

Rheinische-Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik 2
Software Modeling and Verification
Prof. Dr. Ir. Joost-Pieter Katoen

Bachelor Thesis

On-the-fly Model Checking of Probabilistic Programs

Lukas Westhofen

December 15, 2015

First Reviewer:	Prof. Dr. Ir. Joost-Pieter Katoen
Second Reviewer:	apl. Prof. Dr. Thomas Noll
Advisor:	Dr. Nils Jansen

Ich versichere, die vorliegende Arbeit selbstständig verfasst zu haben und keine anderen als die angegebenen Hilfsmittel benutzt zu haben. Sowohl sinngemäße als auch wörtliche Zitate sind als solche kenntlich gemacht.

Aachen, den 10. August 2015.

Abstract

In this paper, we introduce an on-the-fly model checking approach for probabilistic programs represented by the probabilistic guarded command language. While iteratively building the possibly infinite model of a program, we run a model checker after a certain amount of construction steps and check whether the already constructed model parts satisfy the given property. We examine various heuristics allowing us to prioritize paths with greater chances of a satisfaction. Moreover, we prove correctness of the presented approach. We show that for a particular class of properties results can be obtained in finitely many steps, hence achieve a partial completeness result. We present experimental results and analyze the practicability of the heuristics and the general approach.

Contents

1	Introduction	1
2	Preliminaries	4
2.1	Basic notations	4
2.2	Probability theory	4
2.3	Probabilistic models	5
2.4	Probabilistic computation tree logic	10
2.5	Counterexamples	14
3	The probabilistic guarded command language	15
3.1	Syntax of PGCL	15
3.2	Semantics of PGCL	18
4	Iterative on-the-fly PGCL model checking	23
4.1	PGCL models and their properties	23
4.2	Transferability of properties and counterexamples	24
4.3	Handling the observe statement	28
4.4	Iterative PGCL model building and checking	29
4.5	Soundness and completeness	32
5	Heuristics	34
5.1	Local heuristics	36
5.2	Global heuristics	39
6	Implementation and testing	43
6.1	StoRM framework	43
6.2	Performance and validity evaluation	44
6.2.1	N-Cowboys	44
6.2.2	Lotka-Volterra	47
6.2.3	Herman	49
6.2.4	Robot	50
6.2.5	State explosion	51
7	Conclusion and future work	54
8	Appendix	57

1 Introduction

Ever since the accident of the Ariane 5 rocket in 1996, engineers and executives are aware that a correct software is a crucial property. In case of Ariane 5, 290 million Euro were claimed for loss due to a erroneous arithmetic overflow during a floating point conversion. This bug shows that testing software and hardware is a necessary procedure, but often not sufficient. The desire for an automatic way to check systems arises. In this paper, we elaborate on checking *probabilistic programs* in an automatic fashion.

Probabilistic programming approaches, where execution choices of a program are solved in a probabilistic manner, are used in a wide variety of fields utilizing computer programming. This includes machine learning, network and communication protocols, and scientific simulations. Just to name a few examples, the FireWire-protocol, the CSMA/CD-protocol, and TrueSkill online gaming rating system all heavily rely on probabilistic choices [1, 2, 3].

A probabilistic program embodies – on an abstract level – nothing more than a probability distribution over all possible outcomes. To be more specific, we can represent such a program in a familiar textual fashion such that assignments, control structures and probabilistic decision can be made.

In this paper, probabilistic programs are specified using the *probabilistic guarded command language* (PGCL), which resembles the C programming language but allows for probabilistic decisions via the operator $\{\dots\}[p]\{\dots\}$, declaring that the inner part of the left bracket pair is executed with probability p , whereas the right part is executed with probability $1-p$. The language itself is an extension of Dijkstra’s Guarded Command Language [4]. Let us examine PGCL programs and their properties using the example program depicted in Figure 1.

```
1 double a := 0.5;
2 int success := 0;
3 while(success = 0) {
4     // tries to send message
5     {success := 1;} [a] {success := 0;}
6     // simulates transmission fluctuation
7     {a := a + 0.01;} [0.5] {a := a - 0.01;}
8     if(a < 0 | a > 1) {
9         a := 0.5;
10    }
11 }
```

Figure 1: A simple PGCL program simulating a message delivery system with a fluctuating network quality.

This probabilistic program simulates a message sending process on a network with a fluctuating transmission quality. The probability of successfully sending a message is represented by the variable `a`. We set `success` to 1 if the transmission was successful. The program will try to send the message as long as it is not delivered. The fluctuation of

the network quality is simulated by randomly increasing or decreasing the transmission success probability. If the value reaches 0 or 1 it is reset to its initial value 0.5, since probabilities greater than 1 respectively less than 0 are not allowed.

Hence, the network fluctuation is represented by a one-dimensional random walk in the interval $[0, 1]$ with discrete steps of 0.01. As the probability of increasing and decreasing the value is both 0.5, the random walk is guaranteed to attain every value of $[0, 1]$ with probability one. Intuitively, this leads to the chance of successfully sending the message being one in every possible program execution – The only varying part is the number of steps needed for a successful transmission.

As mentioned above, probabilistic programs are frequently used in practice, often in safety-critical environments such as in automotive technology. Hence, the desire for checking the correctness of such programs arises. A *model checker* serves this purpose: For a given model and property, it checks whether that model satisfies the given property. Model checkers for finite probabilistic models are implemented in a vast amount, including the popular model checker PRISM [5], which operates on discrete-time Markov chains, Markov decision processes, and other finite probabilistic models.

In this paper, we introduce a possibility to model check PGCL programs – A task slightly different from what standard model checkers do, as PGCL programs in general yield an infinite model. This is the case for every Turing-complete programming language since infinite executions (i.e. the program not coming to a halt) are possible. We note that our exemplary program of Figure 1 is infinite due to the existence of the infinite run where **success** is always set to 0. Thus, we are given a special problem for which most model checking practices do not suffice. One should note that it is in general not possible to verify all properties of PGCL programs, as the halting problem was proven to be undecidable by Alan Turing in 1936 [6]. This statement effects the content of this paper drastically: We recognize that it is not possible to create a both correct and complete PGCL model checker.

But, on the bright side, it is possible to construct a model checker that yields results for some, but not all properties. We approach the model checking process by building the model of the PGCL program in an iterative manner, i.e. simulating the program step-by-step until we find a point where we can guarantee that the program does not respectively does satisfy the given property. We will later examine in detail on what class of properties is checkable using this approach.

Just to name an example, even for the very basic PGCL program

```
int x := 1; while(x > 0) x := x + 1;
```

we can not verify the property that variable `x` will eventually have the value `-1`, as at no point of the simulation the model checker can guarantee that this assignment will not eventually happen. Using human ingenuity it is of course quite easy to see that this property is violated by the program. In general, we note a failure of the presented approach for properties requiring knowledge of the complete infinite system.

By exploring the state space of the probabilistic program step-by-step, we are offered the chance of a heuristical approach: More promising executions should be explored earlier, whereas unpromising executions are not prioritized. As an example, if the model checker is requested to verify whether a certain state is reachable with a probability greater than 0.8, it should first explore the most probable paths as this paths keep the chance of the probability being greater than 0.8 high. Later on, we evaluate on how the colloquial term “promising” can be defined for paths. As calculating the data needed for an evaluation of the paths can be quite costly, we evaluate different heuristics and compare the trade-off between a more qualitative building process and higher time consumption. We anticipate faster results using a more elaborate way of constructing the model as unnecessary parts of the model can be cut off early – On the other hand, one should keep the runtime of the heuristic in mind.

We structure this paper as follows: First off, we introduce the reader to probabilistic models and the corresponding probabilistic computation tree logic. Ensuing, we define the syntax and operational semantics of the probabilistic guarded command language. As the main part of this thesis, we present the iterative PGCL model checking algorithm and analyze the transferability of properties closely. To extend the iterative approach, we suggest the usage of various heuristics. Both heuristics and the general practicability of our approach are examined by means of experimental results.

2 Preliminaries

2.1 Basic notations

We use the following basic notations in this paper. The empty set $\{\}$ containing no elements is denoted by \emptyset . The disjoint union over two sets X and Y is denoted by $X \dot{\cup} Y$. For a set X the power set of X is defined as $2^X = \{Y \mid Y \subseteq X\}$. For a set X over a universe U , i.e. $X \subseteq U$, the complement of X is defined as $\bar{X} = U \setminus X$. To denote a finite set X we write $|X| < \infty$. The functions $\max(X)$ and $\min(X)$ return the maximal respectively minimal element of a set X for a given strict total order on the elements of X . If no maximal or minimal element exists, it returns ∞ respectively $-\infty$.

For a finite word u and a finite or infinite word w we define $u \cdot w$ to denote the concatenation of both words. For an alphabet Σ and a finite word $u \in \Sigma^*$ we write u^ω to define the infinite iteration of u , hence $u^\omega = u \cdot u \cdot u \cdot \dots$. Similarly, for a finite set of words $U = \{u_1, \dots, u_n\}$ we define the infinite iteration over U as $U^\omega = \{a \mid a: \mathbb{N} \rightarrow U\}$.

2.2 Probability theory

Since probabilistic model checking requires arguing on formal stochastic methods, a short introduction to probability theory is given in the following. For any further read on the topic of stochastics we refer to [7].

In measure theory, stochastic events and their likelihood are defined using probability spaces. Those structures include a set of elementary events, a σ -algebra representing all possible events, and a probability measure assigning a probability to each of those events.

Definition 1 (σ -algebra). *For a set of elementary events Ω a set $\Sigma \subseteq 2^\Omega$ defines a σ -algebra if the following constraints hold:*

1. $\Omega \in \Sigma$ (Inclusion of elementary events.)
2. $A \in \Sigma \implies \bar{A} \in \Sigma$ (Closure under complement.)
3. $A_1, A_2, \dots \in \Sigma \implies \bigcup_{i \geq 1} A_i \in \Sigma$ (Closure under countable union.)

Definition 2 (Probability measure). *A function $P: \Sigma \rightarrow [0, 1]$ over a σ -algebra Σ defines a probability measure if the following constraints hold:*

1. $P(\emptyset) = 0$
2. $P(A) = 1 - P(\bar{A})$ for every $A \in \Sigma$.
3. $P(\bigcup_{i \geq 1} A_i) = \sum_{i \geq 1} P(A_i)$ for every countable sequence of pairwise disjoint sets $(A_i)_{i \geq 1}$.

Definition 3 (Probability space). *A probability space is a three-tuple (Ω, Σ, P) , where Ω is the set of elementary events, Σ is a σ -algebra over Ω , and P is a probability measure for Σ .*

2.3 Probabilistic models

Probabilistic models are derived from standard discrete-time models, but allow for probabilistic transitions. For this purpose, for every state of a probabilistic model a probability distribution over its successor states is defined. In this paper we differentiate between two different probabilistic model types: *Discrete-time Markov chains* and *Markov decision processes*. Whereas the first type only allows for deterministic probabilistic transitions, the latter also incorporates nondeterminism. The hereafter presented definitions and notations can be found in more detail in the book *Principles of Model Checking* by C. Baier and J.-P. Katoen [8], therefore we advise the reader to refer to this book in case any questions arise in the following.

Discrete-time Markov chains, or in short Markov chains, represent a simple type of probabilistic models, where for every state a probability distribution over its successor states is given.

Definition 4 (Discrete-time Markov chain (DTMC)). *A discrete-time Markov chain is a six-tuple $\mathfrak{M} = (S, s_{init}, P, AP, L)$ where*

- *S is the countable set of states.*
- *$s_{init} \in S$ is the initial state.*
- *$P: S \times S \rightarrow [0, 1] \subset \mathbb{Q}$ is the transition probability function, where a rational probability is assigned to each pair of states. We constraint for every state $s \in S$ that $\sum_{s' \in S} P(s, s') \in \{0, 1\}$ holds.*
- *AP is the set of atomic propositions.*
- *$L: 2^{AP} \rightarrow S$ is the labeling function and assigns a set of atomic propositions to every node.*

DTMCs can be, per definition, infinite. Since we want to examine properties of those infinite DTMCs, we need a formalism allowing us to argue on finite parts of infinite systems. Hence, we introduce the notion of *subsystems*. A subsystem of a given DTMC contains a subset of the state space and introduces no novel transitions to the system. To fill up the sum of all outgoing probabilities of a state being one, we add a fresh, self-looping state $*$ to the system and lead every “missing” transition with the remaining possibility into this state.

Definition 5 (Subsystem of a DTMC). *Let $\mathfrak{M} = (S, s_{init}, P, AP, L)$ be a DTMC. Every DTMC $\mathfrak{M}' = (S', s_{init}, P', AP, L')$ is a subsystem of \mathfrak{M} if the following holds:*

- *$S' \subseteq S \dot{\cup} \{*\}$.*
- *If $P'(n, n') = p > 0$ for some states $n, n' \in S'$ then $P(n, n') = p$.*
- *$P(n, *) = 1 - \sum_{n' \in S'} P(n, n')$ for any state $n \in S'$.*
- *For every state $n \in S'$ holds $L'(n) = L(n)$.*

We write $\mathfrak{M}' \sqsubseteq \mathfrak{M}$ and denote \mathfrak{M} as the suprasystem of \mathfrak{M}' .

Figure 2 shows an exemplary Markov chain \mathfrak{M} and a subsystem $\mathfrak{M}' \subseteq \mathfrak{M}$. Two states are labeled: $L(s_2) = \{a\}$ and $L(s_4) = \{b\}$. Note that every state s satisfies the transition function constraint $\sum_{s' \in S} P(s, s') = 1$. The subsystem is reduced to the states s_0 , s_2 and s_3 . Since now one outgoing transition misses for each s_0 and s_3 , both states lead into the artificial state $*$ with the remaining probability to receive a valid DTMC again.

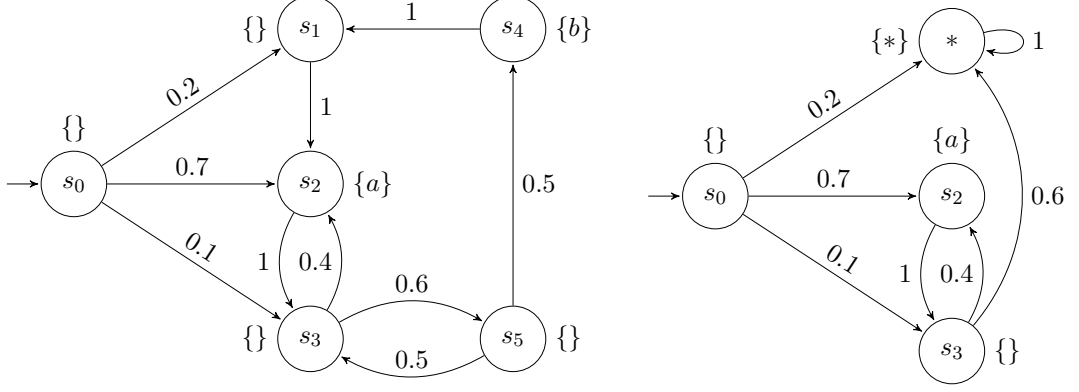


Figure 2: An exemplary DTMC \mathfrak{M} is presented on the left side of this figure. It has $AP = \{a, b\}$ and s_0 as its initial state. The right side shows a subsystem \mathfrak{M}' of \mathfrak{M} .

Markov decision processes represent a generalization of discrete-time Markov chains. As mentioned above, Markov decision processes implement nondeterminism for probabilistic models. They do so by allowing for every state a nondeterministic choice over probability distributions to its successor states.

Definition 6 (Markov decision process (MDP)). A Markov decision process is a five-tuple $\mathfrak{M} = (S, S_{init}, Act, P, AP, L)$ where

- S is the countable set of states.
- $S_{init} \subseteq S$ defines the enumerable set of initial states.
- Act is the set of actions.
- $P: S \times Act \times S \rightarrow [0, 1]$ is the transition probability function where $[0, 1] \subset \mathbb{Q}$. We constraint for every state $s \in S$ and action $a \in Act$ that $\sum_{s' \in S} P(s, a, s') \in \{0, 1\}$ holds.
- AP is the set of atomic propositions.
- $L: S \rightarrow 2^{AP}$ is the labeling function and assigns a set of atomic propositions to every node.

The set of actions represent possible nondeterministic choices, which results in the existence of multiple successor probability distributions for every state. Note that not

for every state we need to specify every possible action: If for some state no outgoing transition for some action is given, the probability of that action to be taken is 0 for that state. It should be clear that every DTMC can be represented using an MDP with only one action in Act .

Figure 3 shows a graphical representation of an MDP \mathfrak{M} with its set of initial states $S_0 = \{s_0, s_1\}$. We indicate the different actions by simply adding them as labels to the transitions. \mathfrak{M} has two different actions that can be executed: α_1 and α_2 . State s_3 illustrates the special case that only one action can be taken.

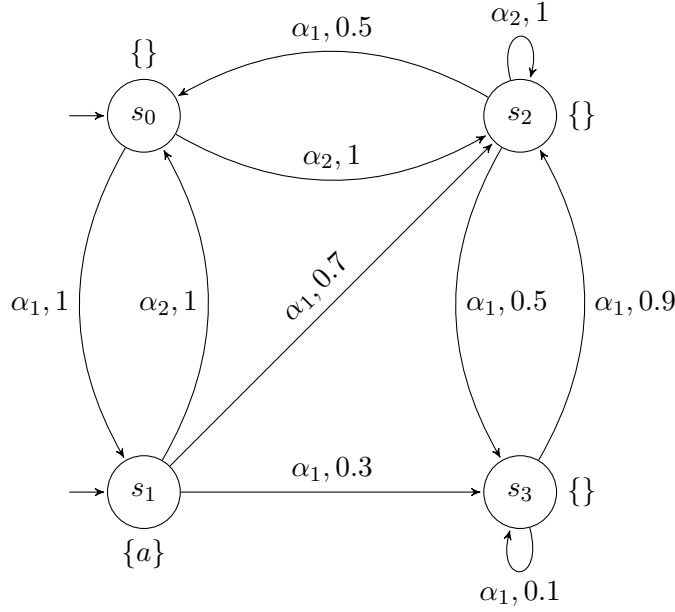


Figure 3: An exemplary MDP \mathfrak{M} with two initial states s_0 and s_1 and two actions α_1 and α_2 .

As PGCL programs contain parameters, we subsequently introduce *parametric MDPs* which allow rational parameters on their transitions. Parametric probabilistic transition systems are examined in [9]. We denote the set of all rational functions as \mathcal{F}_{Var} where $Var = \{v_1, v_2, \dots, v_n\}$ is a finite set of variables.

Definition 7 (Parametric MDP (PMDP)). *A parametric Markov decision process is a six-tuple $\mathfrak{M} = (S, S_{init}, Act, Var, P, AP, L)$ where all components except Var and P are equivalently defined as with MDPs. Var denotes a finite set of variables. $P: S \times Act \times S \rightarrow \mathcal{F}_{Var}$ is the transition probability function.*

Note that MDPs form a subclass of PMDPs as the image of their transition functions can be expressed as constant rational functions. Hence, PMDPs are of the most potent model type introduced in this paper.

To resolve the above-mentioned nondeterminism, we introduce *schedulers*. Schedulers are capable of reducing a given (P)MDP to a DTMC in means of removing the nondeterminism. Notice that applying a scheduler results subsequently in a loss of expressiveness. In this paper only *memoryless* and *deterministic* schedulers are used, meaning that schedulers can not base their decisions on previously taken actions. A memorizing scheduler on the other hand bases its resolving decisions on the complete path chosen prior to the current decision. A randomized scheduler, in contrast to a deterministic scheduler, defines a probability distribution over possible actions to be taken, thus resolves the nondeterminism in a probabilistic manner.

Definition 8 (Scheduler). *Let $\mathfrak{M} = (S, S_{init}, Act, P, AP, L)$ be a (P)MDP. A scheduler $\sigma_{\mathfrak{M}}$ is a tuple containing an initial state and a function, thus $\sigma_{\mathfrak{M}} = (s_{init}, \sigma_{act})$. Here, $s_{init} \in S_{init}$ and σ_{act} is a function $\sigma_{act}: S \rightarrow Act$. We constraint $\sigma_{act}(s) \in \{a \in Act \mid \exists s' \in S(P(s, a, s') \neq 0)\}$. The set of all possible schedulers for \mathfrak{M} is denoted by $Sched_{\mathfrak{M}}$.*

The definition constraints the scheduler to only choose an action available to the current state. We can now apply schedulers to Markov decision processes and hence introducing a determinization of nondeterministic probabilistic models. We call the Markov chain yielded by the application of a scheduler the *scheduler-induced Markov chain*.

Definition 9 (Scheduler-induced Markov chain). *Let $\mathfrak{M} = (S, S_{init}, Act, P, AP, L)$ be a (P)MDP and $\sigma_{\mathfrak{M}} = (s_{init}, \sigma_{act}) \in Sched_{\mathfrak{M}}$. The scheduler-induced Markov chain is defined by $\mathfrak{M}^{\sigma} = (S, s_{init}, P', AP, L)$ where for all states s, s' and all actions α the transition function P' is defined by $P(s, \sigma_{act}(s), s') = p \implies P'(s, s') = p$ and $(P(s, \alpha, s') \neq 0 \wedge \sigma_{act}(s) \neq \alpha) \implies P'(s, s') = 0$.*

Consider the exemplary MDP \mathfrak{M} of Figure 3. We define our scheduler to be $\sigma = (s_0, \{s \mapsto \alpha_1 \mid \forall s \in \{s_0, s_1, s_2, s_3\}\})$ and obtain the scheduler-induced DTMC \mathfrak{M}^{σ} depicted in Figure 4.

To fuse probability theory and stochastic models, we introduce a probability space for a given DTMC. Since MDPs include multiple probability distributions over a state's successors, a direct application of a probability space is not given. We rather transform every MDP into a DTMC using the previously introduced concept of schedulers.

Definition 10 (Paths of probabilistic models). *For a DTMC \mathfrak{M} a path π is a sequence $\pi = s_0 s_1 s_2 \dots \in S^{\omega}$ where $s_i \in S$ and $P(s_i, s_{i+1}) > 0 \forall i \in \mathbb{N}$. $Paths(\mathfrak{M}) \subseteq S^{\omega}$ is the set of all paths of \mathfrak{M} starting in the initial state s_0 . $Paths_{fin}(\mathfrak{M}) \subseteq S^*$ is the set of finite paths starting in the initial state. The paths of \mathfrak{M} starting in some state $s \in S$ are identified with the set $Paths(\mathfrak{M}, s)$. The set of finite paths starting in s is called $Paths_{fin}(\mathfrak{M}, s)$ and the set of finite paths starting in a state s and ending in a state s' is denoted by $Paths_{fin}(\mathfrak{M}, s, s')$. $\pi[i]$ is the i -th state of a path π for some $i \geq 0$.*

As an example, $\pi = s_0(s_2 s_3)^{\omega}$ is a path of the DTMC \mathfrak{M} depicted in Figure 2 for which $\pi \in Paths(\mathfrak{M})$ holds.

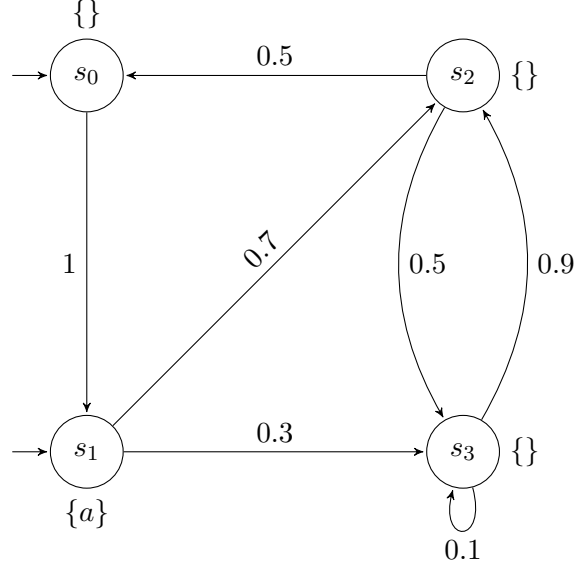


Figure 4: The scheduler-induced Markov chain \mathfrak{M}^σ for the MDP \mathfrak{M} of Figure 3 for the scheduler $\sigma = (s_0, \{s \mapsto \alpha_1 \mid \forall s \in \{s_0, s_1, s_2, s_3\}\})$.

To define a set of possible events on a DTMC, the notion of cylinder sets for a given finite path π_f is used. They represent the set of paths starting with the finite fragment π_f .

Definition 11 (Cylinder set of a path). *For a given DTMC $\mathfrak{M} = (S, s_{init}, P, AP, L)$ the cylinder set of a finite path π_f is defined as $Cyl(\pi_f) = \{\pi \in Paths(\mathfrak{M}) \mid \pi \in \pi_f \cdot S^\omega\}$.*

Intuitively, a cylinder set for a finite path fragment π_f contains all paths of the DTMC having π_f as a prefix. If we again consider the DTMC \mathfrak{M} of Figure 2, the cylinder set of the finite path $s_0s_1s_2s_3 \in Path_{fin}(\mathfrak{M})$ is given as $Cyl(s_0s_1s_2s_3) = \{s_0s_1s_2s_3((s_2s_3)^+(s_5s_3)^* + (s_2s_3)^*(s_5s_3)^+ + (s_2s_3)^*(s_5s_3)^*s_5s_4s_1s_2s_3)^\omega\}$.

The set of elementary events Ω can then be defined as the set of all paths of the DTMC. This results in the following definition of the σ -algebra.

Definition 12 (σ -algebra of a DTMC). *Let \mathfrak{M} be a DTMC. For the set of elementary events $\Omega_{\mathfrak{M}} = Paths(\mathfrak{M})$ we define the σ -algebra of \mathfrak{M} as the smallest σ -algebra $\Sigma_{\mathfrak{M}} = \{Cyl(\pi_f) \mid \pi_f \in Path_{fin}(\mathfrak{M})\}$.*

For the last part of the probability space a probability measure needs to be defined. We use the intuitive notion of multiplying the probabilities of the transition probability function along a path.

Definition 13 (Probability measure of a DTMC). *Let $\mathfrak{M} = (S, s_{init}, P, AP, L)$ be a DTMC. For a σ -algebra $\Sigma_{\mathfrak{M}}$ as in Definition 12 we define the probability measure*

$Pr_{\mathfrak{M}}(\text{Cyl}(s_0 \dots s_n)) = \prod_{i=1}^n P(s_{i-1}, s_i)$ for all $\text{Cyl}(s_0 \dots s_n) \in \Sigma_{\mathfrak{M}}$. If $s_0 = s_n$, we set $Pr_{\mathfrak{M}}(\text{Cyl}(s_0)) = 1$.

Since we now have gathered all parts of our probability space, we can easily define it.

Definition 14 (Probability space of a DTMC). *Let \mathfrak{M} be a DTMC. The probability space of \mathfrak{M} is $(\text{Paths}(\mathfrak{M}), \Sigma_{\mathfrak{M}}, Pr_{\mathfrak{M}})$ with $\Sigma_{\mathfrak{M}}$ and $Pr_{\mathfrak{M}}$ as in Definition 12 respectively Definition 13.*

We can now again examine the executions of the DTMC \mathfrak{M} presented in Figure 2 on Page 6. Consider the probability of reaching the state s_2 which is labeled with an a . Almost every infinite paths visits s_2 sometimes: Only the path $s_0(s_3s_5)^\omega$ does not contain s_2 . We are interested in the probability of every other run, hence $Pr_{\mathfrak{M}}(\text{Runs}(\mathfrak{M}) \setminus \{s_0(s_3s_5)^\omega\})$. But since $Pr_{\mathfrak{M}}$ defines a probability measure, it follows that $Pr_{\mathfrak{M}}(\text{Runs}(\mathfrak{M}) \setminus \{s_0(s_3s_5)^\omega\}) = Pr_{\mathfrak{M}}(\text{Runs}(\mathfrak{M})) - Pr_{\mathfrak{M}}(\{s_0(s_3s_5)^\omega\}) = 1 - Pr_{\mathfrak{M}}(\{s_0(s_3s_5)^\omega\})$. It is easy to see that the probability of the path $s_0(s_3s_5)^\omega$ is zero since $\lim_{n \rightarrow \infty} (0.1 \cdot (0.6 \cdot 0.5)^n) = 0$, hence the probability of reaching a state labeled with an a equals one.

2.4 Probabilistic computation tree logic

In the following, we introduce a logic to allow for formal reasoning of PGCL program properties. In this paper the *Probabilistic computation tree logic (PCTL)* is used. PCTL resembles the well-known CTL logic [10], and additionally introduces an operator for checking the bounds of probability of runs.

PCTL was first defined by Hansson and Jonsson in 1994 [11] with the objective of introducing a logic for reasoning about time and reliability. PCTL allows to constraint formulas to certain probabilities such as saying that “something bad will only happen with a maximum probability of 0.01”. Hence, we are able to express properties such as the previously examined “reaching an a -labeled state somewhen” formally.

Just as in CTL we differentiate between *state* and *path* formulas. State formulas are evaluated for a given state, whereas path formulas define properties of whole runs. The syntax of both formula types are given as context-free grammars.

Definition 15 (Syntax of PCTL). *Let AP be a set of atomic propositions. A PCTL state formula φ over the set AP conforms to the following grammar, where $J \subseteq [0, 1] \subset \mathbb{Q}$:*

$$\varphi ::= \text{true} \mid a \mid \varphi \wedge \varphi \mid \neg \varphi \mid \mathbb{P}_J(\psi)$$

A PCTL path formula ψ conforms to the following grammar, for $n \in \mathbb{N}$:

$$\psi ::= X\varphi \mid \varphi U \varphi \mid \varphi U^{\leq n} \varphi$$

The operator X is called the *next* operator, whereas U represents the *until* operator. The third operator $U^{\leq n}$ is a novel addition in comparison to CTL, denoting the *step-bounded until*. All three path formula operators are recursively embedding state formulas, resulting in an alternated nesting of path and state formulas. The operator \mathbb{P}_J replaces the CTL operators \exists and \forall therefore allows probabilistic reasoning about branches. Intuitively, \mathbb{P}_J is true if the probability of all paths starting in the current state and satisfying the given path formula is in the set J . If $J = [0, i]$ we shorten the probability operator to be written as $\mathbb{P}_{\leq i}$, if $J = [0, i)$ we write $\mathbb{P}_{< i}$, if $J = [i, 1]$ we write $\mathbb{P}_{\geq i}$ and if $J = (i, 1]$ we write $\mathbb{P}_{> i}$. In case $J = \{i\}$ we write $\mathbb{P}_{=i}$.

Definition 16 (Subformulas of a PCTL formula). *Let φ be a PCTL state or path formula. The set $Sub(\varphi)$ is the smallest set such that the following holds:*

- $\varphi \in Sub(\varphi)$.
- If $\neg\psi \in Sub(\varphi)$ then $\psi \in Sub(\varphi)$.
- if $\psi_1 \wedge \psi_2 \in Sub(\varphi)$ then $\psi_1 \in Sub(\varphi)$ and $\psi_2 \in Sub(\varphi)$.
- If $\mathbb{P}_J(\psi) \in Sub(\varphi)$ then $\psi \in Sub(\varphi)$.
- If $X\psi \in Sub(\varphi)$ then $\psi \in Sub(\varphi)$.
- If $\psi_1 U \psi_2 \in Sub(\varphi)$ then $\psi_1 \in Sub(\varphi)$ and $\psi_2 \in Sub(\varphi)$.
- If $\psi_1 U^{\leq n} \psi_2 \in Sub(\varphi)$ then $\psi_1 \in Sub(\varphi)$ and $\psi_2 \in Sub(\varphi)$.

In the following, some short examples and an intuitive explanation on PCTL formulas are given.

1. $\mathbb{P}_{\leq 0.3}(G\neg a)$ – The probability of never visiting a is less than 0.3.
2. $\mathbb{P}_{=1}(F(b \wedge \mathbb{P}_{=1}(Ga)))$ – It is almost-sure that b is visited sometimes and from there on, it is almost sure that on every position a holds.
3. $\mathbb{P}_{> 0.5}(aUb) \wedge \mathbb{P}_{=0}(Fc)$ – The probability of the runs with a finite prefix of the form $aa \dots ab$ is greater than 0.5 and the probability of encountering a c is 0.

To breathe life in these previously semantically defined formulas and formalize the intuitive understanding, we introduce the satisfaction relation \models on states and PCTL state formulas respectively paths and path formulas.

Definition 17 (Satisfaction relation on PCTL state formulas). *Let $\mathfrak{M} = (S, s_{init}, P, AP, L)$ be a DTMC, $s \in S$, φ , φ_1 and φ_2 PCTL state formula, ψ a PCTL path formulas and $J \subseteq [0, 1] \subset \mathbb{Q}$. The satisfaction relation \models between states and PCTL state formulas is defined as follows:*

$$\begin{aligned}
s \models a & \quad :\Leftrightarrow a \in L(s) \\
s \models \neg\varphi & \quad :\Leftrightarrow s \not\models \varphi \\
s \models \varphi_1 \wedge \varphi_2 & \quad :\Leftrightarrow s \models \varphi_1 \wedge s \models \varphi_2 \\
s \models \mathbb{P}_J(\psi) & \quad :\Leftrightarrow Pr_{\mathfrak{M}}(\{\pi \in Paths(\mathfrak{M}, s) \mid \pi \models \psi\}) \in J
\end{aligned}$$

Definition 18 (Satisfaction relation on PCTL path formulas). *Let $\mathfrak{M} = (S, s_{init}, P, AP, L)$ be a DTMC, $\pi \in Paths(\mathfrak{M})$ and φ, φ_1 and φ_2 PCTL state formulas. The satisfaction relation \models between paths and PCTL path formula is defined as follows:*

$$\begin{aligned} \pi \models X\varphi & \quad :\Leftrightarrow \pi[1] \models \varphi \\ \pi \models \varphi_1 U \varphi_2 & \quad :\Leftrightarrow \exists j(j \geq 0 \wedge \pi[j] \models \varphi_2 \wedge \forall i(0 \leq i < j \rightarrow \pi[i] \models \varphi_1)) \\ \pi \models \varphi_1 U^{\leq n} \varphi_2 & \quad :\Leftrightarrow \exists j(0 \leq j \leq n \wedge \pi[j] \models \varphi_2 \wedge \forall i(0 \leq i < j \rightarrow \pi[i] \models \varphi_1)) \end{aligned}$$

Note that the common path formula operators **G** (*globally*) and **F** (*eventually*) can be derived just as in CTL: $\mathbb{P}_J(\mathbf{F}\varphi) \equiv \mathbb{P}_J(true U \varphi)$ and $\mathbb{P}_J(\mathbf{G}\varphi) \equiv \neg \mathbb{P}_J(\mathbf{F}\neg\varphi)$.

Definition 19 (Satisfaction relation of PCTL formulas and DTMCs). *Let $\mathfrak{M} = (S, s_{init}, P, AP, L)$ be a DTMC and φ a PCTL state formula. The satisfaction relation of \mathfrak{M} and φ is defined as $\mathfrak{M} \models \varphi :\Leftrightarrow s_{init} \models \varphi$.*

Definition 20 (Equivalence of formulas). *Let φ_1 and φ_2 be PCTL state formulas. $\varphi_1 \equiv \varphi_2$ iff for all DTMCs \mathfrak{M} it holds that $\mathfrak{M} \models \varphi_1 \Leftrightarrow \mathfrak{M} \models \varphi_2$.*

Definition 21 (Satisfaction set of PCTL state and path formulas). *Let $\mathfrak{M} = (S, S_{init}, P, AP, L)$ be a DTMC, φ a PCTL state formula and ψ a PCTL path formula. The satisfaction set of φ on \mathfrak{M} is $Sat_{\mathfrak{M}}(\varphi) = \{s \in S \mid s \models \varphi\}$. For the PCTL path formula ψ , $Sat_{\mathfrak{M}}(\psi) = \{\pi \in Paths(\mathfrak{M}, s) \mid s \in S \wedge \pi \models \psi\}$.*

Sat(φ) is analogously defined for MDPs. We define $Sat(\psi) = \{\pi \in \{Paths(\mathfrak{M}^\sigma) \mid \forall \sigma \in Sched(\mathfrak{M})\} \mid \pi \models \psi\}$.

As an example, we consider the simple DTMC \mathfrak{M} of Figure 5 and the before mentioned formulas $\varphi_1 = \mathbb{P}_{\leq 0.3}(\mathbf{G}\neg a)$, $\varphi_2 = \mathbb{P}_{=1}(\mathbf{F}b \wedge \mathbb{P}_{=1}(\mathbf{G}a))$ and $\varphi_3 = \mathbb{P}_{>0.5}(aUb) \wedge \mathbb{P}_{=0}(\mathbf{F}c)$. First off, we determine the satisfaction set of the PCTL path formula $\mathbf{G}\neg a$: $Sat_{\mathfrak{M}}(\mathbf{G}\neg a) = \{s_0^+ s_1^\omega\}$. We then continue to model the satisfaction set as a countable union of cylinder sets which yields $\{s_0^+ s_1^\omega\} = \bigcup_{n \geq 1} Cyl(s_0^n s_1)$. To calculate the probability of this set, we take for every $n \geq 1$ the probability of each set $Cyl(s_0^n s_1)$ which is given by $Pr_{\mathfrak{M}}(Cyl(s_0^n s_1)) = 0.5^n$. By stochastic theory it follows that $Pr_{\mathfrak{M}}(\bigcup_{n \geq 1} Cyl(s_0^n s_1)) = \sum_{n=1}^{\infty} 0.5^n = 1$. Hence, $\mathfrak{M} \not\models \varphi_1$ since $1 \notin [0, 0.3]$. Additionally, $\mathfrak{M} \not\models \varphi_2$ as $Sat_{\mathfrak{M}}(\mathbf{G}a) = \{s_0^\omega\}$ and $Sat_{\mathfrak{M}}(\mathbb{P}_{=1}\mathbf{G}a) = \emptyset$. It then follows directly that $Sat_{\mathfrak{M}}(\varphi_2) = \emptyset$ as the intersection with the empty set yields the empty set again, as $Pr_{\mathfrak{M}}(\emptyset) = 0$. However, the third formula φ_3 is satisfied by \mathfrak{M} : $Sat_{\mathfrak{M}}(aUb) = \{s_0^+ s_1^\omega\}$ and as seen above $Pr_{\mathfrak{M}}(\{s_0^+ s_1^\omega\}) = 1$. Since no c -labeled state is present in \mathfrak{M} the probability of $\mathbf{F}c$ equals zero.

Definition 22 (Reachability property). *Let $J \subseteq [0, 1] \subset \mathbb{Q}$ and φ a PCTL state formula not containing the operator \mathbb{P} . Every formula of the form $\mathbb{P}_J(\mathbf{F}\varphi)$ denotes a reachability property.*

In practice, the probabilistic operator $\mathbb{P}_J(\cdot)$ is mostly used with only an upper or lower boundary on the probability. We define $\bowtie \in \{<, >, \leq, \geq\}$ and $\overleftarrow{\bowtie} := \leq$, $\overrightarrow{\bowtie} := \geq$, $\overline{\leq} := >$ and $\overline{\geq} := <$. To shorten the writing in the following, we use $<$ and $>$ as a shorthand for $<$ or \leq respectively $>$ or \geq .

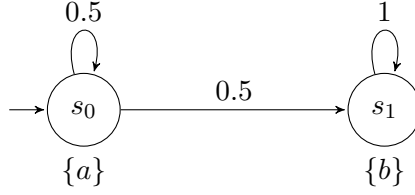


Figure 5: A simple exemplary DTMC \mathfrak{M} over the labels $L = \{a, b, c\}$ for which the satisfaction relation \models is examined for various formulas.

Definition 23 (Probabilistically bounded PCTL formula). For $i \in [0, 1] \subset \mathbb{Q}$ and a PCTL path formula ψ we denote $\mathbb{P}_{\bowtie i}(\psi)$ to be a probabilistically bounded PCTL formula. For $\leq \in \{<, \leq\}$, we call $\leq i$ an upper bound, and for $\geq \in \{>, \geq\}$ we call $\geq i$ a lower bound. $\mathbb{P}_{\bowtie i}(\psi)$ denotes the inversely probabilistically bounded PCTL formula to $\mathbb{P}_{\bowtie i}(\psi)$.

In this paper, only probabilistically bounded PCTL formulas are examined. This rules out some PCTL formulas such as $\mathbb{P}_J(\cdot)$ where J – fixing an enumeration of \mathbb{Q} – is the set of every rational number between 0 and 1 whose enumeration index is even. Those formulas are considered to only have a small practical relevance.

In this paper, only the *verification* of lower bounded properties of the form $\mathbb{P}_{\geq i}(\psi)$ where $i \in [0, 1] \subset \mathbb{Q}$ are considered. This is a result of practical limitations of the iterative model checking approach which are elaborated on in the upcoming sections of the paper. Reversely, the above statement implies that we are interested in checking whether $\mathbb{P}_{\leq i}(\psi)$ is violated – It directly follows that $\mathbb{P}_{< i}(\psi)$ holds.

Since we did not give a formal definition of PCTL formulas on MDPs, we will concisely introduce the reader to the topic. For more details we refer to [8] pages 866 and following. In short, the PCTL formula $\mathbb{P}_J(\psi)$ is satisfied by an MDP \mathfrak{M} iff for all schedulers $\sigma_{\mathfrak{M}} \in \text{Sched}_{\mathfrak{M}}$ holds that the scheduler-induced DTMC $\mathfrak{M}^{\sigma} \models \mathbb{P}_J(\psi)$.

This definition introduces a quantifier to the satisfaction relation. As a result, the formula $\mathbb{P}_{\bowtie i}(\psi)$ and the negation of the inversely probabilistically bounded property $\neg \mathbb{P}_{\bowtie i}(\psi)$ are not equal for MDPs since the negation of the above-mentioned definition results in a swapping of quantifiers, thus $\neg \mathbb{P}_{\bowtie i}(\psi)$ is satisfied iff for *some* scheduler the scheduler-induced DTMC satisfies $\mathbb{P}_{\bowtie i}(\psi)$ – But in contrast $\mathbb{P}_{\bowtie i}(\psi)$ is satisfied iff for *all* schedulers the scheduler-induced DTMC satisfies the formula.

On the other hand, the equivalence $\mathbb{P}_{\bowtie i}(\psi) \equiv \neg \mathbb{P}_{\bowtie i}(\psi)$ as defined in Definition 20 holds for DTMCs due to the absence of quantifiers in the definition of the satisfaction relation. This is easy to see since the event of all paths satisfying ψ and starting in the initial state equals the complement of the event of all paths not satisfying ψ and starting in the initial state – Hence, the probability of all paths satisfying ψ is $\bowtie i$ if and only if the probability of the complement of this events is $\bowtie i$ by definition of the probability

space for probabilistic models.

2.5 Counterexamples

For the most part of program verification one is interested in checking the violation of properties: If such a violation is found, the given system can be adapted and refined in such a way that it will have the desired property. To see why *exactly* a property was not satisfied by the system, we introduce *counterexamples*. Intuitively, a counterexample is a witness of the property violation by defining a finite set of paths violating the property. We can represent such a finite set of path as a *critical subsystem* for the given model.

Definition 24 (Evidences and counterexample). *Let $\mathfrak{M} = (S, s_{init}, P, Act, AP, L)$ be an MDP, $\varphi = \mathbb{P}_{<i}(\psi)$ a probabilistically upper bounded PCTL formula where $i \in [0, 1] \subset \mathbb{Q}$. Let $\sigma \in Sched_{\mathfrak{M}}$.*

A finite paths $\pi_f \in Paths_{fin}(\mathfrak{M}^\sigma)$ is an evidence for ψ iff for all $\pi \in Cyl(\pi_f)$ it holds that $\pi \models \psi$ on \mathfrak{M}^σ .

A counterexample for φ is a finite set of evidences $C_\varphi = \{\pi_0, \dots, \pi_n\} \subseteq Paths_{fin}(\mathfrak{M}^\sigma)$ such that $Pr_{\mathfrak{M}^\sigma}(\bigcup_{j=1}^n Cyl(\pi_j)) \not\leq i$.

Hence, if we consider the exemplary DTMC \mathfrak{M} depicted in Figure 5 and the formula $\varphi_1 = \mathbb{P}_{\leq 0.3}(\mathbf{G}\neg a)$ again, we obtain the counterexample $C_{\varphi_1} = \{\pi_f \in Paths_{fin}(\mathfrak{M}) \mid \pi_f \neq s_0^\omega\}$ as we already determined that $Pr_{\mathfrak{M}}(C_{\varphi_1}) = 1$ which is obviously $\not\leq 0.3$.

Since sets of evidences can be bulky in some use cases, the notion of critical subsystems is introduced. They were firstly outlined by Aljazzar and Leue in [12]. A critical subsystem is a subsystem of the given model as in Definition 5 which induces a set of runs whose probabilities in sum violate the given formula $\mathbb{P}_{<i}(\psi)$. It additionally resolves the nondeterminism using a scheduler that leads to a violation of the property. Thus, a critical subsystem is nothing more than a handy representation of a set of evidences.

Definition 25 (Critical subsystem). *Let $\mathfrak{M} = (S, s_{init}, Act, P, AP, L)$ be an MDP, ψ a PCTL path formula and $i \in [0, 1] \subset \mathbb{Q}$. Let $\sigma \in Sched_{\mathfrak{M}}$. A critical subsystem \mathfrak{M}' for the formula $\varphi = \mathbb{P}_{<i}(\psi)$ is a subsystem of \mathfrak{M}^σ for which holds $\mathfrak{M}' \not\models \varphi$, i.e. $Pr_{\mathfrak{M}'}(\{\pi \in Paths(\mathfrak{M}') \mid \pi \not\models \psi\}) \not\leq i$.*

3 The probabilistic guarded command language

In order to define probabilistic programs, we employ the Probabilistic Guarded Command Language (PGCL). The PGCL, first defined by McIver and Morgan in [13], is derived from Dijkstra’s Guarded Command Language (GCL) [4]. Whereas standard GCL programs allow for guarded command execution, PGCL has the addition of probabilistic execution choices.

3.1 Syntax of PGCL

As a starter, we will examine the syntax of the probabilistic guarded command language by means of the exemplary PGCL program P_1 known from the introduction. Figure 6 illustrates this PGCL program which simulates a network message delivery system. The transmission quality of the network fluctuates over time, hence the probability of successfully delivering a message changes. As we see, we use a C-like syntax with curly brackets as block delimiters, `if` and `while` keywords for conditionals respectively loops and the operator `:=` for assignments to variables. The set of all variables of a PGCL program P is denoted by Var_P , which is $\{x\}$ for our exemplary program P_1 . Since PGCL supports probabilistic executions, we introduce the $[p]$ operator which indicates that the chance of executing the left block is p and executing the right block is $1-p$. As we are additionally interested in parameterized probabilistic executions, we equip every PGCL program P with a set of parameters $Params_P$. A parameter is considered to be a non-initialized variable which can only be used as a probability for the $[p]$ -operator. In the example, the set $Param_{P_1}$ is empty as we are not given any parameters. We will later examine the functioning of the presented example program P_1 in detail, since this section focuses on the syntax of PGCL programs.

```
1 double a := 0.5;
2 int success := 0;
3 while(success = 0) {
4     // tries to send message
5     {success := 1;} [a] {success := 0;}
6     // simulates transmission fluctuation
7     {a := a + 0.01;} [0.5] {a := a - 0.01;}
8     if(a < 0 | a > 1) {
9         a := 0.5;
10    }
11 }
```

Figure 6: A simple PGCL program P_1 simulating a message delivery system with a fluctuating network quality.

Definition 26 (Syntax of the PGCL). A PGCL program P conforms to the following context-free grammar:

$$\begin{aligned}
\langle \text{program} \rangle & ::= \{ \langle \text{simple_statement} \rangle \mid \langle \text{compound_statement} \rangle \} \\
\langle \text{simple_statement} \rangle & ::= [\text{int} \mid \text{double}] \langle \text{variable} \rangle := \langle \text{expression} \rangle ; \\
& \quad \mid \text{observe}(\langle \text{expr} \rangle) ; \\
\langle \text{compound_statement} \rangle & ::= \text{if}(\langle \text{expression} \rangle) \{ \langle \text{program} \rangle \} [\text{else} \{ \langle \text{program} \rangle \}] \\
& \quad \mid \text{while}(\langle \text{expression} \rangle) \{ \langle \text{program} \rangle \} \\
& \quad \mid \{ \langle \text{program} \rangle \} [] \{ \langle \text{program} \rangle \} \\
& \quad \mid \{ \langle \text{program} \rangle \} [\langle \text{probability_expression} \rangle] \{ \langle \text{program} \rangle \}
\end{aligned}$$

$\langle \text{expression} \rangle$ denotes expressions such as addition, conjunction, and negation and $\langle \text{variable} \rangle$ represents a string over the alphabet of latin letters. $\langle \text{probability_expression} \rangle$ constraints an expression to be of a probabilistic type, i.e. representing a rational number between 0 and 1.

The context-sensitive part of the PGCL grammar is defined analogously to other programming languages: Before the usage of a variable in a expression it shall be declared previously, types of variables shall match in expressions, and there shall not be two declarations of the same variable.

In this paper, PGCL programs are typed over integers and doubles. For sake of simplicity we assume in the following that both integer and double variables have the domain \mathbb{Q} .

In addition to non-probabilistic programs we introduced $\{ \dots \} [p] \{ \dots \}$ to the syntax, which we call the binary probabilistic choice operator. The left part is executed with probability p and the right part with probability $1 - p$. Besides probabilistic choices also nondeterministic choices are permitted by the binary operator $\{ \dots \} [] \{ \dots \}$.

Similar to the well-known assert-statement in other languages, the `observe(...)` operator blocks runs where its condition does not hold. It additionally normalizes the probability of the remaining runs in respect to the total where the probabilities of the blocked runs are removed.

To enable formal reasoning on PGCL programs we will introduce a graph representation later on. We assume that every statement has a unique identifier in \mathbb{N} called the *location* of the statement. We set the very first statement of a program to have the location 0. We denote the set of all program locations by Loc_P .

In order to set up the rules for the program graph construction, we define the function $loc(s)$ to return the location of a statement. For a program x , $loc(x)$ returns the location of the first statement of x . Additionally, $next(s)$ returns the location of the next statement on the same level of recursion. If there is no next statement on the same level and s is no direct child of a loop statement $next(s)$ returns the location of the next statement on the level above. If s is the last statement in its (sub)program and in fact a direct child of a loop statement l , $next(s)$ returns $loc(l)$. If s is the last statement of the whole program, $next(s)$ returns -1 .

Definition 27 (Program graph of a PGCL program). *Let \mathbf{P} be a PGCL program. The graph representation of \mathbf{P} is a labeled and directed graph $G_{\mathbf{P}} = (V, L, E)$ where V represent the vertices, L a set of labels, and $E \subseteq V \times L \times V$ the labeled edges.*

$$V = \text{Loc}_{\mathbf{P}} \cup \{-1\} \subseteq \mathbb{Z}.$$

L contains every simple statement and expression of \mathbf{P} along with its negation as a string. We additionally add the labels $\{n_1, n_2\}$ to L .

For each statement s of \mathbf{P} with location l we add the following edges to E :

- *If $s = \text{if}(e)\{p\}$ then $(l, e, \text{loc}(p)) \in E \wedge (l, \neg e, \text{next}(s)) \in E$.*
- *If $s = \text{if}(e)\{p_1\}\text{else}\{p_2\}$ then $(l, e, \text{loc}(p_1)) \in E \wedge (l, \neg e, \text{loc}(p_2)) \in E$.*
- *If $s = \text{while}(e)\{p\}$ then $(l, e, \text{loc}(p)) \in E \wedge (l, \neg e, \text{next}(s)) \in E$.*
- *If $s = p_1 \llbracket p_2$ then $(l, n_1, \text{loc}(p_1)) \in E \wedge (l, n_2, \text{loc}(p_2)) \in E$.*
- *If $s = p_1 [e] p_2$ then $(l, \langle e \rangle, \text{loc}(p_1)) \in E \wedge (l, \langle 1 - e \rangle, \text{loc}(p_2)) \in E$.*
- *If $s = v := e$; then $(l, s, \text{next}(s)) \in E$.*
- *If $s = \text{observe}(e)$; then $(l, s, \text{next}(s)) \in E$.*

Thus, the program graph $G_{\mathbf{P}}$ is built depending on the type of its statements. Guards are represented by two outgoing transitions, one possibility for each boolean evaluation of the guard. Loops are represented by circles in $G_{\mathbf{P}}$, looping from the last statement's node of a loop to the node representing the loop itself. For every probabilistic choice two successors are introduced. Nondeterminism is represented in an analog fashion. The transitions are labeled with n_1 respectively n_2 to be able to differentiate between the nondeterministic choices later on.

We examine this construction by forming the program graph $G_{\mathbf{P}}$ of the example network-transmission simulator \mathbf{P} depicted in Figure 6. This graph is presented in Figure 7. Node 2 represents the loop statement, where the loop's guard is checked. If the loop guard does not hold, the program terminates, i.e. ends up in the -1 node. If this is not the case, the program continues to execute the inner statements, beginning with the probabilistic branch of setting **success** to 1 respectively 0. The curved brackets represent probabilities, therefore both outgoing transitions of 3 are labeled with $\langle p \rangle$, respectively $\langle 1-p \rangle$. This scheme is repeated for the probabilistic modeling of the network quality fluctuation. We then continue to check the guard of the **if** statement and again check the loop's condition via node 2.

During the execution of a PGCL program, multiple variable valuations may be reached. A variable valuation, or valuation in short, is a function mapping to each variable a value of the variable's type.

Definition 28 (Valuations). *Let \mathbf{P} be a PGCL program. A valuation v is a function $v: \text{Var}_{\mathbf{P}} \rightarrow \mathbb{Q}$ where v is constraint to respect the types of the variables. Var as $\text{Val}(\text{Var})$ is the set of all valid valuations over a set of variables.*

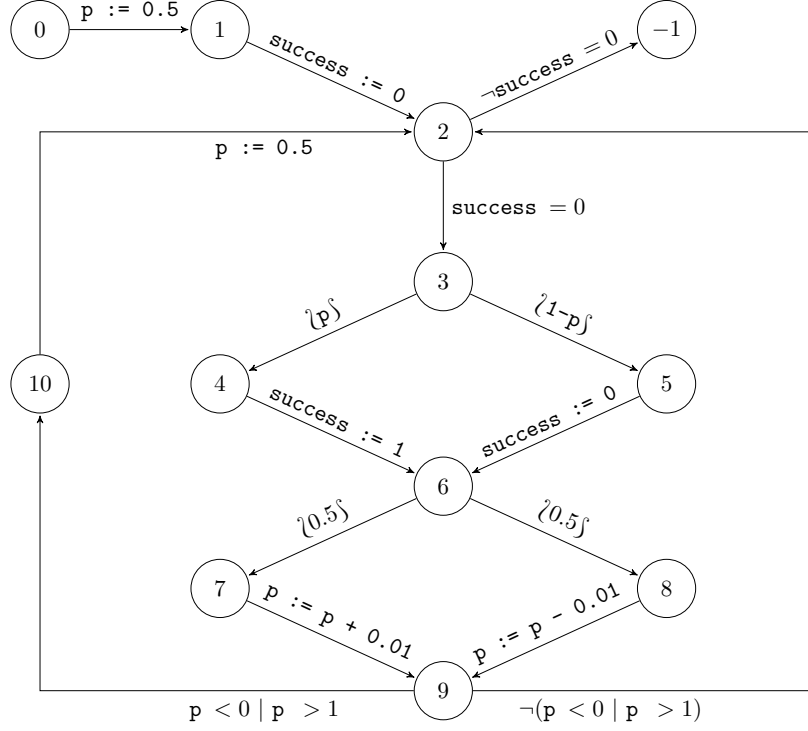


Figure 7: The associated program graph G_P of the exemplary PGCL program P depicted in Figure 6.

To relate expressions and valuations we introduce the notion of evaluating an expression e under a valuation v .

Definition 29 (Expression evaluations). *Let v be a valuation and e an expression. $e[v] \in \mathbb{Q}$ is the evaluation of the expression e where every variable of e is assigned its valuation defined by v . To update valuations, $v[x := i] = \{x \mapsto i\} \cup (v \setminus \{x \mapsto j \mid j \in \mathbb{Q}\})$ for some variable x and type-matching value $i \in \mathbb{Q}$.*

3.2 Semantics of PGCL

To define the semantics of PGCL programs, we will derive for every PGCL program graph G_P a probabilistic model \mathfrak{M}_P representing all its possible executions. This construction is defined in terms of structural operational semantic rules (SOS rules) for PGCL programs. For programs containing no nondeterministic statements and parameters a DTMC suffices for the model. If the program incorporates at least one statement of the form $\{\dots\}[\]\{\dots\}$ we construct an MDP. For parameters, we employ parametric MDPs where the parameterized probabilistic choices are used to represent the parameters of the PGCL program.

Definition 30 (Operational semantics of PGCL programs). *Let P be a PGCL program and $G_P = (V, L_P, E)$ its associated program graph. A finite set $Expr$ of expressions over Var_P is given. The PMDP $\mathfrak{M}_P = (S, S_{init}, Act, Var, P, AP, L)$ is defined as follows:*

- $S = (V \times Val(Var_P)) \cup \{\downarrow, \perp\}$ is the set of the model's nodes.
- $S_{init} = \{(0, \{x \mapsto 0 \mid \forall x \in Var_P\})\}$ is the set of initial nodes.
- $Act = \{\circ, n_1, n_2\}$.
- $Var = Var_P$.
- $AP = Expr \cup \{\downarrow, \perp\}$ defines the labels to be the given expressions plus the special nodes \downarrow and \perp .
- $L: S \rightarrow 2^{AP}$ where $\forall l \in V \quad \forall v \in Val(P) \quad \forall e \in Expr$ we define $e[v] = 1 \Rightarrow e \in L((l, v))$, $L(\downarrow) = \{\downarrow\}$, and $L(\perp) = \{\perp\}$.
- The transition function P is defined by the following SOS rules. For every valuation $v \in Val(Var_P)$, every node n of G_P , and outgoing edge (n, l, n') for some $n' \in V$:

$$\text{Assignments} \quad \frac{l = x := e \quad n \geq 0}{P((n, v), \circ, (n', v[x := e[v]])) = 1}$$

$$\text{Observations} \quad \frac{l = \text{observe}(e) \quad e[v] = 1 \quad n \geq 0}{P((n, v), \circ, (n', v)) = 1}$$

$$\frac{l = \text{observe}(e) \quad e[v] \neq 1 \quad n \geq 0}{P((n, v), \circ, \downarrow) = 1}$$

$$\text{Expressions} \quad \frac{l = e \text{ for an expression } e \quad e[v] = 1 \quad n \geq 0}{P((n, v), \circ, (n', v)) = 1}$$

$$\text{Probabilities} \quad \frac{l = \{e\} \text{ for a probabilistic expression } \{e\} \quad 0 \leq e[v] \leq 1 \quad n \geq 0}{P((n, v), \circ, (n', v)) = e[v]}$$

$$\text{Nondeterminism} \quad \frac{l = n_i \text{ for } i \in \{1, 2\} \quad n \geq 0}{P((n, v), n_i, (n', v)) = 1}$$

$$\text{Terminal states} \quad \frac{n = -1}{P((n, v), \circ, \perp) = 1} \quad \frac{}{P(\perp, \circ, \perp) = 1} \quad \frac{}{P(\downarrow, \circ, \downarrow) = 1}$$

This construction employs the model's probability distribution over successor states as a way to model probabilistic choices in PGCL. Thus, for expression e , every statement of the form $\{\dots\}[e]\{\dots\}$, and valid valuation v for this statement we introduce two new successors in our models, reachable with probability $e[v]$ respectively $1 - e[v]$.

To simplify the model, only reachable states need to be considered during the construction. Note that per definition the model is, even for a simple program such as `int x := 1;`, infinite. This is a result of the program valuations being defined over the rational

numbers which are countably infinite.

To allow for easy access of valuations, we include a set of expressions $Expr$ in the construction process. The constructed PMDP has every state (l, v) labeled with every expression that is satisfied by the valuation v . As a result, we are later able to argue on valuations when using logical formulas. Notice that $Expr$ might be empty. In this case no label is assigned to any state besides \perp and ζ .

For parameterized programs, we utilize the possibility of adding parameters to the probabilistic transitions. If a probabilistic expression $\{e\}$ contains a variable which is not in $Var_{\mathbf{P}}$, we interpret this expression as a rational function, which is then defined to be the parametric probability of the transition.

To resolve the nondeterministic PGCL statements, we directly apply the concept of nondeterminism of MDPs. Constructing MDPs yields to the necessity of introducing two actions n_1 and n_2 , representing the different nondeterministic choices. The action \bigcirc indicates every deterministic transition, meaning if a \bigcirc transition is enabled for some state n , then no other action can be taken from n .

The **observe** statement is modeled using a sink state ζ which catches every run not satisfying the observed condition at some point. As stated above, those runs are excluded from our model. The remaining runs need to be normalized after such a cut. A standard normalization would require iterating over the whole model after finding such an observe, hence we rather use conditional probabilities in our model checker later on and do not give SOS rules for normalizing the model's paths directly. For more information on the approach of modeling observations as conditional probabilities, we refer to [14].

Note that for a program with no nondeterministic statements and no parameters, the construction results in a PMDP which can be transformed into a DTMC by replacing in P every transition $P(n, \bigcirc, n') = p$ with $P(n, n') = p$. Hence, we will not give a separate definition of a DTMC construction, but refer to DTMCs if the given PGCL program was parameterless and deterministic. Similarly, if no parameters are given, we consider the PMDP to be an MDP as no probabilistic parameters are needed.

The previously described SOS rules are illustrated using the known network-transmission example program \mathbf{P} of Figure 6. We present a part of the DTMC $\mathfrak{M}_{\mathbf{P}}$ in Figure 8. Note that it is in fact possible to portray the complete model since there are only finitely many reachable valuations, since p iterates in steps of 0.01 between 0 and 1 and *success* has a binary value. This yields $101 \cdot 2 = 202$ possible valuations. From this we can derive an upper boundary for the number of the model's states as there exist 12 nodes in \mathbf{P} : $|S| \leq 202 \cdot 12 = 2424$. Since it would be impractical to present the whole model, Figure 8 illustrates the part modeling the inner statements of the loop starting with the valuation $(a \mapsto 0.5, success \mapsto 0)$. Note that no MDP is given since \mathbf{P} does not contain any nondeterministic statements. The probabilistic statements are modeled using the branching of the model, where the probability depends on the current valuation of the variable a .

We can now examine the semantics of our exemplary PGCL program: We note that the

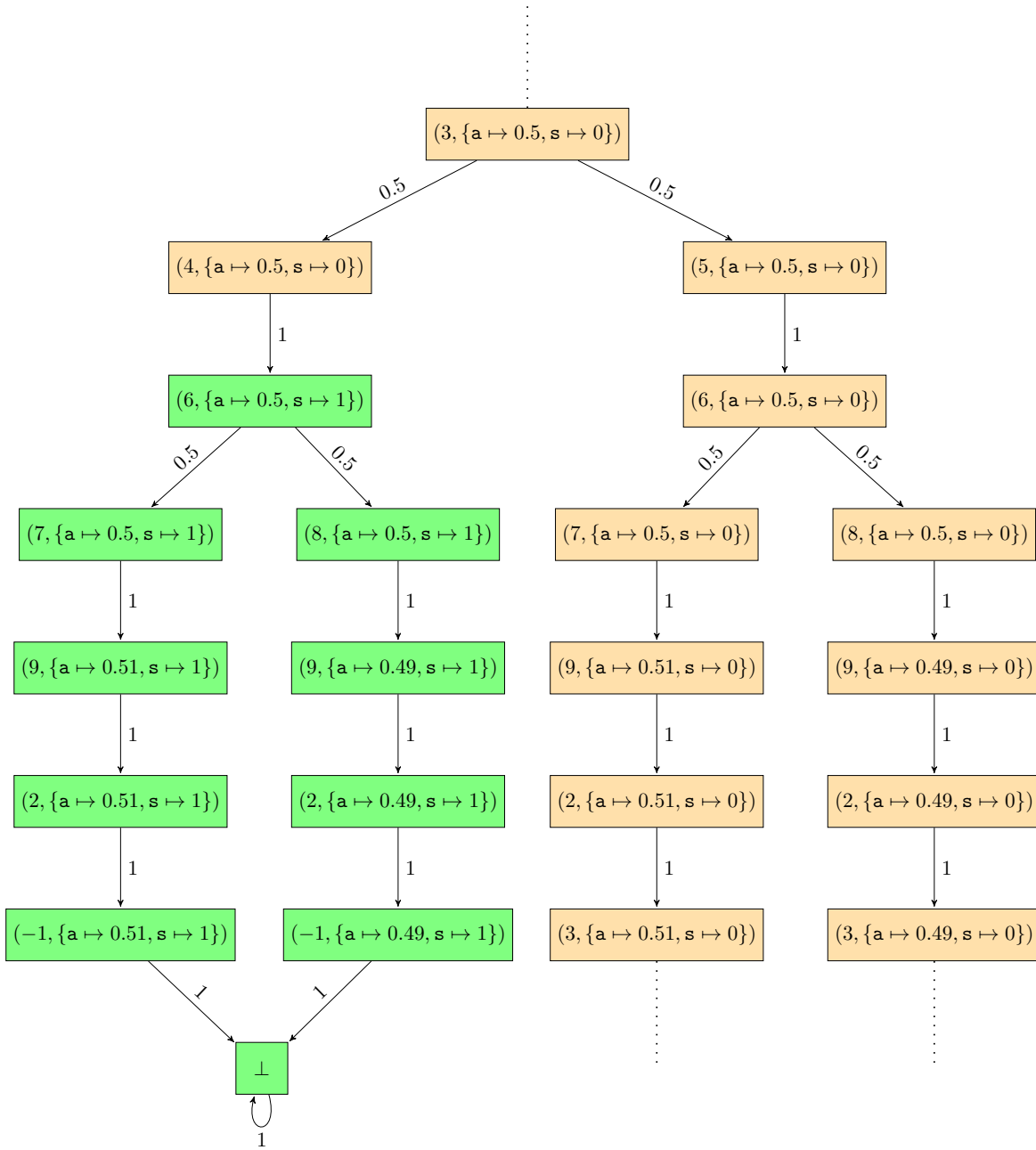


Figure 8: A part of the DTMC \mathfrak{M}_P associated with the exemplary PGCL program P of Figure 6. For practical reasons, the variable *success* is abbreviated as s . The green labeling of the states indicate that the transmission proceeded successfully.

probability a does a *random walk* on $[0, 1]$ with steps of 0.01. A random walk can be illustrated by a person doing, with some given probability, one step forward, and with the remaining probability, one step backward. Since for the probability of 0.5 a random walk is guaranteed to reach every value at some point, we conclude that our PGCL program reaches the state where $a = 1$ somewhen. Thus, the message delivery succeeds eventually. When examining the fragment of \mathfrak{M}_P one can notice that the probability of reaching that state has to be at least 0.5, which is the probability of reaching the state $(4, \{p \mapsto 0.5, s \mapsto 0\})$. But, if we take the fragment one step further, we observe that we reach the same situation again by expanding some of the not depicted nodes, leaving us with a probability of 0.25 of reaching the said state. By continuing this process, we obtain the probability of a successful message delivery to be 1. Concludingly, we are able to verify that the message transmission will happen almost-surely.

Since we did not define variables with finite domains, we employ the nondeterminism to emulate parameters with finite domains – If we wish to have an parameter $x \in \{1, 2, 3\}$ in a PGCL program P , we add the following program part to the beginning of P , whose MDP fragment is depicted in Figure 9. We will use this concept in our benchmarks later on.

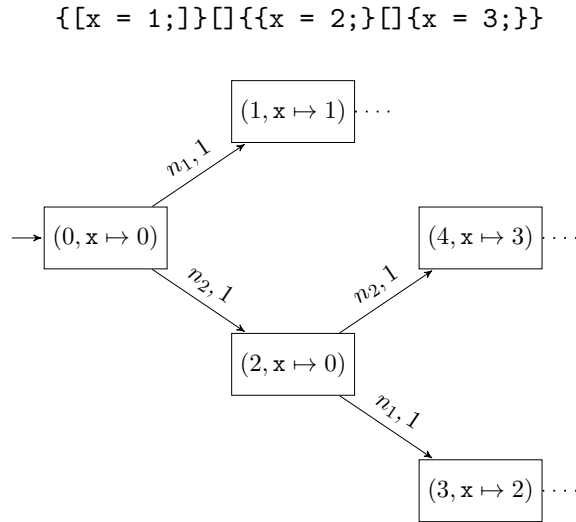


Figure 9: The initial fragment of the MDP representing the PGCL program part $\{[x = 1;]\} [] \{\{x = 2;\} [] \{x = 3;\}\}$.

Since checking the satisfaction of a property on MDPs requires to check the property for all possible schedulers, model checking on infinite MDPs is more involved as in comparison to DTMCs – There may exist infinitely many schedulers. This will lead to some constraints on the PCTL formulas that our approach is able to check, as explained in the upcoming section.

4 Iterative on-the-fly PGCL model checking

4.1 PGCL models and their properties

Since we have defined PGCL program models as well as formulas to declare properties on them, we can now mold both parts together to allow for model checking of PGCL programs.

As a way of illustration, almost-sure termination of PGCL programs is a classic example of a PCTL property of interest. Given a PGCL program P and an associated MDP \mathfrak{M}_P we can check termination by asking $\mathfrak{M}_P \models \mathbb{P}_{\{1\}}(\perp)$, since we let every terminating run end in the \perp node by definition.

As another example, consider our already known transmission-simulation program P_1 from Figure 1 on Page 1 representing a messaging system, where a sent message may not be delivered with some probability a . The success is indicated by a variable *success*, which is set to 1 if the delivery went well. We pass the set $E = \{success = 0\}$ to the probabilistic model construction, subsequently labeling every state (l, v) with $success = 0$ iff $v : success \mapsto 0$. Using the PCTL formula $\mathbb{P}_{<0.01}(Fsuccess = 0)$ we can check whether the chance of failing a delivery is less than 1%.

Both above mentioned examples are *probabilistically upper bounded properties* as in Definition 23.

Definition 31 (Satisfaction relation of PCTL formulas and PGCL programs). *For a PGCL program P and a PCTL formula φ we define $P \models \varphi$ iff for all $\sigma \in Sched(\mathfrak{M}_P)$ holds that $\mathfrak{M}_P^\sigma \models \varphi$.*

As the attentive reader might have observed, we try to check properties on infinite systems. Even for simple programs such as

```
int x := 1; while(x > 0) { x := x + 1; }
```

the resulting probabilistic model is infinite, which is indicated in Figure 10. This leads to a problem when applying standard model checking algorithms – They operate only on finite systems [8]. As a result, our approach builds finite subsystems of the model \mathfrak{M}_P until the given property is guaranteed to be violated respectively proven for the whole system.

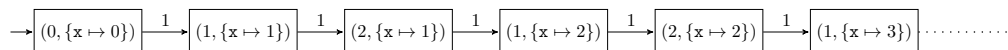


Figure 10: A part of the infinite DTMC defining the semantics of the PGCL program `int x := 1; while(x != 0) { x := x + 1; }.`

This leads to a major constraint on the PCTL formulas that we are able to check:

The model checking result of the given PCTL formula needs to be transferable from finite subsystems to the infinite PGCL program model.

This restriction is examined in the upcoming section, where we will see that we can transfer only some PCTL properties.

4.2 Transferability of properties and counterexamples

As constrained above, the model checking results – whether it be a satisfaction or a violation – need to be transferable from sub- to suprasystems. We formalize this by employing the saturation sets over PCTL path and state formulas as given in Definition 21 on Page 21.

Definition 32 (Satisfaction-transferable property). *We call any PCTL formula φ a satisfaction-transferable property iff for any MDP \mathfrak{M} , any $\sigma \in \text{Sched}_{\mathfrak{M}}$, and any finite subsystem $\mathfrak{M}' \sqsubseteq \mathfrak{M}^\sigma$*

$$\text{Sat}_{\mathfrak{M}'}(\varphi) \subseteq \text{Sat}_{\mathfrak{M}^\sigma}(\varphi)$$

holds.

Since our approach relies on the transferability of formula satisfaction from subsystems to their suprasystems, we need to ensure that we only allow formulas where we can guarantee this transfer. To apply formulas on subsystems, we need to utilize a transformation such that the novel state $*$ of the subsystem is never considered in the satisfaction set of the subformulas. The reason behind this transformation is that the state $*$ is not present in the suprasystem and will lead to some formulas being true for the subsystems but not necessarily true for the suprasystem, despite being a transferable property.

We examine this problem using the exemplary formula $\varphi = \mathbb{P}_{\geq 0.5}(\text{F}\neg a)$ and the probabilistic models depicted in Figure 11. It holds that $\mathfrak{M}_1 \models \varphi$ since $\text{Pr}_{\mathfrak{M}_1}(\{\pi \in \text{Paths}(\mathfrak{M}_1) \mid \exists i(a \notin L_1(\pi[i]))\}) = \text{Pr}_{\mathfrak{M}_1}(\text{Paths}(\mathfrak{M}_1)) = 1$. But $\mathfrak{M}_2 \not\models \varphi$ because $\text{Pr}_{\mathfrak{M}_2}(\{\pi \in \text{Paths}(\mathfrak{M}_2) \mid \exists i(a \notin L_2(\pi[i]))\}) = \text{Pr}_{\mathfrak{M}_2}(\{s_0^\omega\}) = 0$.

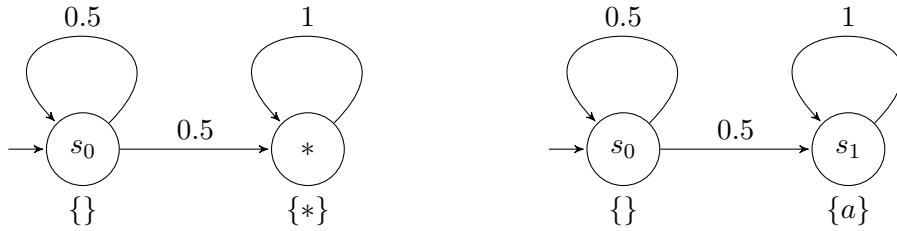


Figure 11: Two transition systems \mathfrak{M}_1 and \mathfrak{M}_2 with $\mathfrak{M}_1 \sqsubseteq \mathfrak{M}_2$ illustrating the need for a formula transformation when examining properties of subsystems.

This problem arises due to the artificial state $*$ being considered in the satisfaction set of the formula. We avoid this by adding $G\neg*$ as a path formula to the probabilistic operator \mathbb{P} . The transformed formula will then be $\varphi = \mathbb{P}_{\geq 0.5}(\mathbf{F}\neg a \wedge \mathbf{G}\neg*)$. It should be clear that this formula does not allow the consideration of any run visiting the state $*$. Hence, after the transformation $\mathfrak{M}_1 \not\models \varphi$ since $Pr(\pi \in Paths(\mathfrak{M}_1) \mid \exists i(a \notin L_1(\pi[i])) \wedge \forall j(* \notin L(\pi[j]))) = Pr(\{s_0^\omega\}) = 0$.

Thus, we generally transform a given formula φ where $*$ $\notin Sub(\varphi)$ as follows: For every atomic proposition a and formula $\neg a \in Sub(\varphi)$ we edit $\neg a$ to be $\neg a \wedge \neg*$. We additionally edit every non-negated atomic proposition $a \in Sub(\varphi)$ to $a \wedge \neg*$.

Claim 1. *Let \mathfrak{M} be an MDP. Consider \mathfrak{M}' a DTMC such that $\mathfrak{M}' \sqsubseteq \mathfrak{M}^\sigma$ for an arbitrary $\sigma \in Sched_{\mathfrak{M}}$ and $\mathbb{P}_{\succ i}(\varphi)$ a PCTL state formula such that φ is a transferable property. We apply the above mentioned transformation on φ .*

It then follows that $\mathbb{P}_{\succ i}(\varphi)$ is also a transferable property, i.e. $Sat_{\mathfrak{M}'}(\mathbb{P}_{\succ i}(\varphi)) \subseteq Sat_{\mathfrak{M}^\sigma}(\mathbb{P}_{\succ i}(\varphi))$.

Recall that we defined $\succ \in \{>, \geq\}$ and $\prec \in \{<, \leq\}$, $\bar{\prec} = \geq$, $\bar{\succ} = >$, $\bar{\succ} = \leq$, and $\bar{\prec} = <$.

Proof. We prove the statement of Claim 1 by contraposition. We show that for all schedulers $\sigma \in Sched_{\mathfrak{M}}$ and the DTMCs \mathfrak{M}^σ , \mathfrak{M}' and the formula ψ as given in the claim

$$\overline{Sat_{\mathfrak{M}^\sigma}(\mathbb{P}_{\succ i}(\varphi))} \subseteq \overline{Sat_{\mathfrak{M}'}(\mathbb{P}_{\succ i}(\varphi))}$$

holds. Thus, we prove that the inclusion is reversed for the complement of both sets, which directly implies that the claimed inclusion for the original sets holds.

Let $s \in \overline{Sat_{\mathfrak{M}^\sigma}(\mathbb{P}_{\succ i}(\varphi))}$. Then, by definition of $Sat_{\mathfrak{M}^\sigma}(\cdot)$, $s \not\models \mathbb{P}_{\succ i}(\varphi)$ holds. By definition of the satisfaction relation it follows that $Pr_{\mathfrak{M}^\sigma}(\{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\}) \bar{\succ} i$.

We notice that $Paths(\mathfrak{M}', x) \setminus \{\pi \in Paths(\mathfrak{M}^\sigma, x) \mid \exists i(* \in L'(\pi[i]))\} \subseteq (Paths_{\mathfrak{M}^\sigma}, x)$ holds obviously for all $x \in S$.

From this observation and the assumption $Sat_{\mathfrak{M}'}(\varphi) \subseteq Sat_{\mathfrak{M}^\sigma}(\varphi)$ it follows that $\{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\} \subseteq \{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\}$. From stochastic theory we know $A \subseteq B \implies P(A) \leq P(B)$ for every probability measure P .

Thus, we conclude $Pr_{\mathfrak{M}^\sigma}(\{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\}) \leq Pr_{\mathfrak{M}^\sigma}(\{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\}) \bar{\succ} i$.

To transition from $Pr_{\mathfrak{M}^\sigma}$ to $Pr_{\mathfrak{M}'}$ we observe that for every $X \in \Sigma_{\mathfrak{M}'}$ where for all $\pi \in X$ holds that $\forall i(* \notin L'(\pi[i]))$ it holds that $Pr_{\mathfrak{M}'}(X) = Pr_{\mathfrak{M}^\sigma}(X)$. Since the formula transformation as stated in the introduction to Claim 1 does never allow to arise paths where $*$ is visited, we conclude that $Pr_{\mathfrak{M}'}(\{\pi \in Paths(\mathfrak{M}^\sigma, s) \mid \pi \models \varphi\}) \bar{\succ} i$, hence $s \in \overline{Sat_{\mathfrak{M}'}(\mathbb{P}_{\succ i}(\varphi))}$.

By contraposition it follows that $Sat_{\mathfrak{M}'}(\mathbb{P}_{\succ i}(\varphi)) \subseteq Sat_{\mathfrak{M}^\sigma}(\mathbb{P}_{\succ i}(\varphi))$. □

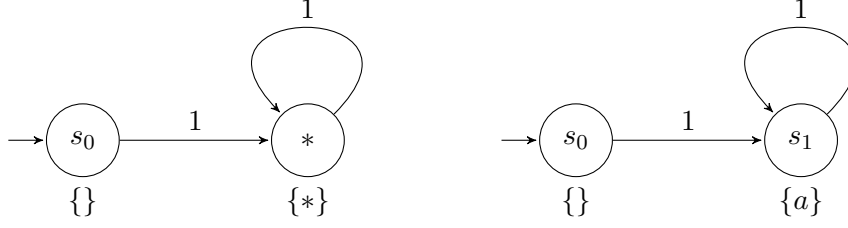


Figure 12: Two example MDPs \mathfrak{M}_1 and \mathfrak{M}_2 with $\mathfrak{M}_1 \sqsubseteq \mathfrak{M}_2$ showing that $\mathbb{P}_{<1}(\mathbf{F}a)$ is a non-transferable property.

As a special case of Claim 1 it follows that *probabilistically lower bounded reachability properties* – properties of the form $\mathbb{P}_{>i}(\mathbf{F}\varphi)$ where φ consists only propositional logic formulas over the atomic propositions – are transferable. For this, we have to show that $Sat_{\mathfrak{M}'}(\mathbf{F}\varphi \wedge \mathbf{G}\neg*) \subseteq Sat_{\mathfrak{M}}(\mathbf{F}\varphi)$. But since $Sat_{\mathfrak{M}'}(\mathbf{F}\varphi \wedge \mathbf{G}\neg*) = \{\pi \in Paths(\mathfrak{M}', s) \mid \text{for } s \in S' \text{ and } \exists i(\pi[i] \models \varphi) \text{ and } \nexists i(* \in L(\pi[i]))\}$, it follows that $Sat_{\mathfrak{M}'}(\mathbf{F}\varphi \wedge \mathbf{G}\neg*) \subseteq \{\pi \in Paths(\mathfrak{M}, s) \mid \text{for } s \in S \text{ and } \exists i(\pi[i] \models \varphi)\} = Sat_{\mathfrak{M}}(\mathbf{F}\varphi)$. Hence $\mathbb{P}_{>i}(\mathbf{F}\varphi)$ is a satisfaction-transferable property.

One should notice that the implication of Claim 1 does not hold for formulas of the type $\mathbb{P}_{<i}(\psi)$, i.e. $\mathbb{P}_{<i}(\psi)$ is *not* transferable in general. This can be seen by the counterexample depicted in Figure 12. Here, $\mathfrak{M}_1 \models \mathbb{P}_{<1}(\mathbf{F}a)$, since there exists no state labeled with a . But the suprasystem \mathfrak{M}_2 satisfies the formula because the probability of visiting the state s_1 is 1. The subformula of the probabilistic operator is in fact again a transferable property since $Sat_{\mathfrak{M}_1}(\mathbf{F}a) = \emptyset \subseteq Sat_{\mathfrak{M}_2}(\mathbf{F}a) = \{s_0 s_1^\omega\}$. Using this counterexample we conclude that probabilistically lower bounded properties are not transferable in general, even if their subformulas are transferable.

But on the bright side we are able to transfer violations of any probabilistically lower bounded formula. Counterexamples have been introduced in Section 2.5. In the upcoming we will show that the transferability of satisfaction as in Definition 32 can be applied to counterexamples in a reversed fashion.

Definition 33 (Violation-transferable property). *Let φ be a PCTL formula, \mathfrak{M} an MDP and $\sigma \in Sched_{\mathfrak{M}}$ such that $\mathfrak{M}^\sigma \not\models \varphi$ holds, where C_φ is a counterexample. We call φ a violation-transferable property iff for any subsystem $\mathfrak{M}' \sqsubseteq \mathfrak{M}^\sigma$*

$$\overline{Sat_{\mathfrak{M}'}(\varphi)} \subseteq \overline{Sat_{\mathfrak{M}}(\varphi)}$$

holds. We additionally call the counterexample C_φ transferable from \mathfrak{M}' to \mathfrak{M} respectively \mathfrak{M}^σ .

Thus, we call a property violation-transferable if we are able to conclude that all states in the subsystem from which a violation of φ arises are also witnesses of the violation in the suprasystem.

Since the transferability of counterexamples is just the application of the transferability of negated formulas, we can easily check for those as well. As Claim 1 states, we are able to transfer satisfaction of properties of the form $\mathbb{P}_{>i}(\psi)$. Since $\mathbb{P}_{>i}(\psi) \equiv \neg\mathbb{P}_{<i}(\psi)$ holds for DTMCs formulas of the form $\mathbb{P}_{<i}(\psi)$ are violation-transferable properties iff ψ is satisfaction-transferable.

Table 1 presents a straightforward overview of the transferability rules for MDPs, where \checkmark indicates that the inherent implication holds.

Table 1: A summary of the transferability of properties for some PCTL formulas. Here, \mathfrak{M} is an MDP and $\mathfrak{M}' \sqsubseteq \mathfrak{M}^\sigma$ for an arbitrary $\sigma \in \text{Sched}_{\mathfrak{M}}$. ψ is a satisfaction-transferable PCTL path formula. \checkmark depicts that the inherent implication holds, whereas \times says that the implication does not hold.

\implies	$\mathfrak{M} \models \mathbb{P}_{<i}(\psi)$	$\mathfrak{M} \models \mathbb{P}_{>i}(\psi)$	$\mathfrak{M} \not\models \mathbb{P}_{<i}(\psi)$	$\mathfrak{M} \not\models \mathbb{P}_{>i}(\psi)$
$\mathfrak{M}' \models \mathbb{P}_{<i}(\psi)$	\times	\times	\times	\times
$\mathfrak{M}' \models \mathbb{P}_{>i}(\psi)$	\times	\times	\times	\times
$\mathfrak{M}' \not\models \mathbb{P}_{<i}(\psi)$	\times	\times	\checkmark	\times
$\mathfrak{M}' \not\models \mathbb{P}_{>i}(\psi)$	\times	\times	\times	\times

As one can derive from Table 1, iterative model checking for MDPs is quite ineffectual, since just one implication can be used: $\mathfrak{M}' \not\models \mathbb{P}_{<i}(\psi) \implies \mathfrak{M} \not\models \mathbb{P}_{<i}(\psi)$. If we restrict our approach to DTMCs, a greater rate of transferability arises due to the absence of schedulers and the universal quantifier in the satisfaction relation. The transferability matrix for DTMCs is depicted in Table 2.

Table 2: A summary of the transferability of properties for some PCTL formulas. Here, \mathfrak{M} is a DTMC and $\mathfrak{M}' \sqsubseteq \mathfrak{M}$. ψ is a satisfaction-transferable PCTL path formula. \checkmark depicts that the inherent implication holds, whereas \times says that the implication does not hold.

\implies	$\mathfrak{M} \models \mathbb{P}_{<i}(\psi)$	$\mathfrak{M} \models \mathbb{P}_{>i}(\psi)$	$\mathfrak{M} \not\models \mathbb{P}_{<i}(\psi)$	$\mathfrak{M} \not\models \mathbb{P}_{>i}(\psi)$
$\mathfrak{M}' \models \mathbb{P}_{<i}(\psi)$	\times	\times	\times	\times
$\mathfrak{M}' \models \mathbb{P}_{>i}(\psi)$	\times	\checkmark	\checkmark	\times
$\mathfrak{M}' \not\models \mathbb{P}_{<i}(\psi)$	\times	\checkmark	\checkmark	\times
$\mathfrak{M}' \not\models \mathbb{P}_{>i}(\psi)$	\times	\times	\times	\times

In the following, we give some examples on satisfaction-transferable PCTL formulas for DTMCs.

- $\mathbb{P}_{\geq 0.5}(Fa)$
- $\mathbb{P}_{=1}(G\neg b)$

- $\mathbb{P}_{\geq 0.1}(aU^{\leq 5}b)$

In particular, every probabilistically upper bounded reachability property on DTMCs is satisfaction-transferable as $Sat_{\mathfrak{M}'}(Fa) \subseteq Sat_{\mathfrak{M}}(Fa)$ for every DTMC \mathfrak{M} and $\mathfrak{M}' \sqsubseteq \mathfrak{M}$ holds obviously.

Additionally, as Table 1 shows, we can transfer violations for MDPs. Some examples on violation-transferable properties for MDPs are given in the following.

- $\mathbb{P}_{\leq 0.9}(Fa)$
- $\mathbb{P}_{< 0.01}(G\neg b)$

Since reachability properties are widely present in practical applications of model checking algorithms, we note a handy utilization of our presented approach on PGCL model checking.

4.3 Handling the observe statement

As mentioned in earlier sections of this paper, an **observe** statement requires a normalization of the remaining runs satisfying the observed condition [14]. Instead of iterating over the whole model after such an observe violation was found, we rather employ the model checker for this purpose: If the PGCL program contains an observe statement, we transform the given PCTL state formula into a *conditional* PCTL state formula which allows for stochastic conditions. Since the goal is to disregard every run visiting a ζ labeled node somewhen, we transform a given formula $\varphi = \mathbb{P}_{\bowtie i}(\psi)$ to $\varphi' = \mathbb{P}_{\bowtie i}(\psi \mid G\neg\zeta)$, where $\mathbb{P}_{\bowtie i}(\psi_1 \mid \psi_2)$ denotes the *conditional probabilistic operator*. It is defined to be true iff $Pr^{\mathfrak{M}}(\psi_1 \mid \psi_2) \bowtie i$, hence iff the probability of satisfying ψ_1 under the condition that ψ_2 is satisfied is $\bowtie i$.

We already adapted the model building process to construct models that lead every observe-violating run to the ζ node. This model building process for PGCL programs containing **observe** statements can be examined by means of the example of Figure 13. We consider the simple nondeterministic PGCL program

```
P = int x := 0; {{x := 5;} [] {x := 2;}} [q] {x := 2;} observe(x>3);
```

and its corresponding MDP \mathfrak{M}_P , which the model building process will yield.

Every PGCL model checking request will result in a non-consideration of the second probabilistic choice going to the node $(2, \{x \mapsto 2\})$. This results in a normalization of the remaining runs such that the probability of reaching $(1, \{x \mapsto 0\})$ will be 1, in contrast to the probability of 0.5 if we do not employ the above-mentioned conditional PCTL state formula.

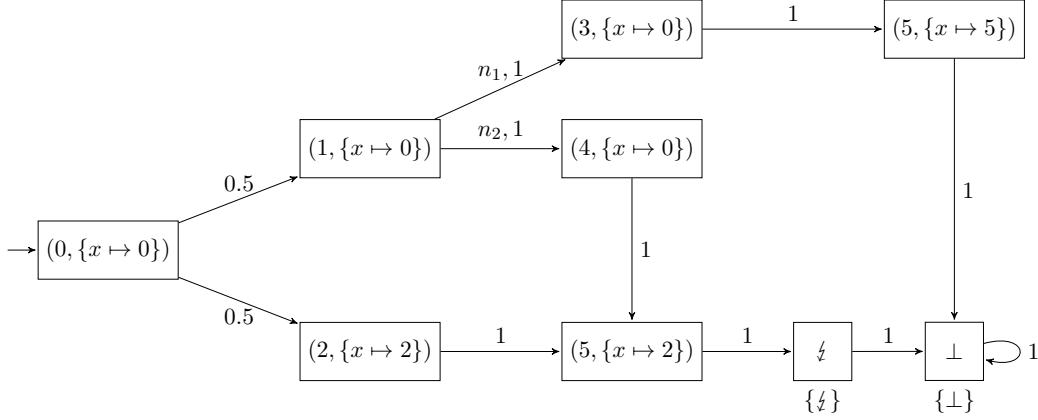


Figure 13: The MDP \mathfrak{M}_P of the PGCL program $P = \{\text{int } x := 0; \{x := 5;\} \square \{x := 2;\} \text{ [q] } \{x := 2;\} \text{ observe}(x > 3);\}$.

4.4 Iterative PGCL model building and checking

For the sake of simplicity, we assume in the following that a *model checker* for finite DTMCs, MDPs, and PMDPs exist. Such a model checker answers the question

For a given finite probabilistic model \mathfrak{M} and a PCTL formula φ does $\mathfrak{M} \models \varphi$ hold?

A result can either be a positive answer or a counterexample showing where the violation occurred. We again refer to [8] for further information on how to implement such a model checker for finite probabilistic systems. For implementing a model checker on PMDPs, we refer to [9].

Since we have proven in Section 4.2 that for some PCTL properties it is in fact possible to transfer them from finite subsystems to the per Definition 30 infinite PGCL models, we are able to iteratively construct finite models until some transferable property is proven or violated. As a rough overview, we apply the SOS rules of Definition 30 on Page 19 step-by-step until we decide that we have constructed enough states and risk the application of a model checker. We fetch the result and decide if we need to continue building states or if we can return a definite answer.

Algorithm 1 presents the construction algorithm for PGCL programs. It is parameterized with the program graph and the PCTL formula and additionally takes a step width which declares after how many steps the model checker shall be called. *borderStates* is a priority queue containing states of the model \mathfrak{M} . A model's state has the form (l, v) where l is a location number of P and v a valuation over $Val(Var_P)$. We assume that the function *modelType*(G_P) returns the fitting model type of the program which can be either a DTMC, an MDP, or a PMDP. As stated above, we expand the state space

step-by-step. For this, a set of *border states* is kept in the memory – It records all states that are not yet expanded. In each algorithm iteration we dequeue one border state and expand it. After the expansion of such a border state, we remove it from the set since it is fully processed now. The newly constructed states for this iteration – Between zero and two in quantity, saved in the *newStates* set – are added to the set of border states. Note that for any expanded state the set of border states grows by two in the worst case which leads to a linear growth of the non-expanded states for each step.

After expanding the model and the given step width is reached, we run the model checker on \mathfrak{M}' which is equivalent to the constructed model \mathfrak{M} beside adding a transition from every unexpanded state to the $*$ node with probability one. If the result is transferable, i.e. conforms to the transferability matrix depicted in Table 1 respectively Table 2, we return the result. Otherwise we continue to expand the model \mathfrak{M} .

Algorithm 1: ITERATIVEPGCLMODELCHICKER

Input: A PGCL program graph G_P , a set of expressions $Expr$, a PCTL formula φ , and an integer $stepWidth$.

Result: *True* if $\mathfrak{M}_P \models \varphi$, *false* and a counterexample otherwise. May not terminate, i.e. is not complete.

```

1 Integer step := 0
2 modelType( $G_P$ )  $\mathfrak{M} := (\{\perp, \zeta\}, \{(0, \{x \mapsto 0 \mid \forall x \in Var_P\})\}, \{(\perp, \perp) \mapsto 1, (\zeta, \zeta) \mapsto 1\}, \{\circ, n_1, n_2\}, Expr, \{\perp \mapsto \perp, \zeta \mapsto \zeta\})$ 
3
4 PriorityQueue borderStates :=  $\emptyset$ 
5 while !empty(borderStates) do
6   State currentState := dequeue(borderStates)
7   Set newStates := expand(currentState,  $\mathfrak{M}$ , Expr,  $G_P$ )
8   for newState in newStates do
9     enqueue(newState, borderStates)
10  step := step + 1
11  if step  $\geq$  stepWidth then
12     $\mathfrak{M}' := \mathfrak{M}$ 
13     $P' := P \cup \{(s, \circ, *) \mapsto 1 \mid \forall s \in borderStates\}$ 
14    result := modelCheck( $\mathfrak{M}'$ ,  $\varphi$ )
15    if isTransferable( $\varphi$ , modelType( $G_P$ ), result) then
16      return result

```

The presented implementation relies on a priority queue for the border states and the initial states to allow for the heuristical approach mentioned in the introduction. This results in the need for a partial order over model states. As most implementations of data structures call for a strict total order over their elements, we rather refer to such an ordering for the upcoming parts. For now we will just assume that such an ordering exists and not explicate how it is defined. The specific heuristics and the reasons of utilizing

a priority queue are discussed in Section 5. For now we suppose that the priority queue is degenerated to standard non-prioritizing queue.

An implementation of the functions *isTransferable* and *expand* can be found in the appendix of this paper. When expanding a state, the newly created states are labeled according to the given set of expressions *Expr*. The *expand* function additionally handles probabilistic parameters: If an expression containing an unknown variable (i.e. a parameter) appears, we add this parameter to the probabilistic transition function of \mathfrak{M} .

We will examine the functioning of Algorithm 3 by means of an example. Figure 14 presents five iterations on our exemplary program P of Figure 6 on Page 15. We already presented a part of the model \mathfrak{M}_P in Figure 7. We again consider this fragment, but construct it using the previously presented Algorithm 1 with a step width of 5, the set of expressions $Expr = \{s = 1\}$, and the PCTL formula $\mathbb{P}_{\geq 0.5}(F(s = 1))$.

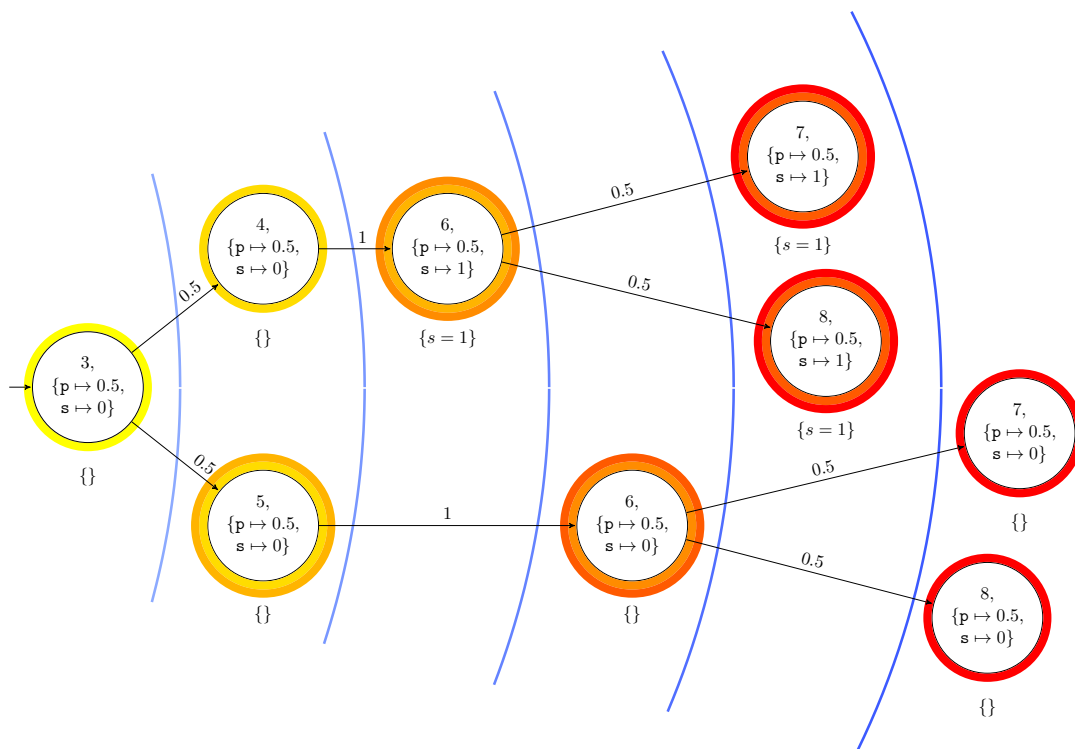


Figure 14: Five iterations of Algorithm 1 on the exemplary program P of Figure 6 starting with the node $(3, \{p \mapsto 0.5, s \mapsto 0\})$.

We expand the model's states step-by-step, which is indicated by the blue bordered lines in Figure 14 – Every state between two lines is created in the same step. We symbolize the current border states by assigning the same border color to every state in the queue of border states.

We start with the priority queue – denoted by ζ – containing only the state $s_1 = (3, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$. Note that the satisfaction is not influenced by starting the algorithm with s_1 since s_1 is reachable with probability 1 from the initial state $(0, \{\mathbf{p} \mapsto 0, \mathbf{s} \mapsto 0\})$. Algorithm 1 is simulated stepwise:

- At first, we expand the only state s_1 which yields two new states $s_2 = (4, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$ and $s_3 = (5, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$. We update $\zeta = (s_2, s_3)$.
- We dequeue s_2 and obtain $s_4 = (6, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 1\})$ and update $\zeta = (s_3, s_4)$.
- We dequeue s_3 . The expansion yields $s_5 = (6, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$ and $\zeta = (s_4, s_5)$.
- We dequeue s_4 . The expansion yields $s_6 = (7, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 1\})$ and $s_7 = (8, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 1\})$. We update $\zeta = (s_5, s_6, s_7)$.
- We dequeue s_5 . The expansion yields $s_8 = (7, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$ and $s_9 = (8, \{\mathbf{p} \mapsto 0.5, \mathbf{s} \mapsto 0\})$. We update $\zeta = (s_6, s_7, s_8, s_9)$.
- The given step width is reached as we expanded the model five times. We run the model checker for the formula $\mathbb{P}_{\geq 0.5}(\mathbf{F}a)$ and the model \mathfrak{M} presented in Figure 14.
- Since the given property holds, we terminate and return a positive model checking result.

Note that by using this iterative construction approach we cannot verify $\mathbb{P}_{=1}(\mathbf{F}(s = 1))$ since we can never construct the full model. Every construction step will only raise the probability of transmission success by a fraction of the previous probability. This reveals the limitations of our approach as proving properties that require stochastic arguing over the infinite model can not be done.

4.5 Soundness and completeness

As mentioned earlier, the presented Algorithm 1 on Page 30 is *not complete*, i.e. will run forever on certain inputs. On the other hand, we can present a *soundness* result by employing Claim 1 on Page 25.

Theorem 1 (Incompleteness). *Algorithm 1 on Page 30 is incomplete, i.e. does not return an answer for all possible inputs.*

Proof. Consider the PGCL program $P = \text{int } x := 1; \text{ while}(x > 0) \{x := x + 1;\}$ and the PCTL formula $\varphi = \mathbb{P}_{=1}(\mathbf{F}end)$. The DTMC \mathfrak{M}_P is depicted in Figure 10 on Page 10. Algorithm 1 builds this model iteratively, i.e. constructs in every iteration a fixed number of nodes concatenated in a linear fashion. The node \perp will never be reachable in the system as every node $(1, \{x \mapsto i\})$ has only one successor, namely $(2, \{x \mapsto i\})$ which again has its only successor $(1, \{x \mapsto i + 1\})$ for $i \in \mathbb{N} \setminus \{0\}$. This pattern is executed to infinity, as P does not terminate. Hence, $\mathfrak{M}_P \not\models \varphi$. Algorithm 1 will not terminate since for all $\mathfrak{M}' \sqsubseteq \mathfrak{M}_P$ it holds that $\mathfrak{M}' \not\models \varphi$, and according to Table 2

on Page 27 we can not transfer this result to \mathfrak{M}_P . Hence, the algorithm will not return an answer on this input. \square

On the other hand, we are able to give a soundness result for the presented algorithm. The proof follows directly from the already proven Claim 1 on Page 25, where it was shown that we can in fact transfer some properties from finite subsystems to infinite suprasystem.

Theorem 2 (Soundness). *Algorithm 1 on Page 30 is sound, i.e. if it returns an answer, it returns a correct one. Hence, for each PGCL program P and PCTL state formula φ , if Algorithm 1 returns yes, then $\varphi \models \mathfrak{M}_P$, and if the answer is no, then $\varphi \not\models \mathfrak{M}_P$.*

We divide the proof in two parts: Firstly, we show that the building algorithm in fact produces valid subsystems of \mathfrak{M}_P , and secondly, we show that if we return a result over the subsystem, it is a correct output for the given program.

Proof. Let P be PGCL program and φ a PCTL state formula.

Part 1. We show that for every iteration of Algorithm 1 it holds that the constructed model \mathfrak{M}' is a subsystem of \mathfrak{M}_P , i.e. $\mathfrak{M}' \sqsubseteq \mathfrak{M}_P$. As the *expand* function is a one-to-one adaption of the SOS rules for PGCL models in Definition 30, every step adds zero to two new states to the model and updates the transition function in a correct manner. Those new states are yet to be expanded and have no successors at that moment. Every non-expanded state, and no other state, will be saved in the set of border states, which implies that every state which is going to be expanded somewhen is a state of the model \mathfrak{M}_P . Since \mathfrak{M}' additionally adds a transition from every unexpanded state to $*$, it follows that $\mathfrak{M}' \sqsubseteq \mathfrak{M}_P$.

Part 2. We show that for every returned result of the form $\mathfrak{M}' \models \varphi$ respectively $\mathfrak{M}' \not\models \varphi$ it follows that $\mathfrak{M}_P \models \varphi$ respectively $\mathfrak{M}_P \not\models \varphi$. From Claim 1 we already know that if φ is a transferable property, it follows that for every $\mathfrak{M}' \sqsubseteq \mathfrak{M}_P$ where $\mathfrak{M}' \models \varphi$ that $\mathfrak{M}_P \models \varphi$ and equivalently in case $\mathfrak{M}' \not\models \varphi$. As Algorithm 1 implements the transferability check correctly, and we only return an answer if the result is transferable, it directly follows that every returned answer is correct. \square

5 Heuristics

The main idea behind exploring the state space in an iterative manner and performing on-the-fly model checking leads to the opportunity of employing a *heuristic approach*. It seems reasonable to explore more useful paths earlier in the model construction process – Where the term “usefulness” heavily depends on the given program and formula. On the other hand, we shall degrade paths which will probably not lead to a violation respectively satisfaction of the formula in a reasonable amount of time.

To get a grasp on how usefulness can be interpreted, we examine a reachability property as an example. Consider $\varphi = \mathbb{P}_{\geq 0.5}(Fa)$. Our goal is to find a finite path with its probability greater or equal to 0.5 which leads to an a labeled state. The initial part of the infinite model we wish to check this property on is depicted in Figure 15.

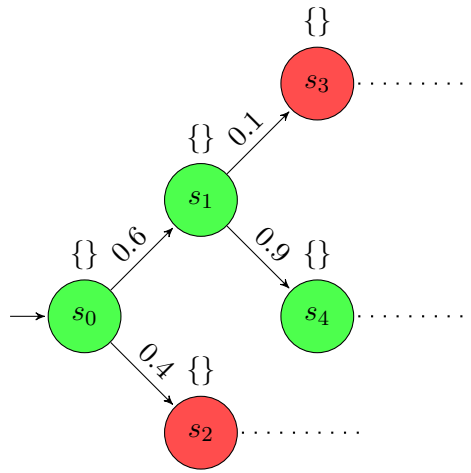


Figure 15: A part of a DTMC clarifying the reasonableness of a heuristical approach. For any probabilistically lower bounded property with $p \geq 0.5$ the red states do not need to be considered in the expansion process at the current model state.

Intuitively, all red tagged states are useless for a further exploration at that current state of the model since the probability to reach them is less than 0.5, namely 0.4 for s_2 and 0.3 for s_3 . This implies that it does not make any sense to expand the states s_2 and s_3 at the moment. On the other hand, it will be useful to explore the state s_4 since the probability of reaching s_4 is 0.54 at that particular model expansion state. Note that it is always possible to find another path leading to s_3 with a probability greater or equal to 0.5. We then need to reconsider s_3 in the expansion process.

The above shown observation of some border states being *useful* whereas other border states being *useless* in the current situation can be exploited for a heuristical approach. We will later present a heuristic which implements this idea and examine how to calculate

the probability of any state.

Besides the harsh non-consideration of states in the expansion process, we can try to prefer some border states over others: If a state promises to lead into a more probable path, it should be explored earlier than a state whose probability is scraping at the given boundary of the formula. Algorithm 1, introduced in Section 4.4, uses a priority queue for which we firstly assumed to be a standard non-prioritizing queue. In this section, we will employ this priority queue to implement the heuristical approach since the best rated state will always be on the first place of the queue. As we always dequeue from the top, the algorithm explores more promising states earlier.

Section 4.4 mentions a strict total order on model states for the priority queue to function. A priority queue is defined over a structure $(A, <)$ where A represents possible objects going into the queue. In our case the queue contains model states, hence we need to consider the structure $(\{(l, v) \mid l \in Loc_P, v \in Val(Var_P)\}, <)$ for a PGCL program P . It therefore remains to define the strict total order relation $<$ over model states. The concrete heuristic used in the algorithm implements such an order. In this section, we will introduce various orders on model states and examine those on their complexity and usefulness.

Subsequently, we will define a heuristic for a PGCL program P and a given MDP $\mathfrak{M} \sqsubseteq \mathfrak{M}_P$ as a relation $<_{\mathfrak{M}} \subseteq \{(l, v) \mid l \in Loc_P, v \in Val(Var_P)\} \times \{(l, v) \mid l \in Loc_P, v \in Val(Var_P)\}$. Each heuristic depends on the current construction status of the model \mathfrak{M}_P . We constraint $<_{\mathfrak{M}}$ to the typical restrictions of a strict total order, where \otimes denotes the exclusive or operator:

- $x <_{\mathfrak{M}} y \wedge y <_{\mathfrak{M}} z \implies x <_{\mathfrak{M}} z$ (Transitivity)
- $x <_{\mathfrak{M}} y \otimes y <_{\mathfrak{M}} x \otimes x = y$ (Trichotomy)

All following heuristics will use a *rating function* to rate the usefulness of a state. For some probabilistic model \mathfrak{M} a rating function is defined as $r_{\mathfrak{M}}: \{(l, v) \mid l \in Loc_P, v \in Val(Var_P)\} \mapsto \mathbb{Q}$, hence we apply a rational score to each state. We can use this rating function to define a heuristic $<_{\mathfrak{M}}$ for two states $s_1 = (l_1, v_1)$ and $s_2 = (l_2, v_2)$:

$$r_{\mathfrak{M}}(s_1) < r_{\mathfrak{M}}(s_2) \implies s_1 <_{\mathfrak{M}} s_2$$

$$r_{\mathfrak{M}}(s_1) = r_{\mathfrak{M}}(s_2) \wedge (l_1 < l_2 \vee (l_1 = l_2 \wedge v_1 < v_2)) \implies s_1 <_{\mathfrak{M}} s_2$$

Hence, this definition employs the rating function and takes the above mentioned restrictions for strict total orders into account: If the ratings are in a strict total order relation, the states are too. In case that both ratings are the same we use either the location numbers or the valuations of the states to define the strict total order since every state has a unique combination of a location number and valuation. We consider $v_1 < v_2$ for two valuations v_1 and v_2 iff $v_1(x) < v_2(x)$ where x is the first variable where v_1 and v_2 differ if we sort the variables in an alphabetical order.

Using a rating function allows to combine various heuristic approaches, e.g. by summing their ratings up, and retrieving a new score which may be more significant or advantageous to the construction process.

Since we are interested in both probabilistically lower and upper bounded properties, heuristics have to differ for each. If we are given a probabilistically upper bounded formula of the form $\mathbb{P}_{\leq i}(\psi)$ we have no option but to find a counterexample in the given model due to the transferability of that property (see Table 2). This leads to the task of finding a set of paths starting in the initial state with their probability being $\overleftarrow{\leq} i$. As a result, our heuristic has to *minimize* the probability of the explored paths.

If we consider probabilistically lower bounded formula of the form $\mathbb{P}_{\geq i}(\psi)$ a counterexample can never be found using our approach (see Table 2). Hence, we need to verify the given formula by finding a set of paths starting in the initial node with their probability being $\overrightarrow{\geq} i$. This leads to the reasonable approach of *maximizing* the probabilities along the building process. For the following parts only heuristics for maximizing paths probabilities will be given. If we want to minimize those probabilities, one can easily reverse the priority queue of Algorithm 1. Then the smallest element will be on the first place of the queue which leads to expanding the states with lower probabilities earlier.

For the following we will divide possible heuristics into two groups to allow for an easier understanding of their complexities:

1. Local heuristics
2. Global heuristics

A *local heuristic* will only use information locally available for the two given states and is not allowed to iterate over the whole model. This means that its runtime does not depend on the current size of the model, but its complexity is rather determined by the state and its neighbored nodes itself. Note that local heuristics are not able to implement the truncating approach mentioned in the introduction: To know the exact maximal probability of a state to be reached from the initial state one needs to iterate over the whole model after each building iteration.

A *global heuristic* on the other hand is allowed to iterate multiple times over the whole model and can gather every bit of information it may desires. For this case, the complexity can be arbitrary and even exponential – Although it should be desirable to keep the runtime as low as possible.

5.1 Local heuristics

Local heuristics provide, as a general overview, not as good results as global heuristics due to the lack of information available to the function. On the other hand, they run in a reasonable amount of time – This feature is very critical since the priority queue compares stored elements during an insertion and deletion.

Heuristic of incoming transition probabilities. As a naive approach, we can try

to compare states using the transitions from their direct predecessors. The sums of the probabilities of the ingoing transitions give a good approximation on how probable it is to reach a given state. For this we consider the multiset of incoming probabilities $InProb$ for a state s of an MDP \mathfrak{M} which is defined as follows:

$$InProb(s, \mathfrak{M}) = \{p \in [0, 1] \mid \exists s' \in S \exists \alpha \in Act(P(s', \alpha, s) = p)\}$$

Note that $InProb$ is a multiset, hence multiples elements of the same value may exist in this set. On this basis we rate a state by forming the sum of the incoming probabilities.

$$r_{\mathfrak{M}}(s) = \sum_{p \in InProb(s, \mathfrak{M})} p$$

The functioning of this heuristic is best examined using an example. Figure 16 presents two states s_1 and s_2 as well as their predecessors for an exemplary model \mathfrak{M} . At first we determine the multisets of incoming probabilities $InProb(s_1) = \{0.25, 0.3, 0.3, 0.9\}$ and $InProb(s_2) = \{1, 0.9, 0.2\}$. Since $r_{\mathfrak{M}}(s_1) = 1.75 < r_{\mathfrak{M}}(s_2) = 2.1$, we conclude that $s_1 <_{\mathfrak{M}} s_2$ and $s_2 \not<_{\mathfrak{M}} s_1$.

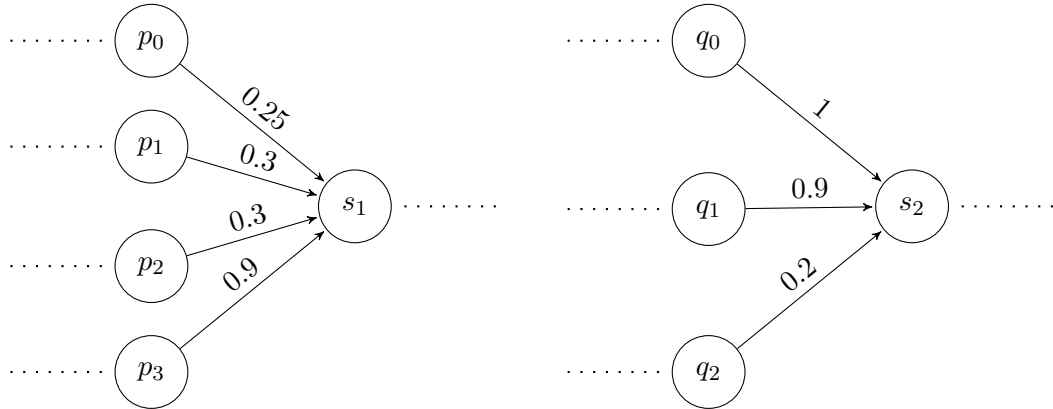


Figure 16: Two states s_1 and s_2 as well as their predecessors for an exemplary model \mathfrak{M} . For the heuristic of incoming transition probabilities it follows that $s_1 <_{\mathfrak{M}} s_2$. The state labeling is omitted for readability.

Concludingly, the algorithm prefers s_1 over s_2 in its expanding process since s_1 has a higher *local* probability of being reached. Note that this higher local probability does not necessarily imply that s_1 has a higher chance being reached from the initial state – We only use the local reachability as a good hint for that property. But since the algorithm constructs states step-by-step and prefers states reachable with a higher local probability, we are automatically holding up a high probability for any constructed state. This leads to a good approximation of the chance of reaching a state while maintaining

a low complexity footprint: The function only iterates over every predecessor of the two input states.

Heuristic of incoming path probabilities. This approach is a generalization of the above examined heuristic of incoming transition probabilities. Instead of taking the direct incoming transitions into account, this heuristic examines all paths leading to the state with length less than or equal to a constant n . For $n = 1$ the heuristic will transform into the first presented approach since considering incoming paths of length 1 is equivalent to examining the direct incoming transitions.

Hence, we define the set of incoming paths of length $\leq n$ for a state s by

$$InPaths(s, n, \mathfrak{M}) = \left\{ \pi \in Paths_{fin}(\mathfrak{M}) \mid \begin{array}{l} \exists i \leq n (|\pi| = i \wedge \pi[i] = s \wedge \nexists \pi' \in Paths_{fin}(\mathfrak{M}) \\ (\exists i < j \leq n (|\pi'| = j \wedge suffix(\pi, \pi')))) \end{array} \right\}$$

where $suffix(x, x')$ is true iff x is a suffix of x' . Similar as above for a state s of a probabilistic model \mathfrak{M} we define the multiset of incoming probabilities of paths with length $\leq n$ for some $n \in \mathbb{N}$ as follows:

$$InProb(s, n, \mathfrak{M}) = \left\{ p \in [0, 1] \mid \begin{array}{l} \exists \pi \in InPaths(s, n, \mathfrak{M}) \exists \alpha_1 \dots \alpha_{|\pi|-1} \\ (p = \prod_{i=1}^{|\pi|-1} P(\pi[i-1], \alpha_i, \pi[i])) \end{array} \right\}$$

Thus, $InProb$ contains all probabilities of paths up to length n leading to s . We do not consider paths that are sub-paths of already considered paths – $\nexists \pi' \in Paths_{fin}(\mathfrak{M}) (\exists i < j \leq n (|\pi'| = j \wedge suffix(\pi, \pi')))$ –, e.g. if a path $q_1 q_2 q_3$ leading into s is taken into account, we exclude the path $q_2 q_3$ from our calculations as this sub-path does not yield any additional value. For any $n \in \mathbb{N}$ we define the rating function $r_{\mathfrak{M}}$ analogously by using the sum of the probabilities in the $InProb$ multiset:

$$r_{\mathfrak{M}}^n(s) = \sum_{p \in InProb(s, n, \mathfrak{M})} p$$

For an easier understanding we again consider a simple example shown in Figure 17. Here, two states s_1 and s_2 along with their predecessors are presented. We set $n = 3$ for our heuristic, hence we are interested in paths of length 3 leading up to each of the states. Note that various of those paths can visit the same states multiple times. The state s_2 has only two paths of length ≤ 3 leading into it, whereas s_1 has 5, namely $p_5 p_2 p_0$, $p_5 p_3 p_0$, $p_5 p_3 p_1$, $p_6 p_3 p_1$ and $p_6 p_4 p_1$. Since we do not consider paths that are sub-paths of already considered paths, hence e.g. the path $p_2 p_0$ is not used in the calculation.

We then move on to multiply the probabilities along the paths to retrieve the content of the two multisets of ingoing probabilities: $InProb(s_1, 3, \mathfrak{M}) = \{0.0375, 0.1125, 0.015,$

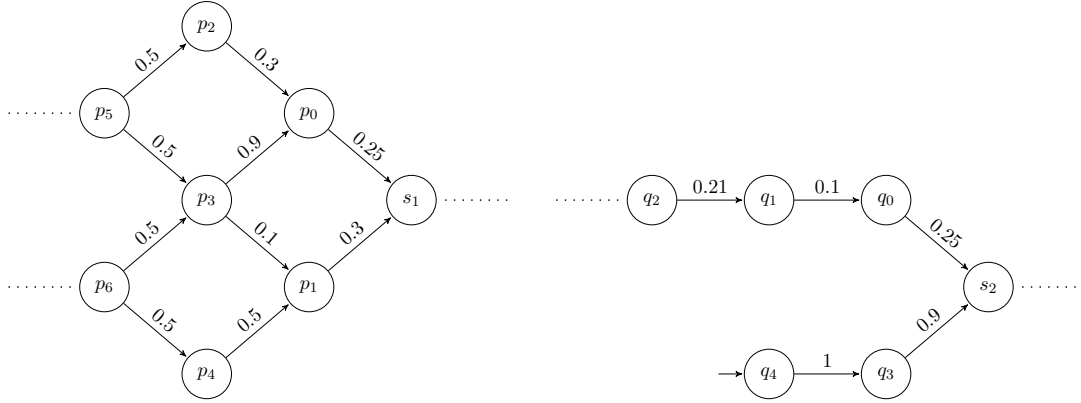


Figure 17: Two states s_1 and s_2 of an exemplary probabilistic model \mathfrak{M} as well as their (indirect) predecessor up to a distance of 3. For the heuristic of incoming paths probabilities it follows that $s_1 <_{\mathfrak{M}} s_2$

$\{0.015, 0.075\}$ and $InProb(s_2, 3, \mathfrak{M}) = \{0.00525, 0.9\}$. We sum up the content both multisets and retrieve the relation $s_1 <_{\mathfrak{M}} s_2$ since $r_{\mathfrak{M}}^3(s_1) = 0.255$ and $r_{\mathfrak{M}}^3(s_2) = 0.90525$.

The advantage of digging a bit deeper into the predecessor transitions of a state is to get a better approximation of the global probability of that state, i.e. the probability of reaching the state from the initial state. As mentioned above, this method is not exact but gives a good hint for the expanding process. In case of PGCL programs a special issue arises: The construction of Definition 30 yields a lot of linear paths with probability 1, which will lead the heuristic of incoming transition probabilities to be equal in a lot of cases. But if we unfold more and more predecessor transitions of a state we will eventually hit a node representing a probabilistic choice. We will achieve this uncovering by employing the heuristic of incoming path probabilities for a good choice of the constant n . Section 6 elaborates on various choices of n for different example programs.

5.2 Global heuristics

Global heuristics allow for a better result but require a greater amount of runtime for their calculations. We will later examine the trade-off between the higher runtime and the better approximation.

Heuristic of the most probable path. Since a global heuristic is able to iterate over the whole model, we can easily fetch the most probable path from the initial node to any state by executing *Dijkstra's algorithm* [15]. We then proceed to rate a state by the greatest probability of being reached – This gives an exact result for the expanding process, hence the most probable paths are expanded first. One should note that Dijkstra's algorithm is contained in the complexity class of $\mathcal{O}(|S| \cdot \log(|S|) + |P|)$ where S is

the set of states and P is the probabilistic transition function of a probabilistic model. For an easier evaluation of all probabilities for which a state s of an MDP \mathfrak{M} is reachable, we define the set of reachable probabilities *ReachProb* as:

$$ReachProb(s, \mathfrak{M}) = \left\{ p \in [0, 1] \mid \begin{array}{l} \exists s_0 \in S_0 \exists \pi \in Paths_{fin}(\mathfrak{M}, s_0, s) \exists \alpha_1 \dots \alpha_{|\pi|-1} \\ (p = \prod_{i=1}^{|\pi|-1} P(\pi[i-1], \alpha_i, \pi[i])) \end{array} \right\}$$

As stated above, we pick the maximum probability for s being reached from the initial state as the state's rating:

$$r_{\mathfrak{M}}^g(s) = \max(ReachProb(s, \mathfrak{M}))$$

The maximal probability $r_{\mathfrak{M}}^g(s)$ for a state s can be efficiently computed by employing Dijkstra's algorithm – We set the inverse probabilities as edge-weights and the states as the nodes of our graph. We then run Dijkstra's shortest path algorithm on every initial state and pick the most minimal path to s . This path has the shortest distance from some initial node to s which – since we defined the distance to be the inverse probabilities – yields us the path with the highest probability leading to s .

We again consider an example to clarify the functioning of this heuristic. Figure 18 depicts a finite MDP \mathfrak{M} where the green colored states s_7 , s_8 and s_9 are waiting to be expanded, i.e. need to be rated by the most probable path heuristic.

To calculate their ratings, we first identify the sets of all paths leading from some initial state to each of the target states. Those are $\{s_0s_2s_4s_7\}$ for s_7 , $\{s_0s_2s_4s_8, s_0s_2s_5s_8, s_1s_2s_5s_8, s_1s_3s_5s_8, s_1s_3s_6s_8\}$ for s_8 and $\{s_0s_2s_5s_9, s_1s_2s_5s_9, s_1s_3s_5s_9\}$ for s_9 . The probabilities along those finite paths are then multiplied and added to the set *ReachProb* for each target state: $ReachProb(s_7, \mathfrak{M}) = \{0.35\}$, $ReachProb(s_8, \mathfrak{M}) = \{0.35, 0.03, 0.003, 0.009, 0.486\}$ and $ReachProb(s_9, \mathfrak{M}) = \{0.27, 0.081, 0.324\}$. The final rating of the states are calculated by fetching the maximum of those sets, thus $r_{\mathfrak{M}}^g(s_7) = 0.35$, $r_{\mathfrak{M}}^g(s_8) = 0.486$ and $r_{\mathfrak{M}}^g(s_9) = 0.324$. We conclude the following heuristical order: $s_9 <_{\mathfrak{M}} s_7 <_{\mathfrak{M}} s_8$.

Hence, the construction algorithm will firstly expand the state s_9 . After this expansion, the ratings are re-calculated since it may be possible that s_7 and s_8 changed their probability of being reached since new transitions to each may have arisen.

We now compare the above determined heuristic result with both local heuristics mentioned above. For the heuristic of incoming transition probabilities it obviously holds that $s_7 <_{\mathfrak{M}} s_8 <_{\mathfrak{M}} s_9$, hence we do not achieve the same result as the global heuristic. In fact, the state which has the lowest probability to be reached, s_9 , is the best rated state. This illustrates that local heuristics will approximate the global heuristic and cannot guarantee optimal results.

If we consider the heuristic of incoming path probabilities for $n = 2$ we are given $r_{\mathfrak{M}}^g(s_7) = 0.35$, $r_{\mathfrak{M}}^g(s_8) = 1.2$ and $r_{\mathfrak{M}}^g(s_9) = 0.63$, hence $s_7 <_{\mathfrak{M}} s_9 <_{\mathfrak{M}} s_8$. Thus, we get the almost

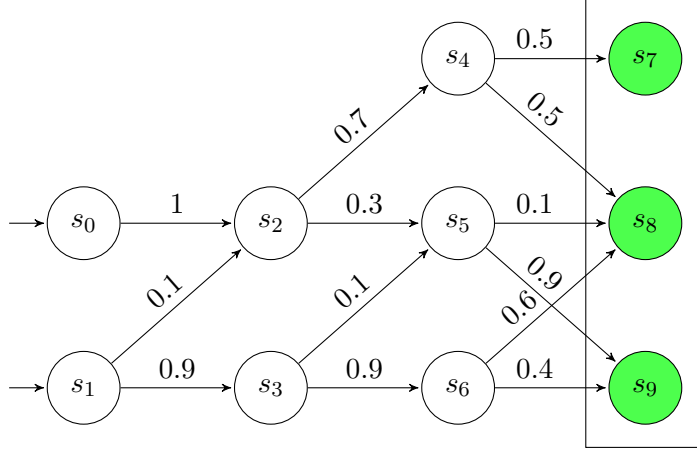


Figure 18: An MDP \mathfrak{M} with two initial states s_0 and s_1 for which we are interested in the rating of the states s_7 , s_8 and s_9 , colored in green. The other states are fully expanded and of no interest to the heuristic. For a better readability, the state labeling and the artificial state $*$ is omitted. When applying the heuristic of the most probable path, $s_9 <_{\mathfrak{M}} s_7 <_{\mathfrak{M}} s_8$ holds.

the same ordering as if we use the most probable path heuristic – Only s_7 and s_9 are swapped in their ordering, but the best rated state s_8 is the same for each heuristic. This illustrates that for a “good” choice of n the local heuristic of incoming path probabilities can achieve a sufficient approximation compared to the global heuristic.

Truncating. As mentioned in the introduction, we can truncate some paths for several formulas, i.e. need not to consider them in the expanding process. This is the case if we are given a probabilistically lower bounded formula $\mathbb{P}_{>i}(\psi)$ and observe that the sum of all paths leading from an initial state to this state is $\lesssim i$ – We then cut off that state and keep on expanding more probable states. This approach calls for the knowledge of all probabilities for a state to be reachable from some initial state, i.e. for the knowledge of $ReachProb(s, \mathfrak{M})$. Local heuristics *do not suffice* for this purpose. This means that we apply the truncating heuristic only in case we have run the heuristic of the most probable path earlier and we are given a formula of the type $\mathbb{P}_{>i}(\psi)$.

Formally, we define the rating function to be either 0 or 1 depending on the comparison between the formula boundary and the state’s probability to be reached:

$$r_{\mathfrak{M}}^t(s) = 0 \text{ if } \sum_{s' \in ReachProb(s)} s' \gtrsim i$$

$$r_{\mathfrak{M}}^t(s) = 1 \text{ if } \sum_{s' \in ReachProb(s)} s' > i$$

Hence, if the chance exists that the currently expanded state will yield path that eventu-

ally satisfies the formula ψ with a probability $> i$, we will expand this state. Otherwise, we set the rating to 0 – In this case it is guaranteed to be a *useless* state for the expansion process (for now). Since the truncating heuristic can only be applied while using the heuristic of the most probable paths, we take both rating functions and tie them together by multiplying both:

$$r_{\mathfrak{M}}^{gt}(s) = r_{\mathfrak{M}}^g(s) \cdot r_{\mathfrak{M}}^t(s)$$

This definition will lead to the rating being 0 for every state not satisfying the truncating approach. Every other state will have a rating greater 0 and will be expanded. To allow for a non-consideration of unusable states, we modify Algorithm 1 to not add any states with rating 0 to the border states. This automatically implies that if the state will become reachable in some later steps with a higher probability, the state may be considered again since its rating will be updated by the heuristic.

Let us again consider Figure 15 on Page 34 and the formula $\mathbb{P}_{\geq 0.5}(Fa)$ from this section's introduction. As stated above, the states s_2 and s_3 need not to be considered for the expansion process at that particular moment – To see why, we calculate the set *ReachProb* for both states: $ReachProb(s_2) = \{0.3\}$ and $ReachProb(s_3) = \{0.4\}$. Since sum of the elements of both sets are < 0.5 , we conclude that $r_{\mathfrak{M}}^t(s_2) = r_{\mathfrak{M}}^t(s_3) = 0$ and consequently $r_{\mathfrak{M}}^{gt}(s_2) = r_{\mathfrak{M}}^{gt}(s_3) = 0$. The expansion process will not put both states in the priority queue and, since $r_{\mathfrak{M}}^g(s_4) = 0.54 > 0$, only s_4 will be expanded.

6 Implementation and testing

The in Section 4.4 and Section 5 examined PGCL model checking approach was implemented in the *Stochastic Reward Model Checker* (StoRM) using C++. StoRM is developed as a replacement for the popular PRISM model checker and succeeds the Markov Reward Model Checker developed by the chair i2 at RWTH Aachen [16, 5]. StoRM is under development and unpublished at the time of writing. In the following, we will give a short introduction to the integration of our approach into the framework.

6.1 StoRM framework

StoRM allows to model check various finite probabilistic models using PCTL and Continuous Stochastic Logic (CSL). It additionally allows for a verification of reward models and compatible reward logics, which are not considered in this paper. StoRM verifies properties over DTMCs, Continuous Time Markov Chains (CTMCs) and MDPs. As StoRM already implements an efficient DTMC and MDP model checker for PCTL formulas, we are given a suitable framework for the implementation of our approach of iterative on-the-fly PGCL model checking. Figure 19 presents how this implementation is integrated into the existing framework. The *Logic* component contains PCTL formulas, whereas DTMCs and MDPs are defined in the *Model* component. Variables and expressions of PGCL programs are handled by the *Expression* component.

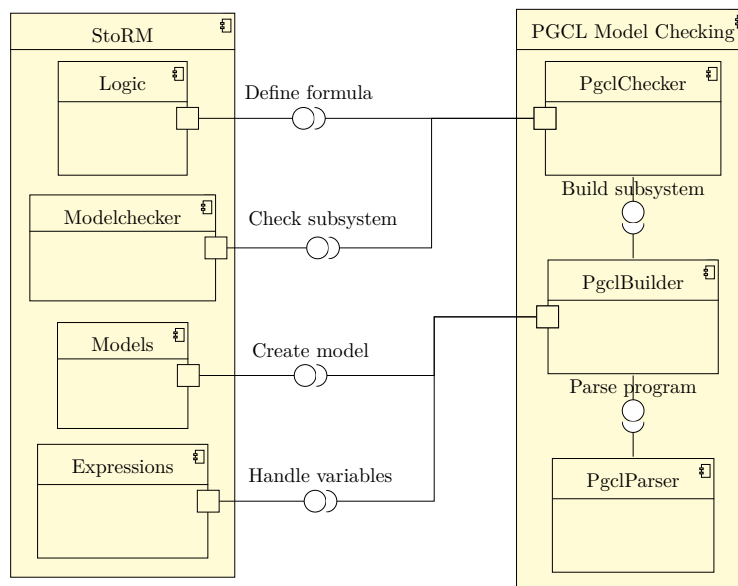


Figure 19: A UML component diagram giving an overview of the relevant modules of the StoRM model checker and the integration of our PGCL model checking approach.

The implementation comprises around 3000 (commented) lines of code, which include a PGCL program parser, as well as the presented iterative PGCL model builder and model checker.

Since StoRM offers an interface for building DTMCs and MDPs, our implemented PGCL model builder creates these probabilistic models as presented in Algorithm 1 up to a given step boundary. The PGCL model checker which initiated the building process then hands both the formula and the model to the appropriate StoRM model checker. StoRM does not yet implement a possibility to check for probabilistically constraint formulas of the form $\mathbb{P}_J(\psi_1 \mid \psi_2)$ for ψ_2 *not* being a reachability property. As this feature is crucial to implement the model checking of the observe statements, we can not check for observations in PGCL programs at the current status of development. As soon as StoRM offers an interface for this, our implementation is ready to check those programs as well.

6.2 Performance and validity evaluation

For the following section, we consider various known benchmarks as well as some small examples to test the performance and validity of our implementation. All tests were performed on an Intel Core i5 3317U with 8 GB DDR3 RAM running a Linux distribution with the kernel version 4.0.7. StoRM was compiled using the GNU Compiler Collection (GCC) version 5.2.0.

In the ensuing tests, we only consider time consumption of the model checker. We use the following test scheme: First, we vary the step width of the building process from small to constructing the full model (or, if the model is infinite, construct the finite part such that only one iteration is needed). Additionally, we compare the execution times for an efficient implementation of the direct predecessor heuristic and no heuristic enabled. We use the *N-Cowboys* benchmark of Section 6.2.1 for a general evaluation of the previously introduced local and global heuristics.

6.2.1 N-Cowboys

The benchmark of N-Cowboys is an adapted version of the cowboy duel introduced in [13, p. 213]. Figure 29 on Page 61 is an expansion of this program such that n cowboys confront each other in a ring configuration. Each cowboys then tries to shoot its right neighbor with a success rate of 0.5. If this event occurs, the neighbor dies and is excluded from the duel. If not, the neighbor itself tries to shoot its right neighbor. This process is continued until exactly one cowboy survives the contest. This yields an infinite MDP, as the starting choice is drawn nondeterministically. Similar to the simple $n = 2$ version of [13], this program terminates almost-surely, i.e. satisfies the PCTL formula $\mathbb{P}_{=1}(\text{Fend})$.

First off, we evaluate on the different heuristics. This includes a naive implementation of the direct predecessor heuristic, an efficient on-the-fly direct predecessor heuristic, the

n-predecessor heuristic, and the most-probable path heuristic. The efficient direct predecessor heuristic sums up the incoming probabilities of the states during the construction of their predecessors, i.e. stores the necessary information such that no additional calculations are needed. The naive approach iterates over each predecessor and sums up the incoming probabilities. The information for the n-predecessor heuristic and the heuristic of the most probable path can not be collected during the construction, i.e. requires extra calculational effort for the retrieval of every state rating. Figure 20 depicts the building times for 45000 states of the N-Cowboys model while varying the previously mentioned heuristics.

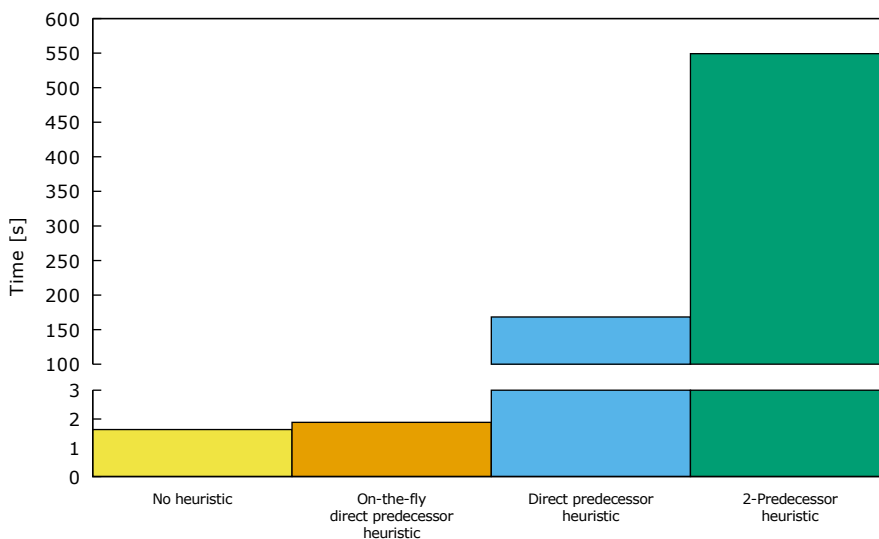


Figure 20: Building times of the model builder for the N-Cowboys model for 45000 states with various heuristics.

We directly note a vast performance gap between the heuristics requiring additional calculations – The direct predecessor heuristic and 2-predecessor heuristic – and the efficient heuristic implementation respectively no heuristic. Precisely, the naive implementation of the direct predecessor heuristic uses around 85 times more computational time. The efficient approaches yield a construction rate of 27600 states per second (no heuristic) respectively 23800 states per second (efficient direct predecessor heuristic), whereas the naive implementation of the direct predecessor heuristic constructs 265 states per second. Using the N-Predecessor heuristic for $n = 2$ results in the building time being more than 275 times as long as the efficient approaches and a construction rate of 81 states per second. A comparison between the naive implementation of the direct predecessor heuristic and the 2-predecessor heuristic reveals a factor of 3.25. The heuristic of the most probable path was implemented, but the model builder did not return from the building process in a time frame of four hours. On the positive side, we observe only a

small performance overhead for the efficient direct predecessor heuristic of around 16%, as the calculational effort is shifted to the storage.

Concluding, we consider the presented heuristical approaches – Besides the efficient direct predecessor heuristic – as highly inefficient as they consume way more computational power than they bring in. Practical tests have shown that for most of the test cases presented hereafter the model checker, when using a heuristic, did not even return a result in a reasonable amount of time. Especially a usage of the heuristic of the most probable path results in a drastic loss of performance. We conclude that it is way more promising to use a fast breadth-first building process either without a heuristic or with the efficient direct predecessor heuristic enabled. As a result, we restrict ourselves to comparing only the efficient building approaches in the upcoming test cases and do not further evaluate on the remaining heuristics.

On the other hand, heuristics can prove useful in some cases, where the state space “explodes” such that a breadth-first building process will get stuck in an exponentially growing part of the model, where no result can be reached. We will examine such an application of the direct predecessor heuristic in Section 6.2.5.

For the performance evaluation of the building process, we vary the step width after which the model checker is called. In this manner, we can observe an optimal parameterization of the model checker for the tested systems, i.e. on which step width it is the fastest. We check the PCTL property $\varphi = \mathbb{P}_{\geq 0.001}(\text{Falive2} = 0)$, i.e. if the chance of cowboy two being dead somewhen is greater than 0.1%. This property is satisfied after constructing around 35000 states. We additionally examine the number of iterations needed by the model checker to return a result and compare the relation between the iteration number and the computation time. Figure 21 presents the before-mentioned data.

We observe an inverse proportionality between time and step width as well as between iterations and step width – Hence, both graphs show a hyperbolic behavior. If we chose a very small step width we obtain a longer runtime as the overhead of running the model checker and initiating a new iteration is quite costly. When reducing the step width, the number of iterations reduces vastly. Additionally, we observe a reduction of the runtime, as we reduce the overhead of running the model checker. It is to be expected that the runtime is at its lowest at a step width of 35000 states since we barely construct the essential part of the model needed to verify the property – We introduce almost no overhead. The runtime increases as the step width rises over 35000 states as we introduce unnecessary states to the model. If turning on the efficient direct predecessor heuristic, the performance of the model checker is influenced only by a linear factor – Both graphs are copies, but shifted vertically. In no case we notice a performance improvement when using the heuristic.

We additionally note local linear growth if we fix the number of iterations. As an example, after 17000 states the model checker needs only two iterations, and from then on the computation time grows linearly until the iteration number is again reduced to

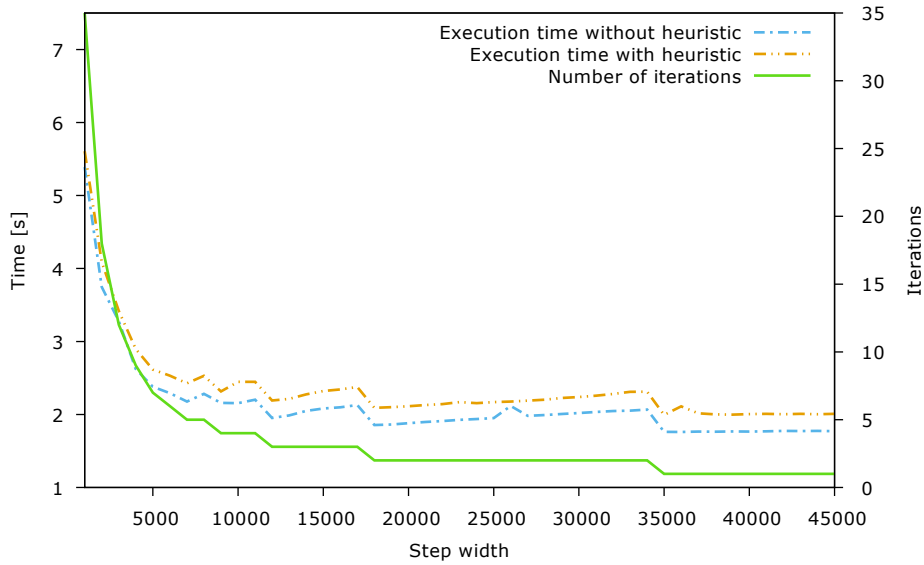


Figure 21: Execution times and iteration numbers for an increasing step width of the iterative PGCL model checker on the N-Cowboys model.

one. This seemingly strange behavior can be explained by the “overreaching” of the builder: Even when performing more than one iteration, multiple unnecessary states are introduced in the very last iteration of the building process. This number of unnecessary states is at its smallest when the number of iterations has just reduced.

6.2.2 Lotka-Volterra

The *Lotka-Volterra* principles describe a dynamic biologic model where a predator and a prey with a mutual dependency on their population number is given. Both populations follow some basic rules which result in a periodic fluctuation of their population numbers, where the number of predators follows the number of preys in particular margin over time. For further information on the equations of Lotka and Volterra, we refer to [17, p. 127]. This process can be modeled as a PGCL program presented in Figure 27 on Page 59, which we adapted from [18]. As noted above, the presented program yields an infinite model, as the predator and prey keep on hunting and reproducing over time. We start with a fixed population of 4 tigers (predators) and 100 goats (preys).

We are now interested in verifying surviving chances of the populations such as whether the probability of all preys to be killed is greater than 1% or whether the probability that at least one tiger is alive is greater than 90%. Particularly, for the performance test we verify $\mathbb{P}_{\geq 0.0001}(\mathit{Fend})$, i.e. if the probability of one population dying out is greater than 0.1%.

Figure 22 depicts the performance behavior using the already-introduced testing scheme. Beside the hyperbolic relation between step width and iteration numbers we again note a local linear behavior for the execution time if we fix an iteration number. But, in contrast to the N-Cowboys model, the performance growth is not as vast for an increasing step width – On average, the execution time stays the same throughout every step width. The largest peak is observed at around 7000 states right before the iteration number drops down to one.

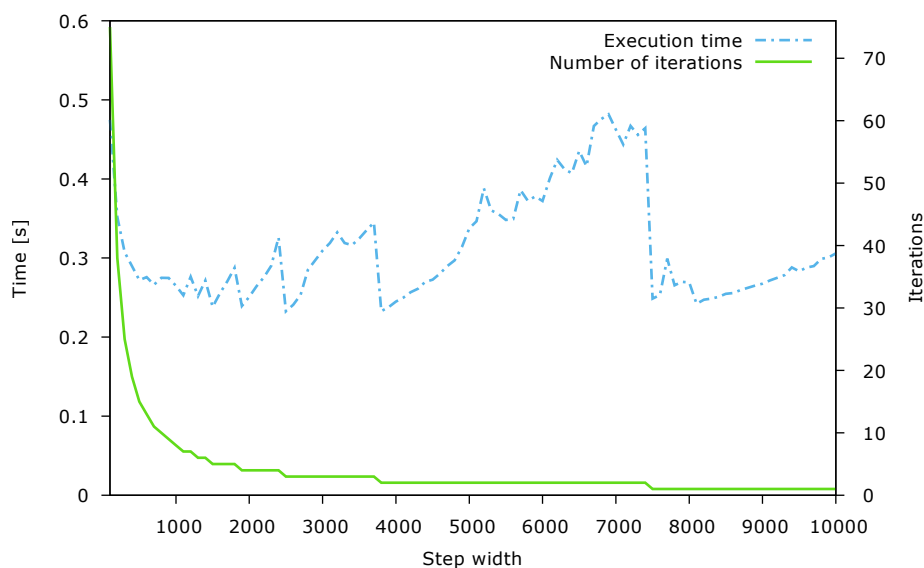


Figure 22: Execution times and iteration numbers for an increasing step width of the iterative PGCL model checker on the Lotka-Volterra model.

The property is verified after constructing around 7500 states. We monitor some interesting characteristics of the model checker performance: Right before constructing 7500 states, we reach a global maximum. This maximum is contingent on the fact that we will build 14000 states until the property is verified – Whereas in contrast for a step width of 8000, only those 8000 states need to be constructed. As the model checker of StoRM relies on an efficient sparse matrix approach, the model checking times remain comparatively low, hence the big factor in the model checking process is the model building. This can be observed by the performance factor between the step width of 7000 and 7500 states: We need around twice as much time for the step width of 7000 states.

Concluding, we note that for infinite models, a right choice of the step width is crucial to the performance of the model checker – Here, human intuition is required to increase the performance as guessing the step width automatically would be a hard problem.

6.2.3 Herman

In network protocols, the problem of *self-stabilization* arises – If we are given a system in an arbitrary (possibly invalid) configuration it has to stabilize itself to a valid configuration. This process can be done using self-stabilizing algorithms. Herman proposed such an algorithm in 1990 [19], where a token ring of n processes is given. Starting with any arbitrary token distribution over the processes, the algorithm always finds a valid configuration where only one process has the token. Figure 28 on Page 60 represents this algorithm for $n = 3$. The popular PRISM model checker uses this model as a benchmark¹ such that we compare our results to PRISM later on.

Since the model of the PGCL program is finite, we can easily expand the whole state space and run a model checker on it. We can verify the correctness of Herman’s protocol: As it is claimed that the algorithm results almost-surely in a valid configuration, we check the PCTL property $\varphi = \mathbb{P}_{=1}(\mathbf{F}(x_1 + x_2 + x_3 = 1))$. This property can only be verified after constructing the whole model, as no subsystem satisfies it.

We run our test cases for $n = 9$ which yields around 20000 model states. The PRISM model uses only about 512 states and 20000 transitions, as the behavior is modeled in a more direct fashion – PGCL programs introduce an overhead in general and are less efficient in terms of storage compared to directly specifying the model. Figure 23 presents the performance of the iterative PGCL model checker on the Herman model.

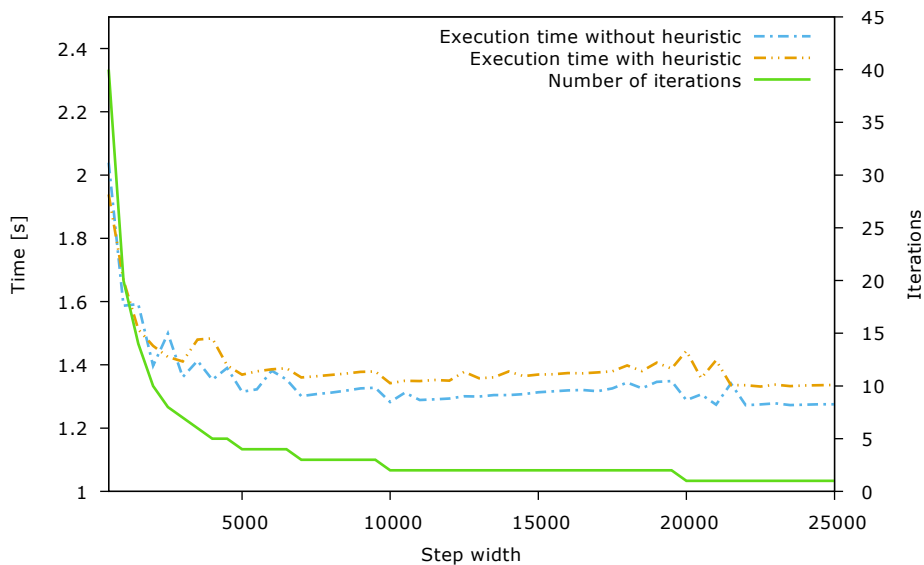


Figure 23: Execution times and iteration numbers for an increasing step width of the iterative PGCL model checker on the Herman model.

¹<http://www.prismmodelchecker.org/casestudies/self-stabilisation.php#herman>, 7.8.2015

We again note the same behaviors as in the previously examined benchmarks: The local linear growth and the global hyperbolic correlation between performance and step width respectively iteration number. As the model is finite and fully constructed at 20000 states, the model checker notices that it does not need to construct any more states and stops the building process early. Hence, we observe the performance after 20000 states staying the on the same level. In contrast to the infinite Lotka-Volterra model, we do not note a peak right before the drop of the iteration number to one – This is explained by the model checker returning early even after starting the second iteration with just a few states left to expand, as the model is finite and we need to construct the whole model anyways. As a result, we note that the step width for finite models and properties calling for full construction is a minor factor for the performance, as long as it does not stay in the very low ranges below 1000 states.

In comparison with the PRISM model checker we obtain quite good results: For $n = 9$, PRISM needs around 1.1 seconds to verify the above stated formula, whereas the iterative PGCL model checker yields an average time consumption of 1.4 seconds. Thus, both results lie in a comparable range, where the fact that the PGCL model is a quite inefficient depiction of the protocol needs to be taken into account.

6.2.4 Robot

The robot test case is adapted from the exemplary models presented for the PRISM model checker [5]. The in Figure 30 on Page 62 presented PGCL program simulates a robot moving on a fixed-size grid, starting in the lower-left corner. The robot moves to the right as long as this is possible with respect to the grid borders. If it reaches this border, it starts moving up, until the upper-right corner is reached eventually. Additionally, a janitor cleans the grid the robot moves on. As the janitor proceeds to randomly choose any direction to move on, it can block the robot’s path. The robot then can not move further and has to wait for the janitor to go on. The movement of the janitor is not constrained by any borders, i.e. he can move infinitely into any direction, which subsequently results in an infinite model. As the janitor can not block the robot’s way forever (he is guaranteed to eventually change its location), the robot will almost-surely reach its destination.

As we are given an infinite model, standard model checking approaches on finite models, e.g. the PRISM model checker, will not suffice – They cannot verify properties over infinite systems directly. This is when our presented approach comes in handy: Even though the model is infinite, we can verify that the robot will reach its target position with some probability. This is a result of the set of all possible combinations of the robot’s path and interruptions by the janitor is finite.

We apply our model checker on the formula $\varphi = \mathbb{P}_{\geq 0.1}(\mathit{Fend})$. Figure 24 depicts the test results. We directly note the heuristic approach leading to a vastly longer runtime for a smaller step width. Hence, we observe a case where even the efficient direct predecessor heuristic proves in fact a lot more time-consuming.

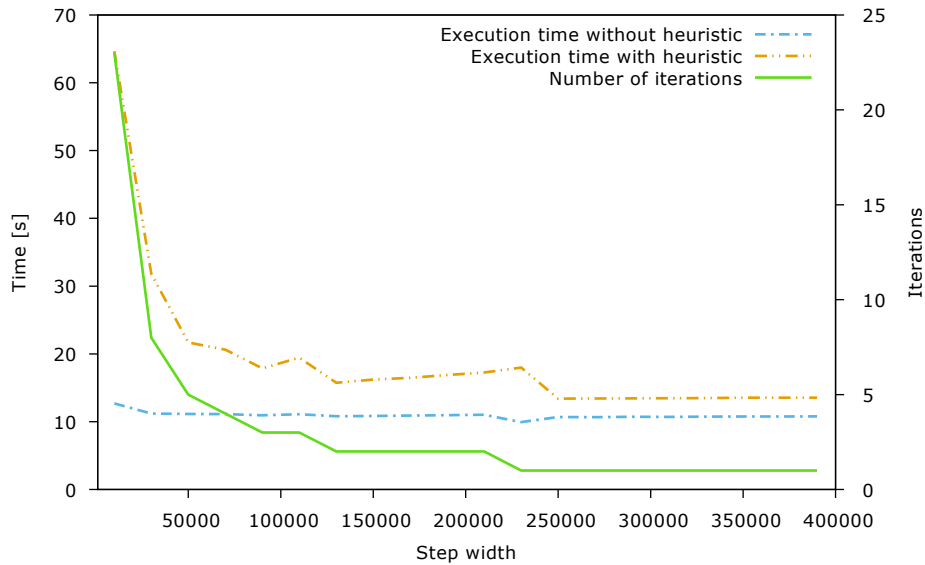


Figure 24: Execution times and iteration numbers for an increasing step width of the iterative PGCL model checker on the Robot model.

The model checker needs to construct at least 22000 states to verify the property, where we observe the global minimum which is around ten seconds. With a disabled heuristic, model checking times do not vary greatly for different step widths. PRISM uses a finite version of the Robot model as a case study². To verify the above-mentioned formula, PRISM needs around one second on the finite version of the Robot model. Hence, ten seconds for the infinite version of the Robot model is a comparable result, as PRISM itself would not be able to inspect the model anyways.

6.2.5 State explosion

Figure 31 on Page 63 presents simple probabilistic program resulting in an exponential growth of the state space. This is done by increasing the “dummy” variable x in the second loop, which leads into a creation of two previously unknown successor states for every state of the second loop. The first loop models a linear growth: Every state has two successor from which only one is fresh – The state where k is increased. A sketch of this probabilistic model is presented in Figure 25. This PGCL programs with a hugely growing infinite state space will lead the builder into troubles due to memory limitations.

When constructing this model using a non-heuristical approach, we construct the model breadth-first. This results in a very slow discovery of valuations of k . Especially if we are interested in properties over k , a heuristical approach yields faster results than a

²<http://www.prismmodelchecker.org/casestudies/robot.php>, 7.8.2015

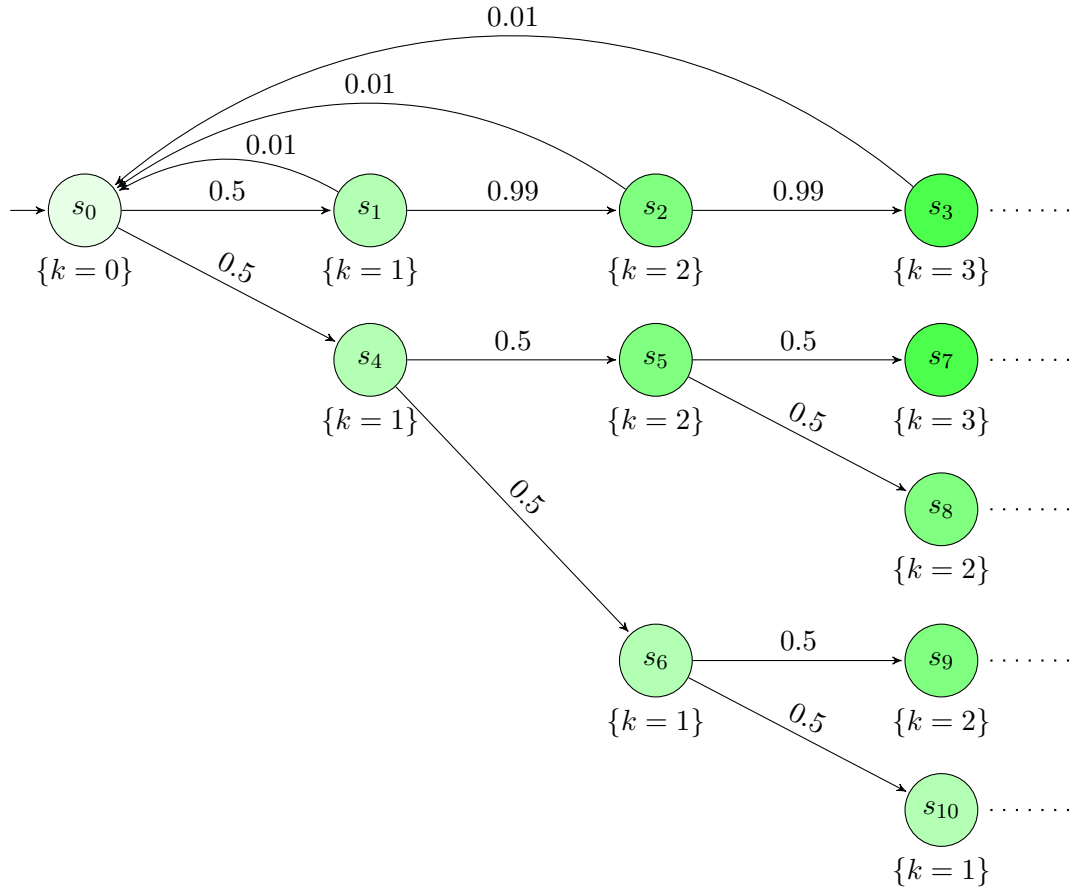


Figure 25: A part of the infinite DTMC representing the probabilistic model of the program depicted in Figure 31 on Page 63.

standard non-prioritized expansion. In some cases we will not even get faster results, but get a result in the first place, where the non-heuristical approach would run into a memory-overflow due to the exponential growth of the model.

Checking the formula $\mathbb{P}_{\geq 0.2}(Fk \geq 10000)$ does in fact get our model checker into troubles when disabling the heuristic: Even after a few hours, no result has been returned. If we turn on the efficient direct predecessor heuristic, the states with increasing k are preferred over the “exploding” model part. The measured performance with an enabled heuristic is depicted in Figure 26, where execution times vary between one and three seconds. We again note the local linear behavior as the property of $k \geq 10000$ is satisfied after around 32000 states. This number is magnitudes greater when using no heuristic.

As a conclusion, we note that using the efficient direct predecessor heuristic is only inducing a small amount of overhead in the most cases, but being very advantageous for

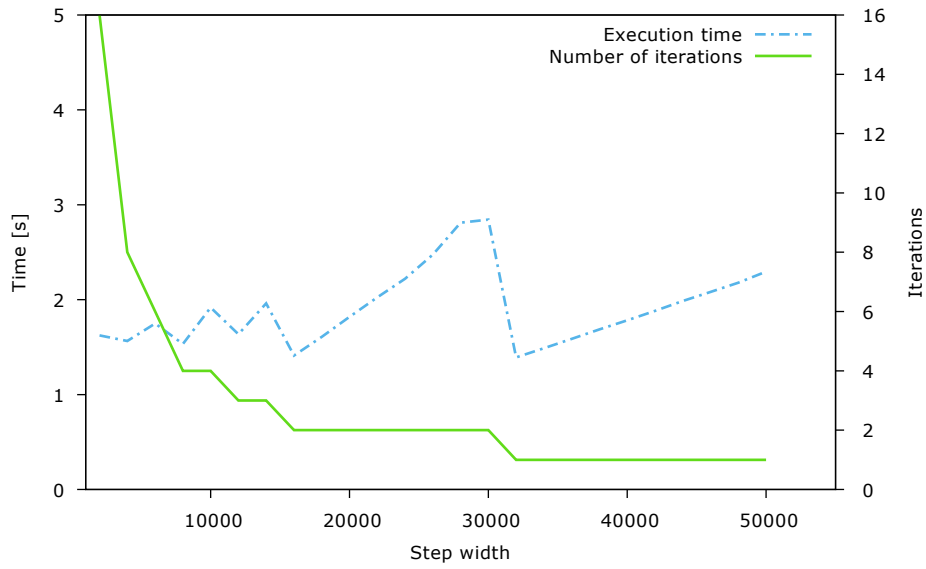


Figure 26: Execution times and iteration numbers for an increasing step width of the iterative PGCL model checker on the state explosion model.

some models where the state space for parts reachable only with small probabilities is exploding. Hence, we advise a general activation of this heuristic.

7 Conclusion and future work

In this paper, we presented an iterative on-the-fly model checking approach for PGCL programs. PGCL programs can either be finite or infinite, and for both types of programs our model checker shows an advantageous behavior as results can be found earlier in the building process. The need for building the complete model and executing the model checker afterwards is dropped.

Test results have shown a competitive performance of our implementation in comparison to the popular PRISM model checker. As an addition to checking finite programs, the iterative PGCL model checker is able to run on infinite programs, and return a result as long as the property is transferable from the current subsystem. The step width of the building process is crucial to the performance: In some tests, a disadvantageous step width lead to a doubled execution time. Nevertheless, human intuition on the size of the model is highly advantageous, as the step width can be chosen intelligently by the user.

Additionally, we examined various local and global heuristics for exposing promising paths to the model builder, but besides an efficient implementation of the direct predecessor heuristic, tests showed that the heuristical approach is too costly in terms of computational time.

As future work, we consider extending the iterative model checking approach such that actual model checking *during* the building process is performed, e.g. collecting data such that reachability properties can be proven directly. This would mold the building and checking processes together and hopefully result in an improved performance. Furthermore, more involved and efficient heuristics can be developed where building times are kept low by storing necessary information in the memory, in the way the efficient direct predecessor heuristic does.

References

- [1] D. Bert. *ZB 2003: Formal Specification and Development in Z and B: Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*. Lecture Notes in Computer Science. Springer, 2003. 1
- [2] Marie Dufлот, Laurent Fribourg, Thomas Herault, Richard Lassaigne, Frédéric Magniette, Stephane Messika, Sylvain Peyronnet, and Claudine Picaronny. Probabilistic model checking of the CSMA/CD, protocol using PRISM and APMC. In *Proceedings of the 4th International Workshop on Automated Verification of Critical Systems (AVoCS)*, volume 128 of *Electronic Notes in Theoretical Computer Science Series*, pages 195–214, 2004. 1
- [3] Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill(tm): A bayesian skill rating system. In *Advances in Neural Information Processing Systems 20*, pages 569–576. MIT Press, January 2007. 1
- [4] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. 1, 15
- [5] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 2, 43, 50
- [6] M. Davis. *Computability & Unsolvability*. Dover Books on Computer Science Series. Dover, 1958. 2
- [7] D.P. Bertsekas and J.N. Tsitsiklis. *Introduction to Probability*. Athena Scientific books. Athena Scientific, 2002. 4
- [8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. 5, 13, 23, 29
- [9] Nils Jansen, Florian Corzilius, Matthias Volk, Ralf Wimmer, Erika Ábrahám, Joost-Pieter Katoen, and Bernd Becker. Accelerating parametric probabilistic verification. *CoRR*, abs/1312.3979, 2013. 7, 29
- [10] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag. 10
- [11] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994. 10
- [12] Husain Aljazzar and Stefan Leue. Directed explicit state-space search in the generation of counterexamples for stochastic model checking, 2009. 14
- [13] Annabelle McIver and Carroll Morgan. *Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science)*. SpringerVerlag, 2004. 15, 44

- [14] Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *CoRR*, abs/1504.00198, 2015. 20, 28
- [15] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001. 39
- [16] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance Evaluation*, 68(2):90–104, 2011. 43
- [17] F. Brauer and C. Castillo-Chavez. *Mathematical Models in Population Biology and Epidemiology*. Texts in Applied Mathematics. Springer New York, 2001. 47
- [18] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE Future of Software Engineering)*. IEEE, May 2014. 47
- [19] Ted Herman. Probabilistic self-stabilization, 1990. 49

8 Appendix

Figure 2 presents a function to check for transferability of a result.

Algorithm 2: ISTTRANSFERABLE Implements the transferability matrix depicted in Table 1 and Table 2.

Input: A PCTL formula φ , a model type $type$ and a model checking result $result$.

Result: Returns *true* if the result is transferable for φ on the given model type, otherwise *false*.

```
1 if ( $\varphi = \mathbb{P}_{<i}(\psi) \wedge result = no$ )  $\vee$  ( $\varphi = \mathbb{P}_{>i}(\psi) \wedge result = yes \wedge type = DTMC$ ) then
2   | return true
3 else
4   | return false
```

The single-step expansion is depicted in Figure 3. This function expands a given state according to the rules of Definition 30, and updates the labeling of the model accordingly.

Figures 27 to 31 present the test cases on which our approach was examined.

Algorithm 3: EXPAND Expands the state s of the model \mathfrak{M} according to the SOS-rules given in Definition 30.

Input: A model state $s = (l, v)$, a model \mathfrak{M} , a set of expressions $Expr$, and the belonging PGCL program graph $G_P = (V, L_P, E)$

Result: Alters the model such that the given state is expanded. Returns a set of newly created states.

```

1 if  $l \geq 0$  then
2   Set  $newStates := \emptyset$ 
3   for  $l'$  in  $\{n \in V \mid \exists(l, n) \in E\}$  do
4     if  $L_P(l, l') = x := e$  then
5        $v := v[x := e]$ 
6       State  $newState := (l', v)$ 
7        $P := P \cup \{(s, \circ, newState) \mapsto 1\}$ 
8     else if  $L_P(l, l') = observe(e) \wedge e[v] = 1$  then
9       State  $newState := (l', v)$ 
10       $P := P \cup \{(s, \circ, newState) \mapsto 1\}$ 
11     else if  $L_P(l, l') = observe(e) \wedge e[v] \neq 1$  then
12       $P := P \cup \{(s, \circ, \perp) \mapsto 1\}$ 
13     else if  $L_P(l, l') = e \wedge e[v] = 1$  then
14      State  $newState := (l', v)$ 
15       $P := P \cup \{(s, \circ, newState) \mapsto 1\}$ 
16     else if  $L_P(l, l') = \{e\} \wedge 0 \leq e[v] \leq 1$  then
17      State  $newState := (l', v)$ 
18       $P := P \cup \{(s, \circ, newState) \mapsto e[v]\}$ 
19     else if  $L_P(l, l') = n_i$  then
20      State  $newState := (l', v)$ 
21       $P := P \cup \{(s, n_i, newState) \mapsto 1\}$ 
22     if  $newState \notin S$  then
23       $newStates := newStates \cup \{newState\}$ 
24       $S := S \cup \{newState\}$ 
25     for  $e \in Expr$  do
26      if  $e[v] = true$  then
27         $L(newState) := L(newState) \cup \{e\}$ 
28 else
29    $P := P \cup \{(s, \circ, \perp) \mapsto 1\}$ 
30 return  $newStates$ 

```

```

1 int goats      := 100;
2 int tigers     := 4;
3 double c1      := 1.0;
4 double c2      := 5.0;
5 double c3      := 1.0;
6 double rate1   := 0.5;
7 double rate2   := 0.5;
8 double rate3   := 0.5;
9 double rate    := 1;
10 int dwellTime := 0;
11 int curTime   := 0;
12 int b         := 0;
13 while (tigers > 0 & goats > 0) {
14     dwellTime := 0;
15     b := 1;
16     if (goats > 0 & tigers > 0) {
17         rate1 := c1 * goats;
18         rate2 := c2 * goats * tigers;
19         rate3 := c3 * tigers;
20         rate := rate1 + rate2 + rate3;
21         // geometric distribution with p = rate
22         while (b >= 1) {
23             {b := 1;} [rate] {b := 0;}
24             dwellTime := dwellTime + 1;
25         }
26         curTime := curTime + dwellTime;
27         {goats := goats + 1;}
28         [rate1/rate]
29         {{goats := goats - 1;} [rate2/rate] {tigers := tigers - 1;}}
30     } else { if (goats > 0) {
31         rate := c1 * goats;
32         // geometric distribution with p = rate
33         while (b >= 1) {
34             {b := 1;} [rate] {b := 0;}
35             dwellTime := dwellTime + 1;
36         }
37         curTime := curTime + dwellTime;
38         goats := goats + 1;
39     } else { if (tigers > 0) {
40         rate := c3 * tigers;
41         // geometric distribution with p = rate
42         while (b >= 1) {
43             {b := 1;} [rate] {b := 0;}
44             dwellTime := dwellTime + 1;
45         }
46         curTime := curTime + dwellTime;
47         tigers := tigers - 1;
48     } } }
49 }

```

Figure 27: *Lotka-Volterra*. A PGCL program representing the (infinite) Lotka-Volterra process which is simulating a periodic predator-prey relationship.

```

1 int x1 := 0;
2 int x2 := 0;
3 int x3 := 0;
4
5 // determine starting token setup on the ring.
6 {x1 := 0;} [] {x1 := 1;}
7 {x2 := 0;} [] {x2 := 1;}
8 {x3 := 0;} [] {x3 := 1;}
9
10 // finds a ring configuration with exactly one token in the ring.
11 while((x1 + x2 + x3) != 1) {
12     { // process 1 has its turn
13         if(x1 = x3) {
14             {x1 := 0;} [0.5] {x1 := 1;}
15         } else {
16             x1 := x5;
17         }
18     } [] {
19     { // process 2 has its turn
20         if(x2 = x1) {
21             {x2 := 0;} [0.5] {x2 := 1;}
22         } else {
23             x2 := x1;
24         }
25     } [] {
26     { // process 3 has its turn
27         if(x3 = x2) {
28             {x3 := 0;} [0.5] {x3 := 1;}
29         } else {
30             x3 := x2;
31         }
32     } } }
33 }

```

Figure 28: *Herman*. The self-stabilizing protocol by Herman where the goal is to find a ring configuration where exactly one element has the token, starting with some arbitrary token distribution. In this example, the ring contains three elements (x_1 , x_2 , and x_3).

```

1 int alive1 := 1;
2 int alive2 := 1;
3 int alive3 := 1;
4 int turn := 1;
5 // nondeterministic starting choice.
6 {turn := 1;} [] {{turn := 2;} [] {turn := 3;}}
7 // duell as long as at least two cowboys are alive.
8 while(alive1 + alive2 + alive3 != 1) {
9     if(turn = 1 & alive1 = 1) {
10         {alive2 := 0; turn := 3;} [0.5] {turn := 2;}
11     }
12     if(turn = 2 & alive2 = 1) {
13         {alive2 := 3; turn := 1;} [0.5] {turn := 3;}
14     }
15     if(turn = 3 & alive3 = 1) {
16         {alive1 := 0; turn := 2;} [0.5] {turn := 1;}
17     }
18 }

```

Figure 29: *N-Cowboys*. An generalized version of a cowboy duel – A ring shootout where each cowboy shoots its neighbor. In this example, we consider three cowboys.

```

1 // robot moves on a 16x16 grid.
2 int sizeofGrid := 16;
3 // robot starts in the lower left corner.
4 int robotX := 1; int robotY := sizeofGrid;
5 // janitor starts in the grid middle.
6 int janitorX := ceil(sizeofGrid/2); int janitorY := ceil(sizeofGrid/2);
7 // signal sending variables.
8 double sendProb := 0.1;
9 double receiveProb := 0.5;
10 int messageOnChannel := 0;
11 // iterates as long as the robot is not in the upper right corner.
12 while(!(robotX = sizeofGrid & robotY = 1)) {
13     // checks if we can go to the right and the janitor is not there.
14     if(robotX < sizeofGrid & !((janitorX = robotX + 1) & (janitorY =
robotY))) {
15         robotX := robotX + 1;
16     }
17     // checks whether we can go up and the janitor is not there.
18     if(robotX = sizeofGrid & !((janitorX = robotX) & (janitorY = robotY -
1))) {
19         robotY := robotY - 1;
20     }
21     // robot tries to send signal if no message is on the channel.
22     if(messageOnChannel = 0) {
23         {messageOnChannel := 1;}[sendProb]{messageOnChannel := 0;}
24     } else {
25         // if some message was on the channel, we try to receive (again).
26         {messageOnChannel := 0;}[receiveProb]{messageOnChannel := 1;}
27     }
28     // moves the janitor randomly in one direction.
29     {janitorX := janitorX + 1;}
30     [0.25]
31     {{janitorX := janitorX - 1;}
32     [1/3]
33     {{janitorY := janitorY + 1;}
34     [0.5]
35     {janitorY := janitorY - 1;}}}}
36 }

```

Figure 30: *Robot*. An infinite PGCL program simulating a robot moving on a limited two-dimensional square. It additionally contains a janitor moving around not limited by any borders and possibly blocking the robot.

```

1 int k := 0;
2 int x := 0;
3 {
4   x := -1;
5   while(1=1) {
6     {k := k + 1;} [0.99] {k := 0;}
7   }
8 } [0.5] {
9   x := 1;
10  while(1=1) {
11    {k := k + 1;} [0.5] {x := x + 1;}
12  }
13 }

```

Figure 31: *State explosion*. A PGCL program generating an exponentially growing state space.