





Out of Control: Reducing Probabilistic Models by Control-State Elimination

Tobias Winkler^(✉) ,
Johannes Lehmann ,
and Joost-Pieter Katoen 

RWTH Aachen University,
Aachen, Germany
{tobias.winkler,
katoen}@cs.rwth-aachen.de,
johannes.lehmann@rwth-aachen.de



Abstract. State-of-the-art probabilistic model checkers perform verification on explicit-state Markov models defined in a high-level programming formalism like the PRISM modeling language. Typically, the low-level models resulting from such program-like specifications exhibit lots of structure such as repeating subpatterns. Established techniques like probabilistic bisimulation minimization are able to exploit these structures; however, they operate directly on the explicit-state model. On the other hand, methods for reducing structured state spaces by reasoning about the high-level program have not been investigated that much. In this paper, we present a new, simple, and fully automatic program-level technique to reduce the underlying Markov model. Our approach aims at computing the summary behavior of adjacent locations in the program’s control-flow graph, thereby obtaining a program with fewer “control states”. This reduction is immediately reflected in the program’s operational semantics, enabling more efficient model checking. A key insight is that in principle, each (combination of) program variable(s) with finite domain can play the role of the program counter that defines the flow structure. Unlike most other reduction techniques, our approach is property-directed and naturally supports unspecified model parameters. Experiments demonstrate that our simple method yields state-space reductions of up to 80% on practically relevant benchmarks.

1 Introduction

Modelling Markov Models. Probabilistic model checking is a fully automated technique to rigorously prove correctness of a system model with randomness against a formal specification. Its key algorithmic component is computing reachability probabilities on stochastic processes such as (discrete- or continuous-time) Markov chains and Markov Decision Processes. These stochastic processes are

This work is supported by the Research Training Group 2236 UnRAVeL, funded by the German Research Foundation.

typically described in some high-level modelling language. State-of-the-art tools like PRISM [33], storm [26] and mcsta [24] support input models specified in e.g., the PRISM modeling language¹, PPDDL [42], a probabilistic extension of the planning domain definition language [22], the process algebraic language MoDeST [9], the jani model exchange format [11], or the probabilistic guarded command language pGCL [34]. The recent tool from [21] even supports verification of probabilistic models written in Java.

Model Construction. Prior to computing reachability probabilities, existing model checkers explore all the program’s reachable variable valuations and encode them into the state space of the operational Markov model. Termination is guaranteed as variables are restricted to finite domains. This paper proposes a simple reduction technique for this model construction phase that avoids unfolding the full model *prior to* the actual analysis, thereby mitigating the state explosion problem. The basic idea is to unfold variables one-by-one—rather than all at once as in the standard pipeline—and apply analysis steps after each unfolding. We detail this *control-state reduction* technique for probabilistic control-flow graphs and illustrate its application to the PRISM modelling language. Its principle is however quite generic and is applicable to the aforementioned modelling formalisms. Our technique is thus to be seen as a model simplification front-end for general purpose probabilistic model checkers.

Approach. Technically our approach works as follows. The principle is to unfold a (set of) variable(s) into the control state space, a technique inspired by static program analyses such as abstract interpretation [28]. The selection of which variables to unfold is property-driven, i.e., depending on the reachability or reward property to be checked. We define the unfolding on probabilistic control-flow programs [19] (PCFPs, for short) and simplify them using a technique that generalizes *state elimination* in (parametric) Markov chains [13]. Our elimination technique heavily relies on classical *weakest precondition reasoning* [16]. This enables the elimination of several states at once from the underlying “low-level” Markov model while preserving *exact* reachability probabilities or expected rewards. Figure 1 provides a visual intuition on the resulting model compression.

The choice of the variables and locations for unfolding and elimination, resp., is driven by heuristics. In a nutshell, our unfolding heuristics prefers the variables that lead to a high number of control-flow locations without self-loops. These loop-free locations are then removed by the elimination heuristics which gives preference to locations whose removal does not blow up the transition matrix of the underlying model. Unfolding and elimination steps are performed in an alternating fashion, but only until the PCFP size reaches a certain threshold. After this, the reduction phase is complete and the transformed PCFP can be fed into a standard probabilistic model checker.

Contributions. In summary, the main contributions of this paper are:

- A simple, widely applicable reduction technique that considers each program variable with finite domain as a “program counter” and selects suitable variables for unfolding into the control state space one-by-one.

¹ <https://www.prismmodelchecker.org/manual/ThePRISMLanguage>.

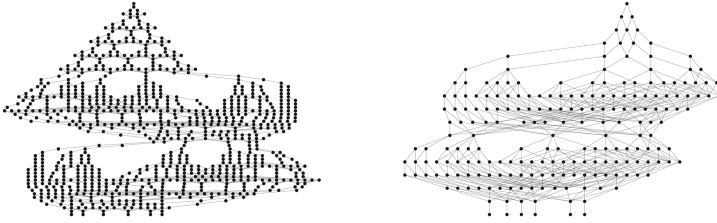


Fig. 1. Left: Visualization of the original NAND model from [35] (930 states, parameters 5/1). Transitions go from top to bottom. Right: The same model after our reduction (207 states). A single “program counter variable” taking at most 5 different values was unfolded and a total of three locations were eliminated thereafter. Note that the overall structure is preserved but several *local* substructures such as the pyramidal shape at the top are compressed significantly. This behavior is typical for our approach.

- A sound rule to eliminate control-flow locations in PCFPs in order to shrink the state space of the underlying Markov model while preserving *exact* reachability probabilities or expected rewards.
- Elimination in PCFPs—in contrast to Markov chains—is shown to have an exponential worst-case complexity.
- An implementation in the probabilistic model checker *storm* demonstrating the potential to significantly compress practically relevant benchmarks.

Related Work. The state explosion problem has been given top priority in both classical and probabilistic model checking. Techniques similar to ours have been known for quite some time in the non-probabilistic setting [18,32]. Regarding probabilistic model checking, reduction methods on the state-space level include symbolic model checking using MTBDDs [1], SMT/SAT techniques [7,40], bisimulation minimization [27,30,38], Kronecker representations [1,10] and partial order reduction [4,12]. Language-based reductions include symmetry reduction [17], bisimulation reduction using SMT on PRISM modules [15], as well as abstraction-refinement techniques [23,31,39]. Our reductions on PCFPs are inspired by state elimination [13]. Similar kinds of reductions on probabilistic workflow nets have been considered in [20]. Despite all these efforts, it is somewhat surprising that simple probabilistic control-flow reductions as proposed in this paper have not been investigated that much. A notable exception is the recent work by Dubslaff *et al.* that applies existing static analyses to control-flow-rich PCFPs [19]. In contrast to our method, their technique yields bisimilar models and exploits a different kind of structure.

Organization of the Paper. Section 2 starts off by illustrating the central aspects of our approach by example. Section 3 defines PCFPs and their semantics in terms of MDPs. Section 4 formalizes the reductions, proves their correctness and analyzes the complexity. Our implementation in *storm* is discussed in Sect. 5. We present our experimental evaluation in Sect. 6 and conclude in Sect. 7. A *full version* of this paper including detailed proofs is available online [41].

```

dtmc
const int N;
module coingame
  x : [0..N+1] init N/2;
  f : bool init false;
  [] 0<x & x<N & !f -> 1/2: (x'=x-1)          + 1/2: (f'=true);
  [] 0<x & x<N & f  -> 1/2: (x'=x-1) & (f'=false) + 1/2: (x'=x+2) & (f'=false);
  [] x=0 | x>=N     -> 1:   (f'=false);
endmodule

```

Fig. 2. The coin game as a PRISM program. Variable x stands for the current budget.

2 A Bird’s Eye View

This section introduces a running example to illustrate our approach. Consider a game of chance where a gambler starts with an initial budget of $x = N/2$ tokens. The game is played in rounds, each of which either increases or decreases the budget. The game is lost once the budget has dropped to zero and won once it exceeds N tokens. In each round, a fair coin is tossed: If the outcome is tails, then the gambler loses one token and proceeds to the next round; on the other hand, if heads occurs, then the coin is flipped again. If tails is observed in the second coin flip, then the gambler also loses one token; however, if the outcome is again heads then the gambler receives *two* tokens.

In order to answer questions such as “Is this game fair?” (for a fixed N), probabilistic model checking can be applied. To this end, we model the game as the PRISM program in Fig. 2. We briefly explain its central components: The first two lines of the `module` block are variable declarations. Variable x is an integer with bounded domain and f is a Boolean. The idea of x and f is to represent the current budget and whether the coin has to be flipped a second time, respectively. The next three lines that each begin with `[]` define *commands* which are interpreted as follows: If the *guard* on the left-hand side of the arrow `->` is satisfied, then one of the updates on the right side is executed with its corresponding probability. For instance, in the first command, x is decremented by one (and f is left unchanged) with probability $1/2$. Otherwise f is set to true. The order in which the commands occur in the program text is irrelevant. If there is more than one command enabled for a specific valuation of the variables, then one of them is chosen non-deterministically. Our example is, however, *deterministic* in this regard since the three guards are mutually exclusive.

Probabilistic model checkers like PRISM and storm expand the above program as a Markov chain with approximately $2N$ states. This is depicted for $N = 6$ at the top of Fig. 3. Given that we are only interested in the winning probability (i.e., to reach one of the two rightmost states), this Markov chain is equivalent to the smaller one on the bottom of Fig. 3. Indeed, *eliminating* each dashed state in the lower row individually yields that the overall probability per round to go one step to the left is $3/4$ and $1/4$ to go two steps to the right. On the program level, this simplification could have been achieved by summarizing the first two commands to

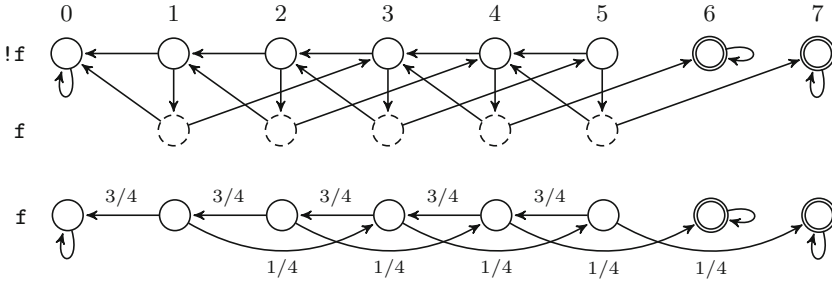


Fig. 3. Top: The Markov chain of the original coin game for $N = 6$. All transition probabilities (except on the self-loops) are $1/2$. Bottom: The Markov chain of the simplified model.

$$\square \quad 0 < x \ \& \ x < N \ \rightarrow \ 3/4: (x' = x - 1) \ + \ 1/4: (x' = x + 2);$$

so that variable f is effectively removed from the program.

Obtaining such simplifications in an *automated* manner is the main purpose of this paper. In summary, our proposed solution works as follows:

1. First, we view the input program as a probabilistic control flow program (PCFP), which can be seen as a generalization of PRISM programs from a single to multiple control-flow locations (Fig. 4, left). A PRISM program (with a single module) is a PCFP with a unique control location. Imperative programs such as *pGCL* programs [34] can be regarded as PCFPs with roughly one location per line of code.
2. We then *unfold* one or several variables into the location space, thereby interpreting them as “program counters”. We will discuss in Sect. 4.1 that—in principle—every variable can be unfolded in this way. The distinction between program counters and “data variables” is thus an informal one. This insight renders the approach quite flexible. In the example, we unfold f (Fig. 4, middle), but we stress that it is also possible to unfold x instead (for any fixed N), even though this is not as useful in this case.
3. The last and most important step is *elimination*. Once sufficiently unfolded, we identify locations in the PCFP that can be eliminated. Our elimination rules are inspired by state elimination in Markov chains [13]. In the example, we eliminate the location labeled f . To this end, we try to eliminate all ingoing transitions of location f . Applying the rules described in detail in Sect. 4, we obtain the PCFP shown in Fig. 4 (right). This PCFP generates the reduced Markov chain in Fig. 3 (bottom). Here, location elimination has also reduced the size of the PCFP, but this is not always the case. In general, elimination adds more commands to the program while reducing the size of the generated Markov chain or MDP (cf. Sect. 6).

These unfolding and elimination steps may be performed in an alternating fashion following the principle “*unfold a bit, eliminate reasonably*”. Here, “reason-

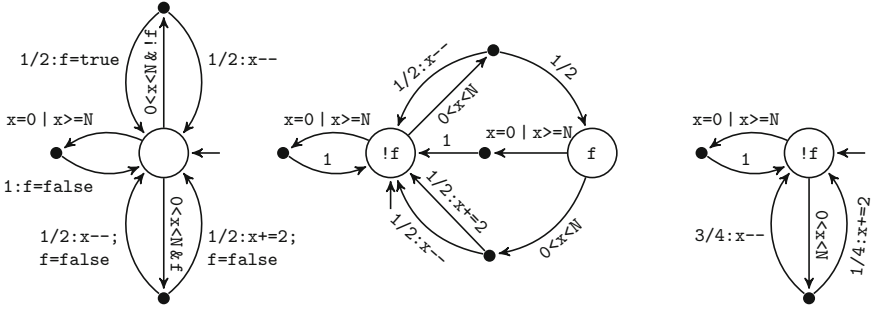


Fig. 4. Left: The coin game as a single-location PCFP \mathfrak{P}_{game} . Middle: The PCFP after unfolding variable f . Right: The PCFP after eliminating the location labeled f .

ably” means that in particular, we must be careful to not blow up the underlying transition matrix (cf. Sect. 5).

Despite its simplicity, we are not aware of any other automatic technique that achieves the same or similar reductions on the coin game model. In particular, bisimulation minimization is not applicable: The bisimulation quotient of the Markov chain in Fig. 3 (top) is already obtained by merging just the two rightmost goal states.

Arguably, the program transformations in the above example could have been done by hand. However, automation is crucial for our technique because the transformation makes the program harder to understand and obfuscates the original model’s mechanics due to the removed intermediate control states. Indeed, *simplification* only takes place from the model checker’s perspective but not from the programmer’s. Moreover, our transformations are rather tedious and error-prone, and may not always be that obvious for more complicated programs. To illustrate this, we mention the work [35] where a PRISM model of the von Neumann NAND multiplexing system was presented. Optimizations with regard to the resulting state space were applied manually already at modeling time². Despite these (successful) manual efforts, our fully automatic technique can further shrink the state-space of the same model by $\approx 80\%$ (cf. Sect. 6).

3 Technical Background on PCFPs

In this section, we review the necessary definitions of Markov Decision Processes (MDPs), Probabilistic Control Flow Programs (PCFPs), and reachability properties. The set of probability distributions on a finite set S is denoted $\text{Dist}(S) = \{p: S \rightarrow [0, 1] \mid \sum_{s \in S} p(s) = 1\}$. The set of (total) functions $A \rightarrow B$ is denoted B^A .

² See paragraph 7 in [35, Sec. III A.].

Basic Markov Models. An *MDP* is tuple $\mathcal{M} = (S, \text{Act}, \iota, P)$ where S is a finite set of states, $\iota \in S$ is an initial state, Act is a finite set of action labels and $P: S \times \text{Act} \dashrightarrow \text{Dist}(S)$ is a (partial) probabilistic transition function. We say that action $a \in \text{Act}$ is *available* at state $s \in S$ if $P(s, a)$ is defined. We use the notation $s \xrightarrow{a, p} s'$ to indicate that $P(s, a)(s') = p$. In the following, we write $P(s, a, s')$ rather than $P(s, a)(s')$.

A *Markov chain* is an MDP with exactly one available action at every state. We omit action labels when considering Markov chains, i.e., the transition function of a Markov chain has type $P: S \rightarrow \text{Dist}(S)$. Given a Markov chain \mathcal{M} together with a goal set $G \subseteq S$, we define the set of paths reaching G as $\text{Paths}(G) = \{s_0 \dots s_n \in S^n \mid n \geq 0, s_0 = \iota, s_n \in G, \forall i < n: s_i \notin G\}$. The *reachability probability* of G is $\mathbb{P}_{\mathcal{M}}(\diamond G) = \sum_{\pi \in \text{Paths}(G)} \prod_{i=0}^{l(\pi)-1} P(\pi_i, \pi_{i+1})$ where $l(\pi)$ denotes the length of a path π and π_i is the i -th state along π . $\mathbb{P}(\diamond G)$ is always a well-defined probability (see e.g. [5, Ch. 10] for more details).

A (memoryless deterministic) *scheduler* of an MDP is a mapping $\sigma \in \text{Act}^S$ with the restriction that action $\sigma(s)$ is available at s . Each scheduler σ induces a Markov chain \mathcal{M}^σ by retaining only the action $\sigma(s)$ at every $s \in S$. Scheduler σ is called *optimal* if $\sigma = \text{argmax}_{\sigma'} \mathbb{P}_{\mathcal{M}^{\sigma'}}(\diamond G)$ (or *argmin*, depending on the context). In finite MDPs as considered here, there always exists an optimal memoryless and deterministic scheduler, even if the above *argmax* is taken over more general schedulers that may additionally use memory and/or randomization [36].

PCFP Syntax and Semantics. We first define (guarded) commands. Let $\text{Var} = \{x_1, \dots, x_n\}$ be a set of integer-valued variables. An *update* is a set of assignments

$$u = \{x'_1 = f_1(x_1, \dots, x_n), \dots, x'_n = f_n(x_1, \dots, x_n)\}$$

that are executed *simultaneously*. We assume that the expressions f_i always yield integers. An update u transforms a *variable valuation* $\nu \in \mathbb{Z}^{\text{Var}}$ into a valuation $\nu' = u(\nu)$. For technical reasons, we also allow *chaining* of updates, that is, if u_1 and u_2 are updates, then $u_1 \mathbin{\&} u_2$ is the update that corresponds to executing the updates in sequence: first u_1 and then u_2 . A *command* is an expression

$$\varphi \rightarrow p_1 : u_1 + \dots + p_k : u_k,$$

where φ is a *guard*, i.e., a Boolean expression over program variables, u_i are updates, and p_i are non-negative real numbers such that $\sum_{i=1}^k p_i = 1$, i.e., they describe a probability distribution over the updates. We further define *location-guided* commands which additionally depend on *control-flow locations* l and l_1, \dots, l_k :

$$\varphi, l \rightarrow p_1 : u_1 : l_1 + \dots + p_k : u_k : l_k.$$

The intuitive meaning of a location-guided command is as follows: It is enabled if the system is at location l and the current variable valuation satisfies φ . Based on the probabilities p_1, \dots, p_k , the system then randomly executes one

of the updates u_i and transitions to the next location l_i . We use the notation $l \xrightarrow{\varphi \rightarrow p_i : u_i} l_i$ to refer to such a possible *transition* between locations. We call location-guided commands simply *commands* in the rest of the paper.

Probabilistic Control Flow Programs (PCFPs) combine several commands into a probabilistic program and constitute the formal basis of our approach:

Definition 1 (PCFP). *A PCFP is a tuple $\mathfrak{P} = (\text{Loc}, \text{Var}, \text{dom}, \text{Cmd}, \iota)$ where Loc is a non-empty set of (control-flow) locations, Var is a set of integer-valued variables, $\text{dom} \in \mathcal{P}(\mathbb{Z})^{\text{Var}}$ is a domain for each variable, Cmd is a set of commands as defined above, and $\iota = (l_\iota, \nu_\iota)$ is the initial location/valuation pair.*

This definition and our notation for commands are similar to [19]. We also allow Boolean variables as *syntactic sugar* by identifying **false** $\equiv 0$ and **true** $\equiv 1$. We generally assume that Loc and all variable domains are *finite* sets. For a variable valuation $\nu \in \mathbb{Z}^{\text{Var}}$, we write $\nu \in \text{dom}$ if $\nu(x) \in \text{dom}(x)$ for all $x \in \text{Var}$. In some occasions, we consider only *partial* valuations $\nu \in \mathbb{Z}^{\text{Var}'}$, where $\text{Var}' \subsetneq \text{Var}$. We use the notations $\varphi[\nu]$ and $u[\nu]$ to indicate that all variables occurring in the guard φ (the update u , respectively) are replaced according to the given (partial) valuation ν . For updates, we also remove assignments whose left-hand side variables become a constant. Recall that the notation $u(\nu)$ has a different meaning; it denotes the result of executing the update u on valuation ν .

The straightforward operational semantics of a PCFP is defined in terms of a Markov Decision Process (MDP).

Definition 2 (MDP Semantics). *For a PCFP $\mathfrak{P} = (\text{Loc}, \text{Var}, \text{dom}, \text{Cmd}, \iota)$, we define the semantic MDP $\mathcal{M}_{\mathfrak{P}} = (S, \text{Act}, \iota, P)$ as follows:*

$$S = \text{Loc} \times \{\nu \in \text{dom}\} \cup \{\perp\}, \quad \text{Act} = \{a_\gamma \mid \gamma \in \text{Cmd}\}, \quad \iota = \langle l_\iota, \nu_\iota \rangle$$

and the probabilistic transition relation P is defined according to the rules

$$\frac{l_1 \xrightarrow{\varphi \rightarrow p : u} l_2 \wedge \nu \models \varphi \wedge u(\nu) \in \text{dom}}{\langle l_1, \nu \rangle \xrightarrow{a_\gamma, p} \langle l_2, u(\nu) \rangle}, \quad \frac{l_1 \xrightarrow{\varphi \rightarrow p : u} l_2 \wedge \nu \models \varphi \wedge u(\nu) \notin \text{dom}}{\langle l_1, \nu \rangle \xrightarrow{a_\gamma, p} \perp}$$

where $a_\gamma \in \text{Act}$ is an action label that uniquely identifies the command γ containing transition $l_1 \xrightarrow{\varphi \rightarrow p : u} l_2$.

An element $\langle l, \nu \rangle \in \text{Loc} \times \{\nu \in \text{dom}\}$ is called a *configuration*. A PCFP is *deterministic* if the MDP $\mathcal{M}_{\mathfrak{P}}$ is a Markov chain. Moreover, we say that a PCFP is *well-formed* if the out-of-bounds state \perp is not reachable from the initial state and if there is at least one action available at each state of $\mathcal{M}_{\mathfrak{P}}$. From now on, we assume that PCFPs are always well-formed.

Example 1. The semantic MDP—a Markov chain in this case—of the two PCFPs in Fig. 4 (left and middle) is given in Fig. 3 (top), and the one of the PCFP in Fig. 4 (right) is depicted in Fig. 3 (bottom). \triangle

Reachability in PCFPs. It is natural to describe a set of good (or bad) PCFP configurations by means of a predicate ϑ over the program variables which defines a set of target states in the semantic MDP $\mathcal{M}_{\mathfrak{P}}$. We slightly extend this to account for information available from previous unfolding steps. To this end, we will sometimes consider a labeling function $L: \text{Loc} \rightarrow \mathbb{Z}^{\text{Var}'}$ that assigns to each location an additional variable valuation ν' over Var' , a set of variables *disjoint* to the actual programs variables Var . The idea is that Var' contains the variables that have already been unfolded (see Sect. 4.1 below for the details). A predicate ϑ over $\text{Var} \uplus \text{Var}'$ describes the following goal set in the MDP $\mathcal{M}_{\mathfrak{P}}$:

$$G_{\vartheta} = \{ \langle l, \nu \rangle \mid l \in \text{Loc}, \nu \in \text{dom}, (\nu, L(l)) \models \vartheta \}$$

where $(\nu, L(l))$ is the variable valuation over $\text{Var} \uplus \text{Var}'$ that results from combining ν and $L(l)$.

Definition 3 (Potential Goal). *Let $(\text{Loc}, \text{Var}, \text{dom}, \text{Cmd}, \iota)$ be a PCFP labeled with valuations $L: \text{Loc} \rightarrow \mathbb{Z}^{\text{Var}'}$ and let ϑ be a predicate over $\text{Var} \uplus \text{Var}'$. A location $l \in \text{Loc}$ is called a potential goal w.r.t. ϑ if $\vartheta[L(l)]$ is satisfiable in dom .*

Example 2. Consider the PCFP in Fig. 4 (middle) with $N = 6$. Note that here, $\text{Var} = \{\mathbf{x}\}$ and $\text{Var}' = \{\mathbf{f}\}$. Let $\vartheta = (\mathbf{x} \geq 6 \wedge \mathbf{f} = \text{false})$. Assume the labeling function $L(!\mathbf{f}) = \{\mathbf{f} \mapsto \text{false}\}$ and $L(\mathbf{f}) = \{\mathbf{f} \mapsto \text{true}\}$. Then the location labeled $!\mathbf{f}$ is a potential goal w.r.t. ϑ because $\vartheta[\mathbf{f} \mapsto \text{false}] \equiv \mathbf{x} \geq 6$ is satisfiable. The other location \mathbf{f} is no potential goal. \triangle

In Sect. 4 below, we introduce PCFP transformation rules that preserve reachability probabilities. This is formally defined as follows:

Definition 4 (Reachability Equivalence). *Let \mathfrak{P}_1 and \mathfrak{P}_2 be PCFPs over the same set of variables Var . For $i \in \{1, 2\}$, let $L_i: \text{Loc}_i \rightarrow \mathbb{Z}^{\text{Var}'}$ be labeling functions on \mathfrak{P}_i . Further, let ϑ be a predicate over $\text{Var} \uplus \text{Var}'$. Then \mathfrak{P}_1 and \mathfrak{P}_2 are ϑ -reachability equivalent if*

$$\text{opt}_{\sigma} \mathbb{P}_{\mathcal{M}_{\mathfrak{P}_1}}^{\sigma}(\diamond G_{\vartheta}) = \text{opt}_{\sigma} \mathbb{P}_{\mathcal{M}_{\mathfrak{P}_2}}^{\sigma}(\diamond G_{\vartheta})$$

for both $\text{opt} \in \{\min, \max\}$ and where σ ranges of the class of memoryless deterministic schedulers for the MDPs $\mathcal{M}_{\mathfrak{P}_1}^{\sigma}$ and $\mathcal{M}_{\mathfrak{P}_2}^{\sigma}$, respectively.

Example 3. For all $N \geq 0$, the PCFPs in Fig. 4 (middle) and Fig. 4 (right) with labeling functions as in Example 2 are reachability equivalent w.r.t. to $\vartheta = (\mathbf{x} \geq N \wedge \mathbf{f} = \text{false})$. This follows from our intuitive explanation in Sect. 2, or alternatively from the formal rules to be presented in the following Sect. 4. \triangle

4 PCFP Reduction

We now describe our two main ingredients in detail: variable *unfolding* and location *elimination*. Throughout this section, $\mathfrak{P} = (\text{Loc}, \text{Var}, \text{dom}, \text{Cmd}, \iota)$ denotes an arbitrary well-formed PCFP.

4.1 Variable Unfolding

Let Asgn be the set of all assignments that occur anywhere in the updates of \mathfrak{P} . For an assignment $\alpha \in \text{Asgn}$, we write $\text{lhs}(\alpha)$ for the variable on the left-hand side and $\text{rhs}(\alpha)$ for the expression on the right-hand side. Let $x, y \in \text{Var}$ be arbitrary. Define the relation $x \rightarrow y$ (“ x depends on y ”) as

$$x \rightarrow y \quad \iff \quad \exists \alpha \in \text{Asgn}: \quad x = \text{lhs}(\alpha) \quad \wedge \quad \text{rhs}(\alpha) \text{ contains } y.$$

This syntactic dependency relation only takes updates but no guards into account. This is, however, sufficient for our purpose. We say that x is (*directly*) *unfoldable* if $\forall y: x \rightarrow y \implies x = y$, that is, x depends at most on itself.

Example 4. Variables \mathbf{x} and \mathbf{f} in the PCFP in Fig. 4 (left) are unfoldable. \triangle

The rationale of this definition is as follows: If variable x is to be unfolded into the location space, then we must make sure that any update assigning to x yields an explicit numerical value and hence an unambiguous location. Formally, unfolding is defined as follows:

Definition 5 (Unfolding). *Let $x \in \text{Var}$ be unfoldable. The unfolding $\text{Unf}(\mathfrak{P}, x)$ of \mathfrak{P} with respect to x is the PCFP $(\text{Loc}', \text{Var} \setminus \{x\}, \text{dom}, \text{Cmd}', \iota')$ where*

$$\text{Loc}' = \text{Loc} \times \text{dom}(x), \quad \iota' = (\langle l, \nu_i(x) \rangle, \nu'_i)$$

where $\nu'_i(x) = \nu_i(x)$ for all $x \in \text{Var}'$, and Cmd' is defined according to the rule

$$\frac{l \xrightarrow{\varphi \rightarrow p:u} l' \text{ in } \mathfrak{P} \quad \wedge \quad \nu: \{x\} \rightarrow \text{dom}(x)}{\langle l, \nu(x) \rangle \xrightarrow{\varphi[\nu] \rightarrow p:u[\nu]} \langle l', u(\nu)(x) \rangle}.$$

Recall that $u[\nu]$ substitutes all x in u for $\nu(x)$ while $u(\nu)$ applies u to valuation ν . Note that even though ν only assigns a value to x in the above rule, we nonetheless have that $u(\nu)(x)$ is a well-defined integer in $\text{dom}(x)$. This is ensured by the definition of unfoldable and because \mathfrak{P} is well-formed. Unfolding preserves the semantics of a PCFP (up to renaming of states and action labels):

Lemma 1. *For every unfoldable $x \in \text{Var}$, we have $\mathcal{M}_{\text{Unf}(\mathfrak{P}, x)} = \mathcal{M}_{\mathfrak{P}}$.*

Example 5. The PCFP in Fig. 4 (middle) is the unfolding $\text{Unf}(\mathfrak{P}_{\text{game}}, \mathbf{f})$ of the PCFP $\mathfrak{P}_{\text{game}}$ in Fig. 4 (left) with respect to variable \mathbf{f} . \triangle

In general, it is possible that no single variable of a PCFP is unfoldable. We offer two alternatives for such cases:

- There always exists a set $U \subseteq \text{Var}$ of variables that can be unfolded *at once* ($U = \text{Var}$ in the extreme case). Definition 5 can be readily adapted to this case. Preferably small sets of unfoldable variables can be found by considering the bottom SCCs of the directed graph $(\text{Var}, \rightarrow)$.
- In principle, each variable can be made unfoldable by introducing further commands. Consider for instance a command γ with an update $x' = y$. We may introduce $|\text{dom}(y)|$ new commands by strengthening γ 's guard with condition “ $y = z$ ” for each $z \in \text{dom}(y)$ and substituting all occurrences of y for the constant z . This transformation is mostly of theoretical interest as it may create a large number of new commands.

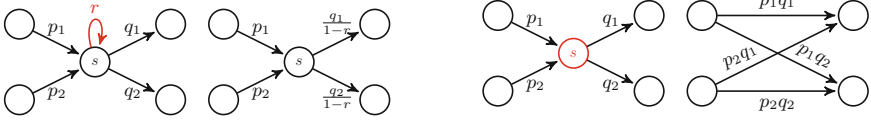


Fig. 5. State elimination in Markov chains. Left: Elimination of a self-loop. Right: Elimination of a state without self-loops. These rules preserve reachability probabilities provided that s is neither initial nor a goal state.

4.2 Elimination

For the sake of illustration, we first recall state elimination in Markov chains. Let s be a state of the Markov chain. The first step is to eliminate all self-loops of s by rescaling the probabilities accordingly (Fig. 5, left). Afterwards, all ingoing transitions are redirected to the successor states of s by multiplying the probabilities along each possible path (Fig. 5, right). The state s is then not reachable anymore and can be removed. This preserves reachability probabilities in the Markov chain provided that s was neither an initial nor goal state. Note that state elimination may increase the total number of transitions. In essence, state elimination in Markov chains is an automata-theoretic interpretation of solving a linear equation system by Gaussian elimination [29].

In the rest of this section, we develop a *location elimination rule for PCFPs* that generalizes state elimination in Markov chains. Updates and guards are handled by weakest precondition reasoning which is briefly recalled below. We then introduce a rule to remove single transitions, and show how it can be employed to eliminate *self-loop-free* locations. For the (much) more difficult case of self-loop elimination, we refer to the full version [41] for the treatment of some special cases. Handling general loops requires finding loop invariants which is notoriously difficult to automatize. Instead, the overall idea of this paper is to *create self-loop-free locations by suitable unfolding*.

Weakest Preconditions. As mentioned above, our elimination rules rely on classical weakest preconditions which are defined as follows. Fix a set Var of program variables with domains dom . Further, let u be an update and φ, ψ be predicates over Var . We call $\{\psi\} u \{\varphi\}$ a valid *Hoare-triple* if

$$\forall \nu \in \text{dom}: \quad \nu \models \psi \quad \implies \quad u(\nu) \models \varphi.$$

The predicate $\text{wp}(u, \varphi)$ is defined as the weakest ψ such that $\{\psi\} u \{\varphi\}$ is a valid Hoare-triple and is called the *weakest precondition* of u with respect to postcondition φ . Here, “weakest” is to be understood as *maximal* in the semantic implication order on predicates. Note that $u(\nu) \models \varphi$ iff $\nu \models \text{wp}(u, \varphi)$. It is well known [16] that for an update $u = \{x'_1 = f_1, \dots, x'_n = f_n\}$, the weakest precondition is given by

$$\text{wp}(u, \varphi) \quad = \quad \varphi[x_1, \dots, x_n \mapsto f_1, \dots, f_n],$$

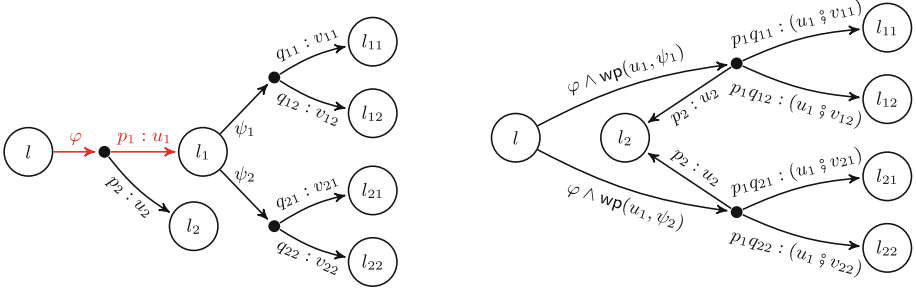


Fig. 6. Transition elimination in PCFPs. Transition $l \xrightarrow{\varphi \rightarrow p_1:u_1} l_1$ is eliminated. The rule is correct even if the depicted locations are not pairwise distinct.

i.e., all free occurrences of the variables x_1, \dots, x_n in φ are *simultaneously* replaced by the expressions f_1, \dots, f_n . For example,

$$\text{wp}(\{x' = y^2, y' = 5\}, x \geq y) = y^2 \geq 5.$$

For chained updates $u_1 \mathbin{\text{\$}} u_2$, we have $\text{wp}(u_1 \mathbin{\text{\$}} u_2, \varphi) = \text{wp}(u_1, \text{wp}(u_2, \varphi))$ [16].

Transition Elimination. To simplify the presentation, we focus on the case of *binary* PCFPs where locations have exactly two commands and commands have exactly two transitions (the general case is treated in [41]). The following construction is depicted in Fig. 6. Let $l \xrightarrow{\varphi \rightarrow p_1:u_1} l_1$ be the transition we want to eliminate and suppose that it is part of a command

$$\gamma: \quad l, \varphi \rightarrow p_1:u_1:l_1 + p_2:u_2:l_2. \quad (1)$$

Suppose that the PCFP is in a configuration $\langle l, \nu \rangle$ where guard φ is enabled, i.e., $\nu \models \varphi$. Intuitively, to remove the desired transition, we must jump with probability p_1 directly from l to one of the possible destinations of l_1 , i.e., either l_{11}, l_{12}, l_{21} or l_{22} . Moreover, we need to anticipate the—possibly non-deterministic—choice at l_1 already at l . Note that guard ψ_1 will be enabled at l_1 iff $u_1(\nu) \models \psi_1$. The latter is true iff $\nu \models \text{wp}(u_1, \psi_1)$. Hence, if $\nu \models \varphi \wedge \text{wp}(u_1, \psi_1)$, then we can choose to jump from l directly to l_{11} or l_{12} with probability p_1 . The exact probabilities p_1q_{11} and p_1q_{12} , respectively, are obtained by simply multiplying the probabilities along each path. To preserve the semantics, we must also execute the updates found on these paths in the right order, i.e., either $u_1 \mathbin{\text{\$}} v_{11}$ or $u_1 \mathbin{\text{\$}} v_{12}$. The situation is completely analogous for the other command with guard ψ_2 .

In summary, we apply the following transformation: We remove the command γ in (1) completely (and hence not only the transition $l \xrightarrow{\varphi \rightarrow p_1:u_1} l_1$) and replace it by *two new commands* γ_1 and γ_2 which are defined as follows:

$$\gamma_i: \quad l, \varphi \wedge \text{wp}(u_1, \psi_i) \rightarrow p_2:u_2:l_2 + \sum_{j=1}^2 p_1q_{ij}:(u_1 \mathbin{\text{\$}} v_{ij}):l_{ij}, \quad i \in \{1, 2\}.$$

Note that in particular, this operation preserves deterministic PCFPs: If ψ_1 and ψ_2 are mutually exclusive, then so are $\text{wp}(u_1, \psi_1)$ and $\text{wp}(u_1, \psi_2)$. If the guards are not exclusive, then the construction transfers the non-deterministic choice from l_1 to l .

Example 6. In the PCFP in Fig. 4 (middle), we eliminate the transition

$$!f \xrightarrow{0 < x < N \rightarrow 1/2:\text{nop}} f.$$

The above transition is contained in the command

$$!f, 0 < x < N \rightarrow 1/2 : \text{nop} : f + 1/2 : x-- : !f.$$

The following two commands are available at location f :

$$\begin{aligned} f, x=0 \mid x \geq N &\rightarrow 1 : \text{nop} : !f \\ f, 0 < x < N &\rightarrow 1/2 : x+=2 : !f + 1/2 : x-- : !f. \end{aligned}$$

Note that $\text{wp}(\text{nop}, \psi) = \psi$ for any guard ψ . According to the construction in Fig. 6, we add the following two new commands to location $!f$:

$$\begin{aligned} !f, 0 < x < N \ \&\ (x=0 \mid x \geq N) &\rightarrow 1/2 : \text{nop} : !f + 1/2 : x-- : !f \\ !f, 0 < x < N \ \&\ 0 < x < N &\rightarrow 1/2 : x-- : !f + 1/4 : x-- : !f \\ &+ 1/4 : x=x+2 : !f. \end{aligned}$$

The guard of the first command is unsatisfiable so that the whole command can be discarded. The second command can be further simplified to

$$!f, 0 < x < N \rightarrow 3/4 : x-- : !f + 1/4 : x=x+2 : !f.$$

Removing unreachable locations yields the PCFP in Fig. 4 (right). \triangle

Regarding the correctness of transition elimination, the intuitive idea is that the rule preserves reachability probabilities if location l_1 is *not* a potential goal. Recall that potential goals are locations for which we do not know whether they contain goal states when fully unfolded. Formally, we have the following:

Lemma 2. *Let $l_1 \in \text{Loc} \setminus \{l_i\}$ be no potential goal with respect to goal predicate ϑ and let \mathfrak{P}' be obtained from \mathfrak{P} by eliminating transition $l \xrightarrow{\varphi \rightarrow p_1 : u_1} l_1$ according to Fig. 6. Then \mathfrak{P} and \mathfrak{P}' are ϑ -reachability equivalent.*

Proof (Sketch). This follows by extending Markov chain transition elimination to MDPs and noticing that the semantic MDP $\mathcal{M}_{\mathfrak{P}'}$ is obtained from $\mathcal{M}_{\mathfrak{P}}$ by applying transition elimination repeatedly, see [41] for the details. \square

Location Elimination. We say that location $l \in \text{Loc}$ has a *self-loop* if there exists a transition $l \xrightarrow{\varphi \rightarrow p:u} l$. In analogy to state elimination in Markov chains, we can directly remove any location *without self-loops* by applying the elimination rule to its ingoing transitions. However, the case $l_1 = l_2$ in Fig. 6 needs to be examined carefully as eliminating $l \xrightarrow{\varphi \rightarrow p_1:u_1} l_1$ actually *creates two new* ingoing transitions to $l_1 = l_2$. Termination of the algorithm is thus not immediately obvious. Nonetheless, even for general (non-binary) PCFPs, the following holds:

Theorem 1 (Correctness of Location Elimination). *If $l \in \text{Loc} \setminus \{l_i\}$ has no self-loops and is not a potential goal w.r.t. goal predicate ϑ , then the algorithm*

$$\text{while } (\exists l' \xrightarrow{\varphi \rightarrow p:u} l \text{ in } \mathfrak{P}) \quad \{ \text{eliminate } l' \xrightarrow{\varphi \rightarrow p:u} l \}$$

terminates with a ϑ -reachability equivalent PCFP \mathfrak{P}' where l is unreachable.

The following notion is helpful for proving termination of the above algorithm:

Definition 6 (Transition Multiplicity). *Given a transition $l' \xrightarrow{\varphi \rightarrow p:u} l$ contained in command γ , we define its multiplicity m as the total number of transitions in γ that also have destination l .*

For instance, if $l_1 = l_2$ in Fig. 6, then transition $l \xrightarrow{\varphi \rightarrow p_1:u_1} l_1$ has multiplicity $m = 2$. If $l_1 \neq l_2$, then it has multiplicity $m = 1$.

Proof (of Theorem 1). With Lemma 2 it only remains to show termination. We directly prove the general case where \mathfrak{P} is non-binary. Suppose that l has k commands. Eliminating a transition entering l with multiplicity 1 does not create any new ingoing transitions (as l has no self-loops). On the other hand, eliminating a transition with multiplicity $m > 1$ creates k new commands, each with $m - 1$ ingoing transitions to l_1 . Thus, as the multiplicity strictly decreases, the algorithm terminates. \square

We now analyze the complexity of the algorithm in Theorem 1 in detail.

Theorem 2 (Complexity of Location Elimination). *Let $l \in \text{Loc} \setminus \{l_i\}$ be a location without self-loops. Let k be the number of commands available at l . Further, let n be the number of distinct commands in Cmd that have a transition with destination l , and suppose that each such transition has multiplicity at most m . Then the location elimination algorithm in Theorem 1 applied to l has the following properties:*

- *It terminates after at most $n(k^m - 1)/(k - 1)$ iterations.*
- *It creates at most $\mathcal{O}(nk^m)$ new commands.*
- *There exist PCFPs where it creates at least $\Omega(n2^m)$ new distinct commands with satisfiable guards.*

Proof (Sketch). We only consider the case $n = 1$ here, the remaining details are treated in [41]. We show the three items independently:

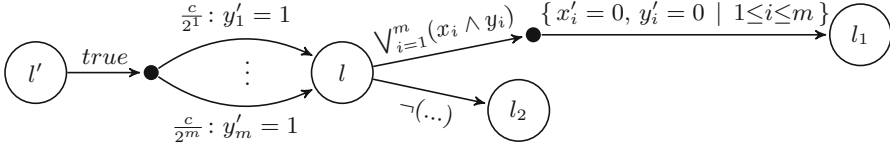


Fig. 7. The PCFP \mathfrak{P} used for the lower bound in Theorem 2. The transitions from l' to l have multiplicity m each. Variables x, y have Boolean domain, c is a normalizing constant.

- The number $I(m)$ of iterations of the algorithm in Theorem 1 applied to location l satisfies the recurrence $I(1) = 1$ and $I(m) = 1 + kI(m - 1)$ for all $m > 1$ since eliminating a transition with multiplicity $m > 1$ yields k new commands with multiplicity $m - 1$ each. The solution of this recurrence is $I(m) = \sum_{i=0}^{m-1} k^i = (k^m - 1)/(k - 1)$ as claimed.
- For the upper bound on the number of new commands, we consider the execution of the algorithm in the following stages: In stage 1, there is a single command with multiplicity m . In stage j for $j > 1$, the commands from the previous stage are transformed into k new commands with multiplicity $m - j + 1$ each. In the final stage m , there are thus k^{m-1} commands with multiplicity 1 each. Eliminating all of them yields $k \cdot k^{m-1} = k^m$ new commands after which the algorithm terminates.
- Consider the PCFP \mathfrak{P} in Fig. 7 where $k = 2$. Intuitively, location elimination must yield a PCFP \mathfrak{P}' with 2^m commands available at location l' because every possible combination of the updates $y'_i = 1, i = 1, \dots, m$, may result in enabling either of the two guards at l . Indeed, for each such combination, the guard which is enabled depends on the values of x_1, \dots, x_m at location l' . Thus in the semantic MDP $\mathcal{M}_{\mathfrak{P}'}$, for every variable valuation ν with $\nu(y_i) = 0$ for all $i = 1, \dots, m$, the probabilities $P(\langle l', \nu \rangle, \langle l_1, \mathbf{0} \rangle)$ are *pairwise distinct*. This implies that \mathfrak{P}' must have 2^m commands (with satisfiable guards) at l' . □

5 Implementation

Overview. We have implemented our approach in the probabilistic model checker *storm* [26]. Technically, instead of defining custom data structures for our PCFPs, we operate directly on models in the *jani* model exchange format [11]. *storm* accepts *jani* models as input and also supports conversion from PRISM to *jani*. The PCFPs described in this paper are a subset of the models expressible in *jani*. Other *jani* models such as timed or hybrid automata are not in the scope of our implementation. In practice, we use our algorithms as a *simplification front-end*, i.e., we apply just a handful of unfolding and elimination steps and then fall back to *storm*'s default engine. This is steered by heuristics that we explain in detail further below.

Features. Apart from the basic PCFPs treated in the previous sections, our implementation supports the following more advanced *jani* features:

- *Parameters.* It is common practice to leave key quantities in a high-level model undefined and then analyze it for various instantiations of those parameters (as done in most of the PRISM case studies³); or synthesize in some sense suitable parameters [14, 29, 37]. Examples include undefined probabilities or undefined variable bounds like N in the PRISM program in Fig. 2. Our approach can naturally handle such parameters and is therefore particularly useful in situations where the model is to be analyzed for several parameter configurations. Virtually, the only restriction is that we cannot unfold variables with parametric bounds.
- *Rewards.* Our framework can be easily extended to accommodate expected-reward-until-reachability properties (see e.g. [5, Def. 10.71] for a formal definition). The latter are also highly common in the benchmarks used in the quantitative verification literature [25]. Formally, in a *reward PCFP*, each transition is additionally equipped with a non-negative reward that can either be a constant or given as an expression in the program variables. Technically, the treatment of rewards is straightforward: Each time we multiply the probabilities of two transitions in our transition elimination rule (Fig. 6), we *add* their corresponding rewards.
- *Parallel composition.* PCFPs can be extended by action labels to allow for synchronization of various parallel PCFPs. This is standard in model checking (e.g. [5, Sec. 2.2.2]). We have implemented two approaches for dealing with this: (1) A “flat” product model is constructed first. This functionality is already shipped with the storm checker. This approach is restricted to compositions of just a few modules as the size of the resulting product PCFP is in general exponential in the number of modules. Nonetheless, in many practical cases, flattening leads to satisfactory results (cf. Sect. 6). (2) Control-flow elimination is applied to each component individually. Here, we may only eliminate *internal*, i.e. non-synchronizing commands, and we forbid shared variables. Otherwise, we would alter the resulting composition.
- *Probability expressions.* Without changes, all of the theory presented so far can be applied to PCFPs with probability expressions like $|x|/(|x| + 1)$ over the program variables instead of constant probabilities only. Expressions that do not yield correct probabilities are considered modeling errors.

Heuristics. The choice of the next variable to be unfolded and the next location to be eliminated is driven by heuristics. The overall goal of the heuristics is to eliminate as many locations as possible while maintaining a reasonably sized PCFP. This is controlled by two configurable parameters, L and T . The heuristics alternates between unfolding and elimination (see the diagram in Fig. 8).

To find a suitable variable for unfolding, the heuristics first analyzes the dependency graph defined in Sect. 4.1. It then selects a variable based on the

³ <https://www.prismmodelchecker.org/casestudies/>.

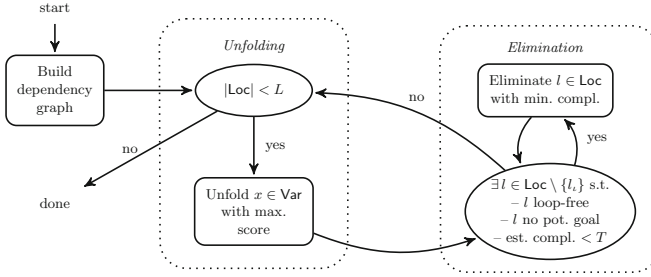


Fig. 8. Our heuristics alternates between unfolding and elimination steps. The next unfold is determined by selecting a variable with maximal *score* as computed by a static analysis (see main text). Loop-free non-potential goal locations are then eliminated until the next elimination has a too high estimated complexity.

following static analysis: For each unfoldable variable x , the heuristics considers each command γ in the PCFP and determines the percentage $p(\gamma, x)$ of γ 's transitions that have an update with writing access to x . Each variable is then assigned a *score* which is defined as the average percentage $p(\gamma, x)$ over all commands of the PCFP. The intuition behind this technique is that variables which are changed in many commands are more likely to create self-loop free locations when unfolded. We consider the percentage for each command individually in order to not give too much weight to commands with many transitions. Unfolding is only performed if the current PCFP has at most L locations. By default, $L = 10$ which in practice often leads to unfolding just two or three variables with small domains.

After unfolding a variable, the heuristics tries to eliminate self-loop-free locations that are no potential goals. The next location to be eliminated is selected by estimating the number of new commands that would be created by the algorithm. Here, we rely on the theoretical results from Theorem 2: In particular, we take the *multiplicity* (cf. Definition 6) of ingoing transitions into account which may cause an exponential blowup. We use the estimate $\mathcal{O}(nk^m)$ from Theorem 2 as an approximation for the elimination complexity; determining the *exact* complexity of each possible elimination is highly impractical. We only eliminate locations whose estimated complexity is at most T , and we eliminate those with lowest complexity first. By default, $T = 10^4$.

6 Experiments

In this section, we report on our experimental evaluation of the implementation described in the previous section.

Benchmarks. We have compiled a set of 10 control-flow intensive DTMC and MDP benchmarks from the literature. Each benchmark model is equipped with a reachability or expected reward property.

Table 1. Reductions achieved by our control-flow elimination. Times are in ms.

Name	Type	Prop. type	Red. time	Params.	States		Transitions		Build time		Check time		Total time	
					orig.	red.	orig.	red.	orig.	red.	orig.	red.	orig.	red.
BRP	dtmc	P	134	2 ¹⁰ /5	78.9K	-44%	106K	-33%	261	-33%	22	-38%	16,418	-46%
				2 ¹¹ /10	291K	-45%	397K	-33%	1,027	-39%	101	-46%		
				2 ¹² /20	1.11M	-46%	1.53M	-33%	3,945	-48%	462	-48%		
				2 ¹³ /25	2.76M	-46%	3.8M	-33%	9,413	-47%	1,187	-47%		
COINGAME	dtmc	P	35	10 ⁴	20K	-50%	40K	-50%	53	-24%	18,500	-79%	18,553	-78%
DICE5	mdp	P	671	n/a	371K	-84%	2.01M	-83%	1,709	-82%	9,538	-99%	11,247	-91%
EAJS	mdp	R	223	10 ³	194K	-28%	326K	-1%	1,242	-43%	220	-32%	18,397	-42%
				10 ⁴	2M	-28%	3.38M	-1%	13,154	-46%	3,780	-31%		
GRID	dtmc	P	117	10 ⁴	300K	-47%	410K	-34%	1,062	-57%	17	-52%	11,716	-52%
				10 ⁵	3M	-47%	4.1M	-34%	10,430	-53%	207	-54%		
HOSPITAL	mdp	P	57	n/a	160K	-66%	396K	-27%	502	-50%	19	-56%	521	-39%
NAND	dtmc	P	80	20/4	308K	-79%	476K	-52%	589	-45%	108	-75%	86,060	-56%
				40/4	4M	-80%	6.29M	-51%	8,248	-50%	1,859	-77%		
				60/2	9.42M	-80%	14.9M	-50%	19,701	-49%	4,685	-76%		
				60/4	18.8M	-80%	29.8M	-50%	40,168	-53%	10,703	-77%		
ND-NAND	mdp	P	106	20/4	308K	-79%	476K	-52%	618	-36%	127	-74%	96,956	-52%
				40/4	4M	-80%	6.29M	-51%	8,783	-42%	2,270	-77%		
				60/2	9.42M	-80%	14.9M	-50%	21,792	-47%	5,646	-75%		
				60/4	18.8M	-80%	29.8M	-50%	44,409	-46%	13,312	-76%		
NEGOTIATION	dtmc	P	148	10 ⁴	129K	-32%	184K	-26%	481	-39%	22	-49%	5,631	-39%
				10 ⁵	1.29M	-32%	1.84M	-26%	4,930	-43%	197	-30%		
POLE	dtmc	R	208	10 ²	315K	-46%	790K	-4%	1,496	-46%	26	-42%	17,431	-45%
				10 ³	3.16M	-46%	7.9M	-4%	15,503	-47%	406	-33%		

BRP models a bounded retransmission protocol and is taken from the PRISM benchmark suite. COINGAME is our running example from Fig. 2. DICE5 is an example shipped with *storm* and models rolling several dice, five in this case, that are themselves simulated by coinflips in parallel. EAJS models energy-aware job scheduling and was first presented in [3]. GRID is taken from [2] and represents a robot moving in a partially observable grid world. HOSPITAL is adapted from [8] and models a hospital inventory management problem. NAND is the von Neumann NAND multiplexing system mentioned near the end of Sect. 2. ND-NAND is a custom-made adaption of NAND where some probabilistic behavior has been replaced by non-determinism. NEGOTIATION is an adaption of the Alternating Offers Protocol from [6] which is also included in the PRISM case studies. POLE is also from [2] and models balancing a pole in a noisy and unknown environment. The problems BRP, EAJS, and NAND are part of the QComp benchmark set [25].

For all examples except DICE5, we have first flattened parallel compositions (if there were any) into a single module, cf. Sect. 5.

Setup. We report on two experiments. In the first one, we compare the number of states and transitions as well as the model build and check times of the original and the reduced program (columns ‘States’, ‘Transitions’, ‘Build time’, and ‘Check time’ of Table 1). We work with *storm*’s default settings⁴. We also report the time needed for the reduction itself, including the time consumed by flatten-

⁴ By default, *storm* builds the Markov model as a sparse graph data structure and uses (inexact) floating point arithmetic.

ing (column ‘Red. time’). We always use the default configuration for our heuristics, i.e., *we do not manually fine-tune* the heuristics for each benchmark. We report on some additional experimental results obtained with fine-tuned heuristics in [41]. For the benchmarks where this is applicable, we consider the different parameter configurations given in column ‘Params.’. Recall that in these cases, we need to compute the reduced program only once. We report the amortized runtime of *storm* on all parameter configurations vs. the runtime on the reduced models, including the time needed for reduction in the rightmost column ‘Total time’. In the second, less extensive experiment, we compare our reductions to bisimulation minimization (Table 2 below). All experiments were conducted on a notebook with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 GB of RAM. The script for creating the table is available⁵.

Results. Our default heuristics was able to reduce all considered models in terms of states (by 28–84%) and transitions (by 1–83%). The total time for building and checking these models was decreased by 39–91%. The relative decrease in the number of states is usually more striking than the decrease in the number of transitions. This is because, as explained in Sect. 4, location elimination always removes states but may add more commands to the PCFP and hence more transitions to the underlying Markov model. Similarly, the time savings for model checking are often higher than the ones for model building; here, this is mostly because building our reduced model introduces some overhead due to the additional commands. The reduction itself was always completed within a fraction of a second and is independent of the size of the underlying state space.

Bisimulation and Control-Flow Reduction. In Table 2, we compare the compression achieved by *storm*’s probabilistic bisimulation engine, our method and *both* techniques combined. We also include the total time needed for reduction, model building and checking. For the comparison, we have selected three benchmarks representing three different situations: (1) for BRP, the two techniques achieve similar reductions, (2) for NAND, our reduced model is smaller than the bisimulation quotient, and (3) for POLE, the situation is the other way around, i.e., the bisimulation quotient is (much) smaller than our reduced model. Interestingly, combining the two techniques yields an even smaller model in all three cases. This demonstrates the fact that *control-flow reduction and bisimulation are orthogonal* to each other. In the examples, control-flow reduction was also faster than bisimulation as the latter has to process large explicit state spaces. It is thus an interesting direction for future work to combine program-level reduction techniques that yield bisimilar models with control-flow reduction.

When Does Control-Flow Reduction Work Well? Our technique works best for models that use one or more explicit or implicit program counters. Such program counters often come in form of a variable that determines which commands are currently available and that is updated after most execution steps. Unfolding

⁵ <https://doi.org/10.5281/zenodo.5497947>.

Table 2. Comparison of bisimulation minimization and our control-flow reduction (‘CFR’). Column ‘Total time’ includes building, reducing and checking the model.

Name	Params.	States			Transitions			Total time		
		Bisim.	CFR	Both	Bisim.	CFR	Both	Bisim.	CFR	Both
BRP	$2^{12}/20$	598K	606K	344K	852K	1.02M	598K	4,767	2,883	2,965
NAND	40/4	3.21M	816K	678K	5M	3.1M	2.46M	17,868	5,588	8,199
POLE	10^3	4.06K	1.72M	1.2K	12.2K	7.54M	9.82K	19,443	10,305	10,801

such variables typically yields several loop-free locations. For example, the variable \mathbf{f} in Fig. 2 is of this kind. However, we again stress that there is no formal difference between program counter variables and “data variables” in our framework. The distinction is made automatically by our heuristics; no additional user input is required. Control-flow reduction yields especially good results if it can be applied compositionally such as in the DICE5 benchmark.

Limitations. Finally, we remark that our approach is less applicable to extensively synchronizing parallel compositions of more than just a handful of modules. The flattening approach then typically yields large PCFPs which are not well suited for symbolic techniques such as ours. Larger PCFPs also require a significantly higher model building time. Another limiting factor are dense variable dependencies in the sense of Sect. 4.1, i.e., the variable dependency graph has relatively large BSCCs. The latter, however, seems to rarely occur in practice.

7 Conclusion

This paper presented a property-directed “unfold and eliminate” technique on probabilistic control-flow programs which is applicable to state-based high-level modeling languages. It preserves reachability probabilities and expected rewards exactly and can be used as a simplification front-end for any probabilistic model checker. It can also handle parametric DTMC and MDP models where some key quantities are left open. On existing benchmarks, our implementation achieved model compressions of up to an order of magnitude, even on models that have much larger bisimulation quotients. Future work is to amend this approach to continuous-time models like CMTCs and Markov automata, and to further properties such as LTL.

References

1. de Alfaro, L., Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 395–410. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_27

2. Andriushchenko, R., Češka, M., Junges, S., Katoen, J.-P., Stupinský, Š: PAYNT: a tool for inductive synthesis of probabilistic programs. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 856–869. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_40
3. Baier, C., Daum, M., Dubslaff, C., Klein, J., Klüppelholz, S.: Energy-utility quantiles. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 285–299. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_24
4. Baier, C., Größer, M., Ciesinski, F.: Partial order reduction for probabilistic systems. In: QEST 2004, pp. 230–239 (2004). <https://doi.org/10.1109/QEST.2004.1348037>
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Ballarini, P., Fisher, M., Wooldridge, M.J.: Automated game analysis via probabilistic model checking: a case study. Electron. Notes Theor. Comput. Sci. **149**(2), 125–137 (2006). <https://doi.org/10.1016/j.entcs.2005.07.030>
7. Batz, K., Junges, S., Kaminski, B.L., Katoen, J.-P., Matheja, C., Schröer, P.: PrIC3: property directed reachability for MDPs. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 512–538. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_27
8. Biagi, M., Carnevali, L., Santoni, F., Vicario, E.: Hospital inventory management through Markov decision processes @runtime. In: McIver, A., Horvath, A. (eds.) QEST 2018. LNCS, vol. 11024, pp. 87–103. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99154-2_6
9. Bohnenkamp, H.C., D’Argenio, P.R., Hermanns, H., Katoen, J.: MODEST: a compositional modeling formalism for hard and softly timed systems. IEEE Trans. Softw. Eng. **32**(10), 812–830 (2006). <https://doi.org/10.1109/TSE.2006.104>
10. Buchholz, P., Katoen, J., Kemper, P., Tepper, C.: Model-checking large structured Markov chains. J. Log. Algebraic Methods Program. **56**(1–2), 69–97 (2003). [https://doi.org/10.1016/S1567-8326\(02\)00067-X](https://doi.org/10.1016/S1567-8326(02)00067-X)
11. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
12. D’Argenio, P.R., Niebert, P.: Partial order reduction on concurrent probabilistic programs. In: QEST 2004, pp. 240–249 (2004). <https://doi.org/10.1109/QEST.2004.1348038>
13. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_21
14. Dehnert, C., et al.: PROPhESY: a PRObabilistic ParamETER SYnthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13
15. Dehnert, C., Katoen, J.-P., Parker, D.: SMT-based bisimulation minimisation of Markov models. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 28–47. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_5
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood (1976)
17. Donaldson, A.F., Miller, A., Parker, D.: Language-level symmetry reduction for probabilistic model checking. In: Proceedings of the QEST 2009, pp. 289–298 (2009). <https://doi.org/10.1109/QEST.2009.21>

18. Dong, Y., Ramakrishnan, C.R.: An optimizing compiler for efficient model checking. In: FORTE XII/PSTV XIX. IFIP Conference Proceedings, vol. 156, pp. 241–256. Kluwer (1999)
19. Dubslaff, C., Morozov, A., Baier, C., Janschek, K.: Reduction methods on probabilistic control-flow programs for reliability analysis. In: 30th European Safety and Reliability Conference, ESREL (2020). <https://www.rpsonline.com.sg/proceedings/esrel2020/pdf/4489.pdf>
20. Esparza, J., Hoffmann, P., Saha, R.: Polynomial analysis algorithms for free choice probabilistic workflow nets. *Perform. Eval.* **117**, 104–129 (2017). <https://doi.org/10.1016/j.peva.2017.09.006>
21. Fatmi, S.Z., Chen, X., Dhamija, Y., Wildes, M., Tang, Q., van Breugel, F.: Probabilistic model checking of randomized Java code. In: Laarman, A., Sokolova, A. (eds.) SPIN 2021. LNCS, vol. 12864, pp. 157–174. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84629-9_9
22. Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.* **20**, 61–124 (2003). <https://doi.org/10.1613/jair.1129>
23. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: PASS: abstraction refinement for infinite probabilistic models. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 353–357. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_30
24. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
25. Hartmanns, A., Klauck, M., Parker, D., Quatmann, T., Ruijters, E.: The quantitative verification benchmark set. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 344–350. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_20
26. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker STORM. *Int. J. Softw. Tools Technol. Transfer* 1–22 (2021). <https://doi.org/10.1007/s10009-021-00633-z>
27. Jansen, D.N., Groote, J.F., Timmers, F., Yang, P.: A near-linear-time algorithm for weak bisimilarity on Markov chains. In: CONCUR 2020. LIPIcs, vol. 171, pp. 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). <https://doi.org/10.4230/LIPIcs.CONCUR.2020.8>
28. Jeannot, B.: Dynamic partitioning in linear relation analysis: application to the verification of reactive systems. *Formal Methods Syst. Des.* **23**(1), 5–37 (2003). <https://doi.org/10.1023/A:1024480913162>
29. Junges, S., et al.: Parameter synthesis for Markov models. CoRR abs/1903.07993 (2019). <http://arxiv.org/abs/1903.07993>
30. Katoen, J.-P., Kemna, T., Zapreev, I., Jansen, D.N.: Bisimulation minimisation mostly speeds up probabilistic model checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 87–101. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_9
31. Kattenbelt, M., Kwiatkowska, M.Z., Norman, G., Parker, D.: A game-based abstraction-refinement framework for Markov decision processes. *Formal Methods Syst. Des.* **36**(3), 246–280 (2010). <https://doi.org/10.1007/s10703-010-0097-6>
32. Kurshan, R., Levin, V., Yenigün, H.: Compressing transitions for model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 569–582. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_48

33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
34. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science, Springer, New York (2005). <https://doi.org/10.1007/b138392>
35. Norman, G., Parker, D., Kwiatkowska, M.Z., Shukla, S.K.: Evaluating the reliability of NAND multiplexing with PRISM. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **24**(10), 1629–1637 (2005). <https://doi.org/10.1109/TCAD.2005.852033>
36. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Series in Probability and Statistics, Wiley, Hoboken (1994). <https://doi.org/10.1002/9780470316887>
37. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.-P.: Parameter synthesis for Markov models: faster than ever. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 50–67. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_4
38. Valmari, A., Franceschinis, G.: Simple $O(m \log n)$ time Markov chain lumping. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 38–52. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_4
39. Wachter, B., Zhang, L.: Best probabilistic transformers. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 362–379. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11319-2_26
40. Wimmer, R., Braitling, B., Becker, B.: Counterexample generation for discrete-time Markov chains using bounded model checking. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 366–380. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-93900-9_29
41. Winkler, T., Lehmann, J., Katoen, J.: Out of control: reducing probabilistic models by control-state elimination. CoRR abs/2011.00983 (2020). <https://arxiv.org/abs/2011.00983>
42. Younes, H.L., Littman, M.L.: PPDDL1.0: an extension to PDDL for expressing planning domains with probabilistic effects. Technical report, CMU-CS-04-162, **2**, 99 (2004)