments will take that state to the associated blocks. Based on this knowledge, the probability distribution is directly given by the probabilistic program.

Example 5. Consider again Example 4 and note that $b_1 = \neg h$ and $b_2 = h$ induce a partition of $\Sigma(Var)$. From the (fixed) probabilities associated with the two assignments, $E_{coin,1}$ and $E_{coin,2}$ in the program (Fig. 1), we can conclude that, for all states s with $s \models (pc = 1)$ and $s \models h$, there is a 0.5 probability to move to block B_1 in the next step and the same holds for B_2.

In contrast, consider the partition Π' induced by $b'_1 = (pc \neq 2)$ and $b'_2 = (pc = 2)$. Then all states satisfying $guard(c_{coin}) = (pc = 1)$ and

$$\bigwedge_{i=1}^{2} wp(b'_2, E_{coin,i}) = (pc + 1 = 2) \wedge (pc + 1 = 2) \equiv (pc = 1)$$

will move to B'_2 with both assignments. In other words, the quotient probability distribution for all states s with $s \models (pc = 1)$ is given by $\mathbf{P}(s, B'_2) = 1.0$ and $\mathbf{P}(s, B'_1) = 0.0$. This can also be seen by looking at the probabilistic program: starting in a state with $(pc = 1)$ will result in a state with $(pc = 2)$ with probability 1.0 after one step, because only c_{coin} is available in these states and all assignments of that command increase pc by one. □

Refinement of a block using SMT. Recall from the start of this section that the key operation required to perform bisimulation minimisation using partition refinement is to split a block B by identifying the different possible quotient distributions for states within B. We now explain, building on the above, how to perform this using queries executed by an Smt solver.

Suppose again, that the current partition is $\Pi = \{B_1, \ldots, B_k\}$, where each block B_i is represented by a Boolean expression b_i, i.e. $B_i = [\![b_i]\!]$, and that we are to refine block $B = [\![b]\!] \in \Pi$. We now formulate this computation step as a series of queries to an Smt solver. Given a command $c = [a]\ g \longrightarrow p_1 : (Var'{=}E_1) + \ldots + p_n : (Var'{=}E_n)$ as before, observe that there is a state $s \in B$ where c is enabled and the jth assignment of c leads into block B_{i_j} for all $1 \leq j \leq n$, i.e. $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$, if and only if the formula

$$b \wedge g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j) \qquad (1)$$

is satisfiable. While the first conjunct ensures that the state is in block B, the rest of the formula guarantees that the state in question exhibits the appropriate behaviour.

Example 6. Recall the partition given by $b_1{=}\neg h, b_2{=}h$ of Example 4. Since:

$$\underbrace{\neg h}_{b_1} \quad \wedge \quad \underbrace{(pc = 1)}_{guard(c_{coin})} \quad \wedge \quad \underbrace{\neg h}_{wp(b_2, E_{coin,1})} \quad \wedge \quad \underbrace{h}_{wp(b_2, E_{coin,2})}$$

is unsatisfiable, we can conclude that there are no states in B_1 that have c_{coin} enabled such that both assignments lead into block B_2. Intuitively, this is because the first assignment flips the value of h while the second one leaves the value untouched. Therefore, there exists no possible value of h such that after executing either one of the assignments h (b_2) always holds. In contrast, because:

$$\underbrace{\neg h}_{b_1} \quad \wedge \quad \underbrace{(pc = 1)}_{guard(c_{coin})} \quad \wedge \quad \underbrace{\neg h}_{wp(b_2, E_{coin,1})} \quad \wedge \quad \underbrace{\neg h}_{wp(b_1, E_{coin,2})} \quad \equiv \neg h \wedge (pc = 1)$$

is satisfiable, we know that there is a state $s \in B_1$ with $s \xrightarrow{c_{coin}} (B_2, B_1)$. Futhermore, these states are exactly characterized by $\neg h \wedge (pc = 1)$. □

Now, we can determine all possible target block tuples (and hence quotient distributions) for a block B under c by checking the corresponding formulas for satisfiability. The problem with this naive approach is the sheer number of satisfiability queries. Consider a guarded command with n assignments and a partition of k blocks. Then there are n^k different block tuples the command might lead into, and we would therefore need as many Smt queries in order to determine all target block tuples, despite the fact that almost all of them will be unsatisfiable. The latter observation justifies the key idea of our approach: we determine the different quotient probability distributions available in B via c using an AllSat enumeration query answered by an Smt solver. That is, we present a formula system to the Smt solver whose solutions correspond to available target block tuples and let the Smt solver enumerate all possible solutions. This means that we only need to create the formula system once and the solver will only enumerate as many solutions as there are. This observation justifies the key idea of our approach: we determine the different quotient probability distributions available in B via c using an AllSat enumeration query answered by an Smt solver. That is, we present a formula system to the Smt solver whose solutions correspond to available target block tuples and let the Smt solver enumerate all possible solutions. This means that we only need to create the formula system once and the solver will only enumerate as many solutions as there are.

To that end, we construct a formula system $S_{\Pi,c}$ that uses unique auxiliary variables $z_{i,j}$ for each weakest precondition $wp(b_j, E_i)$ with the intention that the values of these variables in a satisfying assignment encode the behaviour of some states in B. More concretely, we assert that $z_{i,j}$ implies $wp(b_i, E_j)$ and, additionally, for each assignment E_j we require at least one of the corresponding

$z_{i,j}$ to be true. This results in the following formula system $S_{\Pi,c}$:

$$b \tag{2}$$

$$g \tag{3}$$

$$z_{i,j} \rightarrow wp(b_i, E_j) \text{ for all } 1 \leq i \leq k \text{ and all } 1 \leq j \leq n \tag{4}$$

$$\bigvee_{i=1}^{k} z_{i,j} \text{ for all } 1 \leq j \leq n \tag{5}$$

Observe that this yields a correspondence between satisfying assignments for the variables $z_{i,j}$ and target block tuples under command c available in B as follows.

Example 7. Assume the solver returns a solution (i.e. a valuation $s \in \Sigma(Var)$ such that all formulas evaluate to true) to the formula system $S_{\Pi,c}$ with $z_{1,j} = 1$ for all $1 \leq j \leq n$. Obviously $s \in B$, because $s \models b$. Then, because of (3), the command c is enabled in s. Also, due to (4), we have $s \models \bigwedge_{j=1}^{n} wp(b_1, E_j)$. Stated differently, there exists an $s \in B$ such that $s \xrightarrow{c} (B_1, \ldots, B_1)$.

In general, given a satisfying assignment α, (5) guarantees that there exist indices $1 \leq i_1, \ldots, i_n \leq k$ such that $\alpha(z_{i_1,1}) = \ldots = \alpha(z_{i_n,n}) = 1$. Then, because of (4), it follows that $s \models \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$. Together with (2) and (3) this implies $s \models b \wedge g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)$ (cf. formula (1)). In other words, we can conclude that there exists $s \in B$ with $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$. Note that we are only interested in the values of the $z_{i,j}$ in a satisfying assignment returned by the solver, as they alone enumerate the target block tuples. Also, in (4), implications rather than equivalencies suffice, because $wp(b_{i_k}, E_j)$ and $wp(b_{i_l}, E_j)$ are mutually unsatisfiable for $i_k \neq i_l$. Intuitively, this is because the jth assignment cannot lead to two different blocks B_{i_k} and B_{i_l} from one state. Hence, because of 5), in each satisfying assignment exactly one of the $z_{i,j}$ is assigned true for all $1 \leq j \leq n$ and each solution corresponds to one target block tuple. The solution found can easily be ruled out by additionally asserting $\bigwedge_{j=1}^{n} \neg z_{i_j,j}$ and the solver can then be used to retrieve the next solution if there is any.

The SmtRefine Algorithm. Algorithm 1 presents an abstract implementation of SmtRefine, our Smt-based block refinement procedure. It takes as input a partition Π of $\Sigma(Var)$ given by Boolean expressions and a block $B \in \Pi$ given by Boolean expression b. It returns a partition Π_B of B given by Boolean expressions, such that all states in each block of Π_B share the same quotient distribution wrt. Π. In other words, Π_B is a stable partition of B wrt. Π.

The algorithm computes a (partial) mapping $sig\colon Dist_\Pi \rightarrow 2^{BExpr_{Var}}$ from $Dist_\Pi$, the probability distributions over Π, to a set of (mutually unsatisfiable) Boolean expressions. This is done such that a state $s \in B$ has the quotient probability distribution μ iff it satisfies one of the expressions in $sig(\mu)$, i.e.:

$$[\mathbf{P}(s, \cdot)]_{R(\Pi)} = \mu \quad \Longleftrightarrow \quad \text{there exists } b_\mu \in sig(\mu) \text{ such that } s \models b_\mu \tag{6}$$

where $R(\Pi)$ is the equivalence relation induced by the partition Π. Put differently, sig maps all available quotient probability distributions in B to Boolean expressions that characterize exactly the states having these distributions. Then, upon termination of SmtRefine, Π_B is given by the expressions:

$$\left\{ \bigvee_{b_\mu \in sig(\mu)} b_\mu \;\mid\; \mu \in keys(sig) \right\} \tag{7}$$

where $keys(sig)$ denotes the domain of sig, i.e. the quotient distributions in B.

The algorithm works as follows. We start by initialising the mapping sig to the empty mapping. Then, for each command c in the probabilistic program, we build the formula system $S_{\Pi,c}$ and pass it to the solver ($assert(S_{\Pi,c})$). As long as the solver finds any solutions (hasNextSolution()), we retrieve that solution, say α, via $getSolution()$ from the Smt solver. Note that α is a mapping of all the variables in Var and the auxiliary variables $z_{i,j}$ to their respective domains such that all formulas of $S_{\Pi,c}$ evaluate to true. As we are only interested in the values of the variables $z_{i,j}$, we can drop the other parts of the assignment.

As previously shown, such a solution implies that there exists a state $s \in B$ with $s \xrightarrow{c} (B_{i_1}, \ldots, B_{i_n})$ for certain indices i_j, $1 \leq j \leq n$. Given a solution α to the formula system, the function $getBlockCombination(\alpha)$ can compute these indices. As there may be different assignments of c leading s into the same block, we need to determine the quotient distribution μ by summing up the corresponding probabilities given by c. This is done by the function $compDist()$, which obviously needs to take the target block tuple as a parameter. Now that we know that there exists (at least) one state $s \in B$ whose quotient distribution is μ, we need to update the mapping sig accordingly. This is done by adding to $sig(\mu)$ the expression characterizing exactly those states that lead into the current target block tuple via c. Finally, we rule out the solution α previously found by the solver. More precisely, we rule out that particular combination of the $z_{i,j}$ being set to 1, because we do not want to enumerate this target block tuple again. If the solver now still finds possible target block tuples, we repeat the whole procedure. Note that the while-loop in Alg. 1 realises an AllSat procedure, as all solutions of the formula system are first found (via $getSolution()$) and ruled out later (via $ruleOutSolution(\alpha)$) as long as there exists another solution. After all target block tuples have been enumerated, we need to determine whether the block needs to be split. This is the case, if there is more than one quotient probability distribution available in B, i.e. if the domain of sig consists of more than one element. In that case, a stable partition of B is given by sig according to equation (7). If there is exactly one quotient distribution available, we don't need to split B and just return b itself.

The correctness of the Smt-based refinement algorithm is captured by the following theorem:

Theorem 1 (Correctness). *Given a partition $\Pi = \{B_1, \ldots B_k\}$ represented by a set of Boolean expressions $\{b_1, \ldots, b_k\}$ and a block $B = [\![b]\!] \in \Pi$,* SmtRefine *returns a partition Π_B of B given by mutually unsatisfiable Boolean expressions $\{b'_1, \ldots, b'_m\}$ such that for all $s_1, s_2 \in B$:*

$$\mathbf{P}(s_1, T) = \mathbf{P}(s_2, T) \textit{ for all } T \in S/\Pi \quad \textit{iff} \quad s_1 \models b'_i \Leftrightarrow s_2 \models b'_i \textit{ for all } 1 \leq i \leq m$$

Algorithm 1 SMT-based block refinement

```
Require: partition $\Pi$ given by $\pi = \{b_1, \ldots, b_k\}$, block $B = [\![b]\!] \in \Pi$
Ensure: returns stable (wrt. $\Pi$) partition $\Pi_B$ of $B$
  procedure SMTREFINE($b, \pi = \{b_1, \ldots, b_k\}$)    ▷ Refines $B$ wrt. $\Pi$
    $sig = \emptyset$    ▷ Initialize mapping of $Dist_\Pi$ to set of expressions in $BExpr_{Var}$
    for each $c = [a]\ g \longrightarrow p_1 : Var' = E_1 + \ldots + p_n : Var' = E_n \in Comm$ do
      $assert(S_{\Pi,c})$
      while hasNextSolution() do
        $\alpha \leftarrow getSolution()$    ▷ Retrieve solution from solver
        $(B_{i_1}, \ldots, B_{i_n}) \leftarrow getBlockCombination(\alpha)$    ▷ Compute a target block combination induced by $\alpha$
        $\mu \leftarrow compDist(B_{i_1}, \ldots, B_{i_n})$    ▷ Compute the corresponding distribution
        $sig(\mu) \leftarrow sig(\mu) \cup \{b \wedge g \wedge \bigwedge_{j=1}^{n} wp(b_{i_j}, E_j)\}$    ▷ Update signature mapping
        $ruleOutSolution(\alpha)$    ▷ Rule out current solution for solver
      end while
    end for

    if $|keys(sig)| > 1$ then
      return $\{ \bigvee_{b_\mu \in sig(\mu)} b_\mu \mid \mu \in keys(sig)\}$    ▷ Split block only if necessary
    else
      return $\{b\}$    ▷ Otherwise return input block
    end if
  end procedure
```

Example 8. Let P_{Ex} be the probabilistic program in Fig. 1. Furthermore, let the initial partition be $\Pi_{init} = \{[\![pc \neq 4]\!], [\![pc = 4]\!]\}$ given by the Boolean expressions $b_1 = (pc \neq 4)$ and $b_2 = (pc = 4)$. Now assume that the first block that is to be refined is $B_1 = [\![b_1]\!]$.

For the outermost loop in Alg. 1, we consider the commands in the order in which they appear in P_{Ex}. Accordingly, we start with c_{coin} and build the formula system $S_{\Pi,c_{coin}}$ as:

$$\begin{array}{ll}
\underbrace{pc \neq 4}_{b_1} & \underbrace{pc = 1}_{guard(c_{coin})} \\
z_{1,1} \rightarrow \underbrace{pc + 1 \neq 4}_{wp(b_1, E_{coin,1})} & z_{1,2} \rightarrow \underbrace{pc + 1 = 4}_{wp(b_2, E_{coin,1})} \\
z_{2,1} \rightarrow \underbrace{pc + 1 \neq 4}_{wp(b_1, E_{coin,2})} & z_{2,2} \rightarrow \underbrace{pc + 1 = 4}_{wp(b_2, E_{coin,2})} \\
z_{1,1} \vee z_{1,2} & \\
z_{2,1} \vee z_{2,2} &
\end{array}$$

Because the formula $pc = 1$ is part of the formula system, the value of pc is fixed to one, which in turn means that $z_{1,2}$ and $z_{2,2}$ can never be set to true in a solution of the system. This corresponds to the fact that there all states that have c_{coin} enabled (and therefore need to satisfy its guard, namely $pc = 1$) have no way of going to block B_2 with any of the assignments, because pc is only increased by 1.

In fact, the solver only returns a solution with $z_{1,1} = 1$ and $z_{2,1} = 1$, meaning that, for all states in B_1 that have c_{coin} enabled, both assignments lead into B_1. Accordingly, the quotient distribution μ_1 is given by:

$$\mu_1(B_1) = 1.0 \text{ and } \mu_1(B_2) = 0.0$$

We update the previously empty mapping sig to:

$$sig = \{\mu_1 \mapsto \{ \underbrace{pc \neq 4 \wedge pc = 1}_{s \in B_1 \wedge s \models guard(c_{coin})} \wedge \underbrace{pc + 1 \neq 4 \wedge pc + 1 \neq 4}_{s \models wp(b_1, E_{coin,1}) \wedge wp(b_1, E_{coin,2})} \}\}$$

and continue with constructing the formula system for command c_{proc}:

$$\begin{array}{ll} pc \neq 4 & \\ pc = 2 & \\ z_{1,1} \rightarrow pc + 1 \neq 4 & z_{1,2} \rightarrow pc + 1 = 4 \\ z_{2,1} \rightarrow pc + 1 \neq 4 & z_{2,2} \rightarrow pc + 1 = 4 \\ z_{1,1} \vee z_{1,2} & \\ z_{2,1} \vee z_{2,2} & \end{array}$$

Except for the guard it looks exactly as the previous formula system, because the assignments of this command do exactly the same transformation to pc as the assignments of c_{coin}. Exactly because of this, the solver will once again only return a solution with $z_{1,1} = 1$ and $z_{2,1} = 1$, which means that the corresponding states also possess the same quotient distribution μ_1 as before. This yields:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\} \quad \} \end{aligned}$$

The next command to consider is c_{rtn1}. The formula system looks as follows:

$$\begin{array}{ll} pc \neq 4 & \\ pc = 3 \wedge h \wedge \neg f & \\ z_{1,1} \rightarrow pc + 1 \neq 4 & z_{1,2} \rightarrow pc + 1 = 4 \\ z_{2,1} \rightarrow pc + 1 \neq 4 & z_{2,2} \rightarrow pc + 1 = 4 \\ z_{1,1} \vee z_{1,2} & \\ z_{2,1} \vee z_{2,2} & \end{array}$$

This time, the solver will return $z_{1,2} = z_{2,2} = 1$ as the only solution. Intuitively, this is due to the fact that all states satisfying the guard of c_{rtn1} must have $pc = 3$, which means that, after executing either one of the assignments, the value of pc will be 4 and thus will lead to a state in block B_2. As both updates lead to B_2, the corresponding quotient distribution in these states is given by:

$$\mu_2(B_1) = 0.0 \text{ and } \mu_2(B_2) = 1.0$$

which results in:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\}, \\ & \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f\} \qquad \}. \end{aligned}$$

Apart from the guard, the formula system for the next command c_{rtn2} is the same as the one before and also has the same solution, which means that these states also have μ_2 as the outgoing quotient distribution. This updates sig to:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\}, \\ & \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f, \\ & \qquad pc \neq 4 \wedge pc = 3 \wedge \neg h \wedge \neg f\} \qquad \}. \end{aligned}$$

The next command is c_{rest}, which leads to $S_{\Pi, c_{rest}}$ as follows:

$$\begin{array}{ll} pc \neq 4 & \\ pc = 3 \wedge f & \\ z_{1,1} \rightarrow 1 \neq 4 & z_{1,2} \rightarrow 1 = 4 \\ z_{2,1} \rightarrow pc + 1 \neq 4 & z_{2,2} \rightarrow pc + 1 = 4 \\ z_{1,1} \vee z_{1,2} & \\ z_{2,1} \vee z_{2,2} & \end{array}$$

for which the solver identifies $z_{1,1} = 1$ and $z_{2,2} = 1$ as the only solution. Intuitively, this says that for all states $s \in B_1$ that have this command enabled, the first assignment will lead back into B_1 while the second assignment leads into B_2, which is obvious considering that the first assignment resets pc to 1 and the second assignment increases pc from 3 to 4. This means that all states $s \in B_1$ with $s \models guard(c_{rest})$ possess the quotient distribution μ_3 with:

$$\mu_3(B_1) = 0.99 \text{ and } \mu_3(B_2) = 0.01$$

which updates sig to:

$$\begin{aligned} sig = \{ \quad & \mu_1 \mapsto \{pc \neq 4 \wedge pc = 1 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4, \\ & \qquad pc \neq 4 \wedge pc = 2 \wedge pc + 1 \neq 4 \wedge pc + 1 \neq 4\}, \\ & \mu_2 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge h \wedge \neg f, \\ & \qquad pc \neq 4 \wedge pc = 3 \wedge \neg h \wedge \neg f\}, \\ & \mu_3 \mapsto \{pc \neq 4 \wedge pc = 3 \wedge f \wedge 1 \neq 4 \wedge pc + 1 = 4\} \qquad \}. \end{aligned}$$

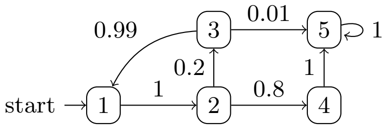

Fig. 3. The bisimulation quotient $[\![P_{Ex}]\!]/\sim$.

Finally, for the last command, namely c_{done}, the formula system is as follows:

$$\begin{array}{ll} pc \neq 4 & pc = 4 \\ z_{1,1} \rightarrow pc \neq 4 & z_{1,2} \rightarrow pc = 4 \\ z_{2,1} \rightarrow pc \neq 4 & z_{2,2} \rightarrow pc = 4 \\ z_{1,1} \vee z_{1,2} & \\ z_{2,1} \vee z_{2,2} & \end{array}$$

where already the two formulas in the first line are (together) unsatisfiable. This reflects the fact that no state in $B_1 = [\![pc \neq 4]\!]$ has this command enabled, because its guard requires pc to be equal to 4. Therefore sig is not updated and as there are no more commands to consider, the loop terminates.

Consequently, B needs to be split into three sub-blocks according to the different quotient distributions recorded in sig. Continuing this for all blocks, we compute the stable partition (expressions have been simplified to improve readability):

$$\begin{aligned} \pi_{Ex}^{sta} = \{ & \underbrace{pc = 1 \wedge \neg f}_{B_1^{sta}}, \underbrace{pc = 2 \wedge \neg f}_{B_2^{sta}}, \underbrace{pc = 3 \wedge f}_{B_3^{sta}}, \\ & \underbrace{pc = 3 \wedge \neg f}_{B_4^{sta}}, \underbrace{pc = 4}_{B_5^{sta}}, \underbrace{pc = 1 \wedge f}_{B_6^{sta}}, \underbrace{pc = 2 \wedge f}_{B_7^{sta}} \}. \end{aligned}$$

For this final partition, we extract the quotient DTMC depicted in Fig. 3. Note that the distributions for the blocks can easily be kept track of during the refinement steps and are thus known. Also note that we already omitted the unreachable blocks B_6^{sta} and B_7^{sta}. □

4 Experiments

Implementation. We implemented a C++-based prototype of our algorithm, using Microsoft's Z3 as the backend SMT solver and comprising about 5000 lines of code. We restrict our attention to PRISM models in which expressions involve linear integer arithmetic, which is typically the case in practice. Also, we use some optimisation techniques to the approach previously described, one of which passes an additional conjunct to the solver that rules out some unreachable blocks in the refinement procedure.

		original DTMC					quotient DTMC		factor	
N	K		hybrid (default)		best				low	high
		states	constr.	verif.	constr.	verif.	states	constr.		
4	9	19817	6.00	88.75	5.99	3.18	10	11.40	0.81	7.78
4	11	44107	41.74	543.85	23.91	14.20	10	26.57	1.43	22.04
5	9	236745	414.91	9456.78	483.86	13.48	12	497.91	1.0	18.99
5	11	644983	4083.82	60784.22	3695.16	45.60	12	1945.99	1.92	33.33
6	9	2659233	TO	TO	53695.52	643.97	14	28548.85	1.90	–

Table 1. Results for the synchronous leader election protocol.

Case studies. We evaluated our implementation on a set of probabilistic model checking benchmarks,[3] running our experiments on a Core i7 processor with 2.6GHz, and limited to 5GB RAM and 48 hours of runtime. If the experiment ran out of time or memory, this is marked as TO and MO, respectively. For the comparisons we considered all three engines of Prism (hybrid, sparse and MTBDD) and give the times for the default (hybrid) and the respective best engine for the corresponding example for both construction (column *constr.*) as well as verification (column *verif.*). Please note that the quotient DTMCs were first computed by our prototype as well as verified by Prism afterwards, but as the verification of the quotient took negligible time (i.e. less than 5ms) for all our experiments, we omitted these entries in the tables.

As it is already known that bisimulation minimisation can lead to drastically smaller state spaces, the key point we want to compare is not the model size in terms of states, but the time needed to verify the properties of interest. For this reason, we list time reduction factors: they are the ratio of the total time consumption of Prism for the given model to the total time needed to minimise the model using our prototype and verify it afterwards using Prism, where the verification, as explained earlier, took virtually no time at all. Since the time Prism needs for the full model strongly depends on the engine used, we list two reduction factors, where the lower is computed with respect to the best and the higher with respect to the worst Prism engine for the given model.

Synchronous leader election protocol. In Itai and Rodeh's protocol, N processors each probabilistically chose one of K different values to pass synchronously around a ring in order to determine a unique leader. We computed the probabilities that a leader is eventually elected and that a leader is elected within L rounds, for $L = 3$. The results are shown in Table 1. The construction and verification of the bisimulation quotient is about as fast as the best (i.e., sparse) Prism engine. However, in comparison to the default Prism engine, we achieve substantial speed-ups using Smt-based quotienting. Note that the quotient system can be used to verify the step-bounded property for arbitrary values of L, as it preserves all PCTL* formulae and does not depend on L. So, verifying this property for more values of L will increase the reduction factors roughly linearly.

[3] All models are available from `http://www.prismmodelchecker.org/casestudies/`.

		original DTMC					quotient DTMC		factor	
N	R		hybrid (default)		best				low	high
		states	constr.	verif.	constr.	verif.	states	constr.		
10	5	110562	0.17	0.48	0.17	0.29	73	1.13	0.58	1.76
10	20	$4.4 \cdot 10^9$	MO	MO	31.18	1623.54	313	21.11	78.39	–
20	5	2036647	174.97	111.59	180.80	8.14	73	2.67	70.71	107.25
20	20	?	MO	MO	MO	MO	313	56.26	–	–
25	5	5508402	MO	MO	MO	MO	73	3.74	–	–
25	20	?	MO	MO	MO	MO	313	80.03	–	–
500	5	?	MO	MO	MO	MO	73	8969.46	–	–
500	20	?	MO	MO	MO	MO	313	106724.27	–	–
600	5	?	MO	MO	MO	MO	73	18219.01	–	–
600	20	?	MO	MO	MO	MO	MO	MO	–	–

Table 2. Impact of bisimulation minimisation on the Crowds protocol model.

Crowds protocol. Reiter and Rubin's Crowds protocol aims to send a message anonymously to a destination by randomly routing it R times through a crowd of size N. For this model, we restricted the reachable blocks by over-approximating the reachable state space by an expression capturing the obvious fact that the total number of member observations cannot exceed the number of instances the protocol has been run. Our model is a slight amendment of the model available on the Prism website with less variables. Table 2 summarizes the results, where we computed the probability that the original sender was discovered more than once. Using our technique, we not only outperform Prism in terms of runtime, but are also able to treat significantly larger model parameters. In fact, for the parameters where the state space size is indicated as unknown ("?"), using Prism we were not able to build the state space let alone perform the actual verification.[4] Here, the crucial advantage of the Smt-based quotienting becomes apparent: since it avoids building the full state space of the original model, it shortens computation times while reducing the required memory.

Comparison with Sigref. In addition to the comparison with Prism, we compared our prototype to Sigref, a tool that performs bisimulation minimisation symbolically on a BDD representation of the system [21]. We integrated Sigref into Prism in a way that works directly on the internal format of the model checker, which was possible because they share the MTBDD library Cudd. Table 3 illustrates the experimental results for both case studies, where the time reduction factor is the ratio of time needed for minimisation using Sigref plus the verification using Prism to the time needed for the minimisation using our Smt-based prototype plus the verification time using Prism. Note that while the quotient DTMCs are isomorphic, the verification times differ between the two approaches, because the BDD representing the (same) system is different. For the first case study, we observe minor speed-ups compared to Sigref. However, due to memory requirements, Sigref was unable to minimise the state space of

[4] We tried to build the state space of the smallest of these models (N=20, R=20) on a cluster with 196GB of memory, but aborted the experiment after one week.

(a) Synchronous leader election

N	K	constr.	verif.	red. factor
4	9	18.39	≈ 0	1.61
4	11	51.57	≈ 0	1.94
5	9	580.40	≈ 0	1.17
5	11	MO	MO	–
6	9	MO	MO	–

(b) Crowds protocol

N	R	constr.	verif.	red. factor
10	5	9.67	≈ 0	8.51
10	20	6100.093	90.26	293.26
20	5	481.022	≈ 0	180.02
20	5	MO	MO	–

Table 3. Comparison with Sigref as the minimisation engine.

the last two models. For the Crowds protocol, note that Sigref needs to build the full BDD representation of the state space prior to minimisation. As the time needed for model construction dominates the runtime, there is (almost) no scope for Sigref to improve on Prism's runtimes. Even worse, the additional intermediate BDDs prevented the minimisation for the parameters 20/5 under the memory restriction.

Possible extensions. We conclude this section with an overview of several ways that our SMT-based approach to bisimulation can be extended.

Rewards. A Markov Reward Model (MRM) $(\mathcal{D}, r)$ is a DTMC $\mathcal{D}$ equipped with a function $r\colon S \to \mathbb{R}_{\geq 0}$ that assigns a non-negative real value to each state of $\mathcal{D}$. Upon passing through state s the reward $r(s)$ is gained, providing a quantitative measure in addition to the probabilities. In the Prism modelling language, state-based rewards are defined in a similar fashion as commands, essentially attaching a reward expression e to all states satisfying a given Boolean expression b. If e is an expression evaluating to a constant, i.e., all states satisfying b share the same reward value, then rewards can be easily supported by our implementation by adjusting the initial partition appropriately.

Nondeterminism. In its current form, both the algorithm and the implementation treat only DTMCs and CTMCs and do not support their nondeterministic counterparts MDPs and CTMDPs, respectively. Our prototype can be extended in order to also support these models. For this, we lift the formula system $S_{\Pi,c}$ of section 3 to a system S_Π incorporating all commands with additional auxiliary variables x_c for $c \in \mathit{Comm}$ as follows:

$$b \tag{8}$$

$$x_c \leftrightarrow \mathit{guard}(c) \text{ for all } c \in \mathit{Comm} \tag{9}$$

$$z_{c,i,j} \rightarrow \mathit{wp}(b_j, E_{c,i}) \text{ for all } c \in \mathit{Comm}, \text{ all } 1 \leq j \leq k \text{ and all } 1 \leq i \leq |c| \tag{10}$$

$$x_c \rightarrow \bigvee\nolimits_{j=1}^{k} z_{c,i,j} \text{ for all } c \in \mathit{Comm} \text{ and } 1 \leq i \leq |c| \tag{11}$$

where the current partition $\Pi = [\![\pi]\!]$ is given by $\pi = \{b_1, \ldots, b_k\}$, $|c|$ refers to the number of assignments of c and $E_{c,i}$ denotes the ith assignment of c. Enumerating the satisfying assignments will now induce sets of simultaneously enabled commands and the corresponding target block combinations.

Parametric Markov chains. If the probabilities in a given probabilistic program are not concrete values but rather *parameters*, it corresponds to a parametric Markov chain [8] instead of a DTMC. As the only part of our algorithm that deals with probabilities is the computation of the probability distribution induced by a command and a target block combination, it is fairly straightforward to incorporate parameters. Instead of computing a concrete value associated with each successor block, we symbolically derive an expression involving the parameters and only consider two parametric probability distributions equal if they syntactically coincide (in a certain normal form). This way, the computed quotient preserves all PCTL* properties for *all* possible parameter values.

5 Conclusion and Further Work

We have presented an SMT-based approach to extract, from a probabilistic program specified in the PRISM modelling language, the Markov chain representing its coarsest bisimulation quotient. No state space is generated—our bisimulation minimisation is a truly symbolic program manipulation. Experiments yielded encouraging results, even without optimisations such as formula simplification, which we plan to incorporate in future work. Application of the SMT-based approach to either parametric programs or programs with infinite state space but finite bisimulation quotient is straightforward and the approach can easily be adapted to perform compositional minimisation. We therefore believe that this approach represents a promising alternative to enumerative and BDD-based bisimulation minimisation algorithms.

Acknowledgements. The first two authors are funded by the EU FP7 project CARP (see http://www.carpproject.eu/) and the work was part-funded by the ERC Advanced Grant VERIWARE. We also thank Marta Kwiatkowska for facilitating the first author's visit to Oxford, where this work was initiated.

References

1. A. Aziz, V. Singhal, and F. Balarin. It usually works: The temporal logic of stochastic systems. In *CAV*, volume 939 of *LNCS*, pages 155–165, 1995.
2. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking c programs. In T. Margaria and W. Yi, editors, *TACAS*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
3. P. Buchholz. Exact and ordinary lumpability in finite Markov chains. *Journal of Applied Probability*, 31:59–75, 1994.
4. N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, and W. Serwe. Ten years of performance evaluation for concurrent systems using CADP. In *ISoLA (2)*, pages 128–142, 2010.
5. S. Derisavi, H. Hermanns, and W. H. Sanders. Optimal state-space lumping in Markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.
6. K. Fisler and M. Y. Vardi. Bisimulation minimization and symbolic model checking. *Formal Methods in System Design*, 21(1):39–78, 2002.

7. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In O. Grumberg, editor, *CAV*, volume 1254, pages 72–83. Springer, 1997.
8. E. M. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric Markov models. *STTT*, 13(1):3–19, 2011.
9. H. Hermanns, B. Wachter, and L. Zhang. Probabilistic CEGAR. In A. Gupta and S. Malik, editors, *CAV*, volume 5123 of *LNCS*, pages 162–175. Springer, 2008.
10. J.-P. Katoen, T. Kemna, I. S. Zapreev, and D. N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *LNCS*, pages 87–101. Springer, 2007.
11. J.-P. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Perf. Ev.*, 68(2):90–104, 2011.
12. M. Kattenbelt, M. Z. Kwiatkowska, G. Norman, and D. Parker. Game-based probabilistic predicate abstraction in PRISM. *Electr. Notes Theor. Comput. Sci.*, 220(3):5–21, 2008.
13. M. Kattenbelt, M. Z. Kwiatkowska, G. Norman, and D. Parker. A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods in System Design*, 36(3):246–280, 2010.
14. M. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In *CAV*, volume 4144 of *LNCS*, pages 234–248, 2006.
15. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591, 2011.
16. K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.
17. M. Mumme and G. Ciardo. A fully symbolic bisimulation algorithm. In *Workshop on Reachability Problems (RP)*, volume 6945 of *LNCS*, pages 218–230, 2011.
18. A. Valmari and G. Franceschinis. Simple $O(m \log n)$ time Markov chain lumping. In *TACAS*, volume 6015 of *LNCS*, pages 38–52, 2010.
19. B. Wachter, L. Zhang, and H. Hermanns. Probabilistic model checking modulo theories. In *QEST*, pages 129–140, 2007.
20. R. Wimmer, S. Derisavi, and H. Hermanns. Symbolic partition refinement with automatic balancing of time and space. *Perform. Eval.*, 67(9):816–836, 2010.
21. R. Wimmer, M. Herbstritt, H. Hermanns, K. Strampp, and B. Becker. Sigref- a symbolic bisimulation tool box. In *ATVA*, volume 4218 of *LNCS*, 2006.