

Bachelor Thesis

**Qualitative Model Checking
of Markov Decision Processes on GPUs**

Sascha Vincent Kurowski

June 18, 2015

submitted to the Chair for Computer Science 2
in partial fulfillment of the requirements for the degree of Bachelor of
Science
at the RWTH Aachen University

First reviewer:
Prof. Dr. Ir. Joost-Pieter Katoen

Second reviewer:
Apl. Prof. Dr. Thomas Noll

Acknowledgements

First of all, I want to thank Prof. Katoen for his inspiring lecture *Introduction to Model Checking*, which got me interested in formal verification.

Also I want to thank Christian Dehnert for his support, advice, and for his helpful explanations of the internals of the StoRM framework as well as his nearly instantaneous replies to my various emails.

Furthermore, I give thanks to Marko Kajzer and Simon Kern for their helpful feedback on all stages of this thesis and especially for their extensive proofreading.

I owe thanks to my girlfriend Esra for being so supportive throughout this busy time of working.

Last but not least, I want to express my gratitude towards my mom for everything she did for me and towards my dad as well, being the one to draw my interest to computer science in the first place.

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 18. Juni 2015

Sascha Vincent Kurowski

Zusammenfassung

Probabilistisches Model Checking ist eine Technik zur formalen Verifikation von Systemen, die probabilistisches und nicht-deterministisches Verhalten aufzeigen. In der Praxis jedoch werden Modelle oftmals sehr groß und die Analyse ebenjener aufgrund von Zeit- oder Speicherbeschränkungen undurchführbar. Grafikprozessoren (GPUs) sind für große parallele Berechnungen an Matrizen und Vektoren optimiert, welche auch Hauptbestandteil von probabilistischem Model Checking sind. Wir zeigen ein Verfahren der Nutzung von Allzweck-Berechnung auf Grafikprozessoren (GPGPU) zur Parallelisierung der Algorithmen für die Analyse des qualitativen Fragments der Probabilistic Computation Tree Logic (PCTL). Abschließend evaluieren wir die Wirksamkeit unseres Ansatzes anhand verschiedener Fallstudien.

Abstract

Probabilistic Model Checking is a technique for the formal verification of systems that exhibit probabilistic and nondeterministic behavior. However, in practice the system models become very large and their analysis becomes infeasible because of memory or time constraints. Graphics Processing Units (GPUs) are optimized for massively parallel computations over matrices and vectors, which are also at the core of probabilistic model checking. We present a method of using general purpose computing on GPUs (GPGPU) to parallelize the algorithms used for the analysis of the qualitative fragment of the probabilistic computation tree logic (PCTL). Finally, we consider different benchmark models in order to evaluate the effectiveness of the approach.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	State Space Explosion	3
1.3	Outline	3
2	Preliminaries	5
2.1	Markov Models	5
2.1.1	Discrete-Time Markov Chains	5
2.1.2	Markov Decision Processes	7
2.2	Probabilistic Computational Tree Logic	10
2.2.1	Qualitative Fragment	12
2.3	StoRM Framework	12
2.4	The GPGPU approach	13
2.4.1	CUDA	13
3	Qualitative Model Checking	17
3.1	Algorithms	17
3.1.1	ProbGreater0E	18
3.1.2	ProbGreater0A	19
3.1.3	Prob1E	21
3.1.4	Prob1A	23
3.2	Input	26
4	Implementation and evaluation	31
4.1	Implementation	31
4.2	Case studies	36
4.2.1	coin case study	36
4.2.2	csma case study	37
4.2.3	leader case study	38
5	Conclusion	41
5.1	Summary & Evaluation	41
5.2	Future Work	41
	Appendix	43
	Bibliography	51

List of Figures

1	Schematic of the model checking approach	2
2	DTMC for the <i>gambler's ruin problem</i>	7
3	Non-deterministic interleaving of probabilistic processes	8
4	Schematic of the CUDA GPU architecture	15
5	Example Markov Decision Process	20
6	Example execution of Prob1E	24
7	Schematic of the general algorithm execution	32
8	Shared coin protocol results	37
10	CSMA/CD protocol results	39
11	Asynchronous leader election protocol results	40

1 Introduction

This chapter introduces the notion of (probabilistic) model checking as a strict method to verify that a system meets a specification. We illustrate the motivation for such a technique and point out one of the fundamental problems of model checking before we end this chapter with an outline of this thesis.

1.1 Motivation

With system designs becoming more and more complex, the difficulty of manually analyzing them is increasing rapidly towards impossibility. However, for some systems, e.g. safety-critical hardware or software systems, being able to verify their compliance with a given specification is of prime importance. While building a prototype and testing it may be used to increase confidence in the system, a successful test does not guarantee the absence of specification violations - tests merely facilitate finding bugs because usually only running a subset of all possible tests of a system is feasible due to time and complexity constraints.

The need for comprehensive analysis techniques led to the idea of creating mathematical proofs such as the Hoare calculus by Hoare [10]. Another approach to automatic analysis is model checking, a formal verification technique used to determine the compliance of a system with a specification. Figure 1 depicts the general model checking approach. The model checker, a software model checking tool, requires a formal representation of the system in question as well as a formal specification of the properties the system should fulfill. The formal representation of the system is usually a labeled transition system comprised of the system's states and the possible transitions between those states. The specification's formalization is usually a set of temporal logic formulas, e.g. CTL, which can express qualitative properties such as "the system will never reach a bad state" or "there can never be more than one process in the critical section". If the system fulfills all the properties of the specification, the model checker terminates with the result *satisfied*. If, however, the system violates any of the properties, the model checker terminates with the result *violated* and optionally a counterexample, i.e. a violating state traversal of the system, which can be used to find flaws in the system (or the specification). After formalizing the system and the requirements, which is an one time only task unless changes occur, model checking the system is a fully automated procedure.

Qualitative model checking has shown to be very successful in non-probabilistic settings, but is not of much use in settings where quantitative models

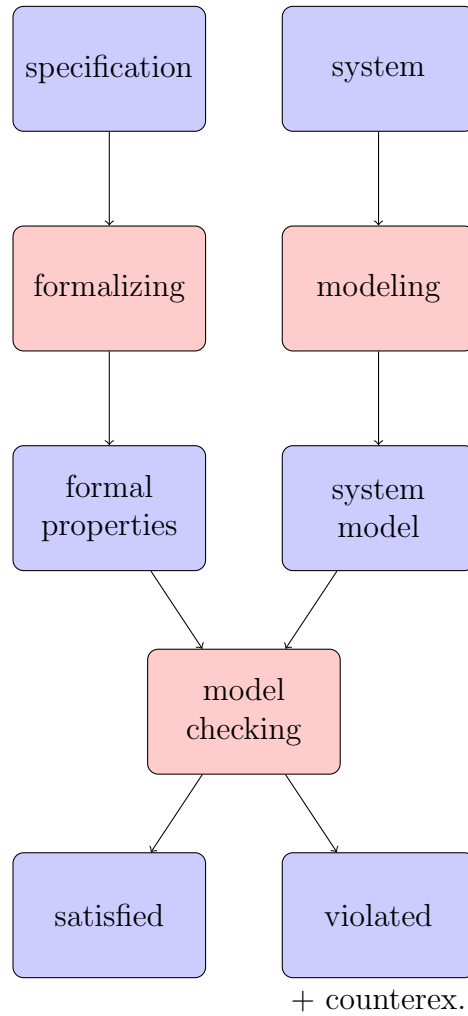


Figure 1: Schematic of the model checking approach [3]

or specifications are involved. Consider, for example, the CSMA/CD protocol, which obviously violates the property "eventually all packets must be sent successfully". However a lot of the packets will eventually be sent successfully, so a qualitative model checker could derive that some packets might be sent successfully. Probabilistic model checking extends conventional model checking with quantitative aspects such as probabilities, so that properties like "at least 95% of all packets must be sent successfully" can be verified. In this thesis we focus on models which expose probabilistic behavior.

1.2 State Space Explosion

One of the fundamental problems model checking encounters is known as the state space explosion problem which refers to the number of states in the system transition graph which often becomes too large to be stored in memory. For example, a system whose state can be represented by n variables from a domain of k possible values has up to k^n states, i.e. the number of states increases exponentially with the number of variables. Current state-of-the-art model checkers are able to handle up to 10^8 to 10^9 states using explicit state-space enumeration. Symbolic data structures, which make use of similarity between states, may be used to diminish this problem for models with structural symmetries.

Nevertheless, even models that can be stored in memory might still be too large to apply model checking, just because the execution of the model checker takes too long to be feasible. This is particularly true for probabilistic model checking, where slow numerical operations occur a lot. Many strategies have been examined in order to enable model checking real-world systems by minimizing the state space.

1.3 Outline

This thesis is divided as follows. Chapter 2 establishes the theoretical background, i.e. the models and concepts we deal with in the thesis. The subsequent chapter 3 gives a detailed description of qualitative model checking. In chapter 4 we present our GPU-based implementation, evaluate the approach and compare it to the StoRM model checker. Finally, we summarize the results of the approach in chapter 5 and give an overview about possible future work.

2 Preliminaries

This chapter introduces Markov decision processes, the logics allowing us to formalize properties as well as the concept of maximal and minimal probabilities. We conclude the chapter with a description of the StoRM model checker.

2.1 Markov Models

Just like transition systems Markov Models comprise states and transitions between them, however the models also contain information about the probability that a certain transition is taken, i.e. the successor state is chosen stochastically. Markov models derive their name from the fact that the stochastic choice in each state only depends on the current state and not on the run thus far, a property known as the memoryless property or Markov property.

One type of Markov models is the Discrete-Time Markov Chain [17]. Many protocols such as the Crowds protocol by Reiter and Rubin [19] and the bounded re-transmission protocol by D'Argenio et al. [7] can be modeled using Discrete-Time Markov Chains (DTMCs).

2.1.1 Discrete-Time Markov Chains

Let 2^A denote the power set of A and $Dist_S$ the set of discrete probability distributions over a set S , i.e. $Dist_S = \{\mu : S \rightarrow [0, 1] \mid \sum_{s \in S} \mu(s) = 1\}$.

Definition 2.1. A Discrete-Time Markov Chain (DTMC) is a tuple

$$\mathcal{D} = (S, \mathbf{P}, s_{init}, AP, L)$$

where

- S is a countable, non-empty set of states,
- $\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability function that assigns to each pair $(s, s') \in S \times S$ of states the probability $\mathbf{P}(s, s')$ of moving from state s to s' in one step such that $\mathbf{P}(s, \cdot) \in Dist_S$,
- $s_{init} \in S$ is the initial state,
- AP is a set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is the labeling function that assigns the possible empty set of atomic propositions $L(s)$ to a state $s \in S$.

In the literature, DTMCs are often defined with an initial distribution $l_{init} : S \rightarrow [0, 1]$ instead of the initial state s_{init} . However, this definition is not more expressive as the one with a single initial state, because models with an initial distribution can be transformed into an equivalent DTMC with only one initial state. Let $\mathcal{D} = (S, \mathbf{P}, l_{init}^{\mathcal{D}}, AP, L)$ be a DTMC with an initial distribution $l_{init}^{\mathcal{D}} \in Dist_S$, then $\bar{\mathcal{D}} = (S_{\mathcal{D}} \cup \{s_{init}^{\mathcal{D}}\}, \bar{\mathbf{P}}_{\mathcal{D}}, s_{init}^{\mathcal{D}}, AP_{\mathcal{D}}, L_{\mathcal{D}})$ for a state $s_{init}^{\mathcal{D}} \notin S_{\mathcal{D}}$ with

$$\bar{\mathbf{P}}_{\mathcal{D}}(s, s') = \begin{cases} l_{init}^{\mathcal{D}}(s') & \text{if } s = s_{init}^{\mathcal{D}} \\ P_{\mathcal{D}}(s, s') & \text{if } s, s' \neq s_{init}^{\mathcal{D}} \\ 0 & \text{otherwise} \end{cases}$$

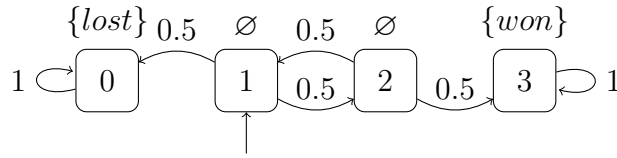
is an equivalent DTMC using the single initial state notation.

A path in \mathcal{D} is an infinite sequence of states $\pi = s_0 s_1 s_2 \dots \in S^{\omega}$ s.t. $s_i \in S_{\mathcal{D}}$ and $P_{\mathcal{D}}(s_i, s_{i+1}) > 0$ for all s_i . With $Paths(\mathcal{D})$ we denote the set of all paths in \mathcal{D} and with $Paths_{fin}(\mathcal{D})$ the set of finite path fragments $s_0 s_1 \dots s_n$ where $n \geq 0$ and $\mathbf{P}_{\mathcal{D}}(s_i, s_{i+1}) > 0$ for all $0 \leq i \leq n$. With $Paths(s)$ and $Paths_{fin}(s)$ we refer to the set of all paths that start in s and the set of all finite path fragments that start in s respectively. If for some states s and t it holds that $\mathbf{P}(s, t) > 0$, we call s a predecessor of t and t a successor of s .

Example 2.1. Consider a gambler in possession of 1\$ who plays until he has either accumulated 3\$ or has no money left. He bets 1\$ in each game and with a probability of 50% gets 2\$ back. This special case of the so called *gambler's ruin problem* is modeled in the DTMC $\mathcal{D} = (S_{\mathcal{D}}, \mathbf{P}_{\mathcal{D}}, s_{init}^{\mathcal{D}}, AP_{\mathcal{D}}, L_{\mathcal{D}})$ depicted in Figure 2 where

- $S_{\mathcal{D}} = \{0, 1, 2, 3\}$,
- $s_{init}^{\mathcal{D}} = 1$,
- $AP_{\mathcal{D}} = \{lost, won\}$,
- $L_{\mathcal{D}}(0) = \{lost\}$, $L_{\mathcal{D}}(3) = \{won\}$, $L_{\mathcal{D}}(1) = L_{\mathcal{D}}(2) = \emptyset$ and, amongst others,
- $\mathbf{P}_{\mathcal{D}}(0, 0) = 1$, $\mathbf{P}_{\mathcal{D}}(1, 0) = 0.5$ and $\mathbf{P}_{\mathcal{D}}(1, 2) = 0.5$.

The probability of winning every single bet, i.e. the path $\pi_1 = 123^{\omega} \in Paths_{\mathcal{D}}$, is given by $Pr_{\mathcal{D}}(\{\pi_1\}) = 0.5 \cdot 0.5 = 0.25$.

Figure 2: DTMC for the *gambler's ruin problem* [16]

2.1.2 Markov Decision Processes

Models are called non-deterministic if there is absolutely no information about which of the possible transitions is scheduled. Non-determinism may be used to model

- interleaving of concurrent actions,
- implementational freedom and
- competition of multiple processes over shared resources.

Although states may have multiple possible successor states, Markov chains can not be used to model non-determinism because the choices are equipped with probability information. Markov Decision Processes (MDPs), which allow for both non-deterministic and probabilistic choices, have been introduced by Bellman [4] and later on discussed by Howard [11] in order to represent non-determinism in Markov models.

Definition 2.2. A (discrete-time) Markov Decision Process (MDP) is a tuple

$$\mathcal{M} = (S, Act, \mathbf{P}, s_{init}, AP, L)$$

where

- S , s_{init} , AP and L are the same as for DTMCs (see Definition 2.1),
- Act is a set of actions, and
- $\mathbf{P} : S \times Act \times S \rightarrow [0, 1]$ is the transition probability function that assigns to each tuple $(s, \alpha, s') \in S \times Act \times S$ the probability $\mathbf{P}(s, \alpha, s')$ of moving from state s to s' with action α s.t. for all $s \in S$ and $\alpha \in Act : \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\}$.

Example 2.2. Consider as an example the two probabilistic processes represented by DTMCs depicted at the top of fig. 3(a), which get executed on a single processing unit. The order in which the steps of the processes are

executed depends on the scheduling in the processing unit. The scheduling used might completely favor one of the two processes and execute it first, but could also switch between the processes after each step. Depicted in fig. 3(b) is the MDP that models the non-deterministic interleaving of the two probabilistic processes. Here, the action α corresponds to executing a step of the first process and β to the second process accordingly.

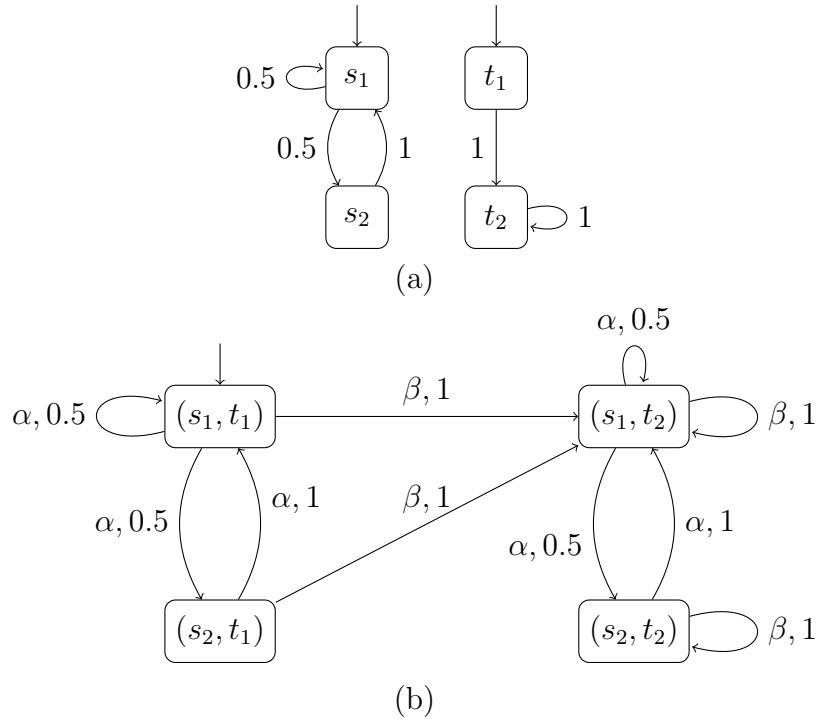


Figure 3: Two concurrent probabilistic processes (a) and their non-deterministic interleaving (b)

Let $\mathcal{M} = (S_{\mathcal{M}}, Act_{\mathcal{M}}, \mathbf{P}_{\mathcal{M}}, s_{init}^{\mathcal{M}}, AP_{\mathcal{M}}, L_{\mathcal{M}})$ be an MDP. $\mathbf{P}(s, \alpha, T)$ lifts the transition probability to sets of states $T \subseteq S_{\mathcal{D}}$ in the canonical way. If for an action $\alpha \in Act_{\mathcal{M}}$ and a state $s \in S_{\mathcal{M}}$, it holds that $\sum_{s' \in S_{\mathcal{M}}} \mathbf{P}_{\mathcal{M}}(s, \alpha, s') = 1$, α is called enabled in s . We call the set of all enabled actions in a state $s \in S_{\mathcal{M}}$ $Act_{\mathcal{M}}(s)$. For a state $s \in S_{\mathcal{M}}$ and an enabled action $\alpha \in Act(s)$, we call the tuple (s, α) a choice in \mathcal{M} . The set of choices in \mathcal{M} is referred to as $Choices(\mathcal{M})$. Every DTMC can be considered an MDP without non-deterministic choices. Similarly, \mathcal{M} is equivalent to a DTMC if in each state there is exactly one action enabled, i.e. $|Act_{\mathcal{M}}(s)| = 1$ for all $s \in S_{\mathcal{M}}$. A path in \mathcal{M} is an alternating sequence of states and actions $\pi = s_0 \alpha_0 s_1 \alpha_1 s_2 \alpha_2 \dots \in (S_{\mathcal{M}} \times Act)^{\omega}$ such that $s_i \in S_{\mathcal{M}}$, $\alpha_i \in Act(s_i)$ and $P(s_i, \alpha_i, s_{i+1}) > 0$ for all $i \in$

\mathbb{N} . We refer to s_i with $\pi[i]$ and, similarly to DTMCs, $Paths_{\mathcal{M}}$ denotes the set of all paths in \mathcal{M} and $Paths_{\mathcal{M}}(s)$ the set of paths that start in s , i.e. $p[0] = s$. If for some states s, t and an action α it holds that $\mathbf{P}(s, \alpha, t) > 0$, we call s a predecessor of t and t a successor of s . The set of probability distributions of actions that are enabled in a state $s \in S_{\mathcal{M}}$ is referred to as $Dist_{\mathcal{M}}(s)$ with $Dist_{\mathcal{M}}(s) = \{\mu \in Dist_S \mid \text{there exists } \alpha \in Act(s) \text{ s.t. } \mathbf{P}_{\mathcal{M}}(s, \alpha, \cdot) = \mu\}$. In order to determine probabilities of paths in MDPs we need to be able to resolve all non-deterministic choices. Therefore, we introduce *schedulers*.

Definition 2.3. Let $\mathcal{M} = (S, Act, \mathbf{P}, s_{init}, AP, L)$ be an MDP. A scheduler $\sigma : S^+ \rightarrow Act$ for \mathcal{M} is a function that maps each finite sequence of states $\omega = s_0 s_1 s_2 \dots s_n$ to one of the actions that are enabled in the last state s_n , i.e. $\sigma(\omega) \in Act_{\mathcal{M}}(s_n)$.

σ is called *memoryless* if the choice of the action depends only on the last state s_n of the sequence, i.e. $\sigma(a_0 a_1 a_2 \dots s_n) = \sigma(b_0 b_1 b_2 \dots s_n)$ for all $a_i \in S, b_j \in S, 0 \leq i < n, 0 \leq j < n$.

A scheduler for an MDP resolves all non-deterministic choices by selecting exactly one of the available actions. Thus, an MDP \mathcal{M} and a scheduler $\sigma_{\mathcal{M}}$ for \mathcal{M} induce a DTMC \mathcal{M}^{σ} . \mathcal{M}^{σ} behaves exactly as \mathcal{M} would if all decisions were made by $\sigma_{\mathcal{M}}$.

Definition 2.4. Let $\mathcal{M} = (S, Act, \mathbf{P}, s_{init}, AP, L)$ be an MDP and σ a scheduler for \mathcal{M} . Then \mathcal{M} and σ induce the DTMC

$$\mathcal{M}^{\sigma} = (S^+, \mathbf{P}_{\sigma}, s_{init}, AP, L')$$

such that for all $\omega = s_0 s_1 s_2 \dots s_n \in S^+, s_{n+1} \in S$

- $P_{\sigma}(\omega, \omega s_{n+1}) = \mathbf{P}(s_n, \sigma(\omega), s_{n+1})$, and
- $L'(\omega) = L(s_n)$.

\mathcal{M}^{σ} induces the probability measure $Pr_{\mathcal{M}^{\sigma}}$ on $Paths_{\mathcal{M}^{\sigma}}$. We refer to this probability measure as $Pr_{\mathcal{M}}^{\sigma}$. Considering a Markov Model \mathcal{M} with state space S , atomic propositions AP and a state $s \in S$, \mathcal{M}_s denotes the Markov Model that results from setting the initial state of \mathcal{M} to s . Similarly, the probability measure is denoted by $Pr_{\mathcal{M}}^s$ in case \mathcal{M} is a DTMC and $Pr_{\mathcal{M}}^{s, \sigma}$ if \mathcal{M} is an MDP and σ a scheduler for \mathcal{M} . DTMCs are considered to be finite if their set of states and atomic propositions are finite, MDPs are further required to have a finite set of actions. Sometimes we need to represent the probability and rate functions of Markov Models with matrices. For DTMCs this is simply the matrix $(\mathbf{P}(s, s'))_{s, s' \in S}$. For a finite MDP \mathcal{M} with

$S_{\mathcal{M}} = \{s_1, \dots, s_n\}$ and $Act_{\mathcal{M}} = \{\alpha_1, \dots, \alpha_n\}$, we represent $\mathbf{P}_{\mathcal{M}}$ with the $n \cdot S \times S$ matrix

$$T = \begin{pmatrix} \mathbf{P}_{\mathcal{M}}(s_1, \alpha_1, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_1, \alpha_1, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_1, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_1, \alpha_n, s_n) \\ \mathbf{P}_{\mathcal{M}}(s_2, \alpha_1, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_n, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_n, \alpha_n, s_n) \end{pmatrix}.$$

2.2 Probabilistic Computational Tree Logic

In order to perform Model Checking we not only need to formalize the model but also the requirements it is supposed to comply with. Temporal logics like Computational Tree Logic (CTL) and Linear Temporal Logic (LTL) [3] may be used to express requirements for qualitative model checking.

However, considering Markov Models we usually want to verify properties with quantitative aspects. While a probabilistic interpretation of LTL exists, we will focus on Probabilistic Computational Tree Logic (PCTL), a branching time logic for DTMCs and MDPs which was introduced by Hansson and Jonsson [9]. Similarly to CTL, there are two types of PCTL formulae - state formulae, whose truth values may be evaluated for states only, and path formulae, which are to be interpreted over paths. Taking a state-based view on Markov Models, we require the formula Ω , which we want to verify, to be a state formula. Thus, model checking a Markov Model \mathcal{M} with initial state s_{init} against Ω comes down to checking whether s_{init} satisfies Ω .

Definition 2.5. PCTL state formulae over a set of atomic propositions AP may be constructed using the following rules

$$\Omega ::= true \mid a \mid \Omega_1 \wedge \Omega_2 \mid \neg \Omega_1 \mid \mathbb{P}_J(\varphi)$$

where $a \in AP$, φ is a PCTL path formula, Ω_1 and Ω_2 are PCTL state formulae and $J = [n, m] \subseteq [0, 1]$ with $n, m \in \mathbb{Q}$. The syntax of PCTL path formulae is given by the following rules:

$$\varphi ::= \mathbf{X} \Omega_1 \mid a \mid \Omega_1 \mathbf{U} \Omega_2 \mid \Omega_1 \mathbf{U}^{\leq k} \Omega_2$$

where Ω_1 and Ω_2 are PCTL state formulae and $k \in \mathbb{N}$.

PCTL extends CTL with the operator $\mathbb{P}_J(\varphi)$. Given a state s , it asserts that the probability mass of the paths which start in s and satisfy the path formula φ lies in the interval J .

Definition 2.6. Let $\mathcal{D} = (S, P, s_{init}, AP, L)$ be a DTMC and $s \in S$ a state of \mathcal{D} . Then the satisfaction relation of state formulae is given by

$$\begin{aligned} s &\models true \\ s &\models a && \text{iff } a \in L(s) \\ s &\models \Omega_1 \wedge \Omega_2 && \text{iff } s \models \Omega_1 \text{ and } s \models \Omega_2 \\ s &\models \neg \Omega_1 && \text{iff } s \not\models \Omega_1 \\ s &\models \mathbb{P}_J(\varphi) && \text{iff } Pr_{\mathcal{D}}(s \models \varphi) \in J \end{aligned}$$

where $Pr_{\mathcal{D}}(s \models \varphi) = Pr_{\mathcal{D}}^s(\{\pi \in Paths_{\mathcal{D}}(s) \mid \pi \models \varphi\})$.

Let $\pi = s_0 s_1 s_2 \dots \in Paths_{\mathcal{D}}$ be a path on \mathcal{D} . Then the satisfaction relation of path formulae is given by

$$\begin{aligned} \pi &\models \mathbf{X} \Omega_1 && \text{iff } \pi[1] \models \Omega_1 \\ \pi &\models \Omega_1 \mathbf{U} \Omega_2 && \text{iff } \exists j \geq 0. (\pi[j] \models \Omega_2 \wedge \forall 0 \leq k < j. (\pi[k] \models \Omega_1)) \\ \pi &\models \Omega_1 \mathbf{U}^{\leq k} \Omega_2 && \text{iff } \exists 0 \leq j \leq k. (\pi[j] \models \Omega_2 \wedge \forall 0 \leq l < j. (\pi[l] \models \Omega_1)) \end{aligned}$$

We write $\mathcal{D} \models \Omega$ if $s_{init} \models \Omega$ for a state formula Ω .

The PCTL satisfaction relation for MDP differs from the one for DTMC because it needs to account for the different schedulers. The $\mathbb{P}_J(\varphi)$ operator now asserts that the probability mass of all paths satisfying φ lies in the interval J for every possible resolution of non-deterministic choices, i.e. for all schedulers.

Definition 2.7. Let $\mathcal{M} = (S, Act, s_{init}, AP, L)$ be a MDP and $s \in S$ a state of \mathcal{M} . The satisfaction relation of state formulae is the same as for DTMCs (see 2.6) except for

$$s \models \mathbb{P}_J(\varphi) \quad \text{iff } Pr_{\mathcal{M}}^{\sigma}(s \models \varphi) \in J \text{ for all schedulers } \sigma \text{ for } \mathcal{M}$$

where $Pr_{\mathcal{D}}^{\sigma}(s \models \varphi) = Pr_{\mathcal{D}}^{s, \sigma}(\{\pi \in Paths_{\mathcal{D}}(s) \mid \pi \models \varphi\})$. The satisfaction relation of path formulae is identical to the one for DTMCs.

With $Sat_{\mathcal{M}}$, or briefly $Sat(\Phi)$, we denote the set $\{s \in S \mid s \models \Phi\}$ of states that satisfy Φ .

Inspecting MDPs we are often interested in whether a property holds even in the best/worst case of nondeterministic choices, which leads to the notion of *maximal and minimal probabilities* [6].

Definition 2.8. Let $\mathcal{M} = (S, Act, s_{init}, AP, L)$ be a MDP and φ be a PCTL path formula. The minimal and maximal probabilities of φ on \mathcal{M} are defined as follows:

$$Pr_{max}^{\mathcal{M}}(\varphi) = \sup_{\sigma \in Schedulers(\mathcal{M})} (Pr^{\mathcal{M}\sigma}(\varphi))$$

$$Pr_{min}^{\mathcal{M}}(\varphi) = \inf_{\sigma \in \text{Schedulers}(\mathcal{M})} (Pr^{\mathcal{M}_\sigma}(\varphi))$$

where $\text{Schedulers}(\mathcal{M}) \subseteq S^+ \rightarrow \text{Act}$ denotes the set of all schedulers for \mathcal{M} .

Note that for a scheduler σ the probability measure $Pr^{\mathcal{M}_\sigma}(\varphi)$ can be determined by calculating $Pr^{\mathcal{D}}(\varphi)$ where \mathcal{D} is the DTMC induced by applying σ to \mathcal{M} as explained earlier.

2.2.1 Qualitative Fragment

Definition 2.9. State formulae from the *qualitative fragment* of PCTL over a set of atomic propositions AP are constructed using the following rules:

$$\Omega ::= \text{true} \mid a \mid \Omega_1 \wedge \Omega_2 \mid \neg \Omega_1 \mid \mathbb{P}_{>0}(\varphi) \mid \mathbb{P}_{=1}(\varphi)$$

where $a \in AP$, Ω_1 and Ω_2 are state formulae and φ is a path formula formed according to the following grammar:

$$\varphi ::= \mathbf{X} \Omega_1 \mid \Omega_1 \mathbf{U} \Omega_2$$

where Ω_1 and Ω_2 are state formulae of the qualitative fragment of PCTL.

We call the formulae that belong to the qualitative fragment *qualitative PCTL formulae*.

While only > 0 and $= 1$ are allowed as probability bounds due to the definition, $= 0$ and < 1 may be derived like this:

$$\mathbb{P}_{=0}(\varphi) \equiv \neg \mathbb{P}_{>0}(\varphi) \text{ and } \mathbb{P}_{<1}(\varphi) \equiv \neg \mathbb{P}_{=1}(\varphi).$$

Note that qualitative PCTL formulae do not contain the bounded until operator $\mathbf{U}^{\leq k}$.

2.3 StoRM Framework

We use the Stochastic Reward Model Checker (StoRM) framework as the foundation of our work. It is the successor project of the MRMC framework by Katoen et al. [14], however it has not been published to the public yet. StoRM is able to parse and check various types of models like DTMCs and MDPs and is easily extendable with additional parsers and model checker implementations.

2.4 The GPGPU approach

Most operations of modern computers are executed on the Central Processing Unit (CPU), an electronic circuit that is able to perform basic arithmetic, logical, and input/output operations. As the name suggests the CPU is the fundamental part of the computer and thus designated and optimized for the execution of basic programs. While modern CPUs often allow for parallelism by using multiple cores, they usually are part of a Single Instruction, Single Data (SISD) architecture, i.e. they execute a single list of instructions, each of which operates on a single data element.

While many tasks like text processing fit perfectly to the SISD architecture in general and to the execution on CPUs in particular, other tasks, e.g. graphics related programs like computer games and video encoders, often incorporate application of the same instruction on multiple data points. Consider for example, changing the brightness of an image. Images are usually represented as a matrix of points, each of which consists of the brightness of the red, green and blue portion of its color. In order to change an image's brightness the three color brightness values of all points have to be adjusted by the same amount. A CPU (considering a single core) would have to iterate over all points and adjust the three values sequentially. To solve problems like this more efficiently Graphics Processing Units (GPUs) have been introduced to perform a single instruction on multiple data at once.

Originally the sole purpose of GPUs was to output images to the display attached to it. To perform fast manipulations and processing of the input images matrix and vector manipulations such as matrix-vector-multiplication were supported natively on GPUs. Naturally, these operations are also heavily used in scientific computation and form the foundation of many mathematical algorithms. Thus, many scientific programs have been implemented to operate on GPUs in order to achieve faster execution times. An implementation of the LU factorization from 2005 was one of the first scientific programs that performed faster on the GPU than on the CPU [8].

2.4.1 CUDA

In the early days of General Purpose Computation on Graphics Processing Unit (GPGPU) one had to make use of the graphical primitives as supplied by the two most common APIs for GPUs, OpenGL and DirectX, to implement mathematical algorithms for the graphics device. The high efforts that had to be pursued for those implementations were resolved with the rise of languages and APIs aimed specifically at general purpose programming for GPUs, such as Accelerator [21].

The two APIs for GPGPU that are most commonly used for scientific programming today are NVIDIA’s Compute Unified Device Architecture (CUDA) and OpenCL from the Khronos Group. Both allow the programmer to ignore the underlying graphical primitives and offer concepts more common to scientific programming. While OpenCL is available for many CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors, CUDA is limited to GPUs manufactured by NVIDIA. However, this broader diversity of architectures which OpenCL targets results in noticeable performance losses of OpenCL in comparison to CUDA (on identical NVIDIA hardware). Use of OpenCL on NVIDIA GPUs may lead to performance losses of 5 to 50% [13]. Because of this and the fact that efforts have already been made by Berger [5] to port parts of the StoRM framework to make use of NVIDIA GPUs through the use of the CUDA API, we have chosen CUDA as the foundation of the work made for this thesis and OpenCL will not be considered further.

The CUDA platform allows us to implement highly parallel parts of our program in so called *kernels*, which are run on the graphics device. Kernels differ from regular functions in that they are executed by a multitude of CUDA threads. The sequential part of the program is, as is usually the case, executed on the CPU and interspersed with invocations of kernels. A kernel is executed in a blockwise manner, where each block is composed of the same number of threads. The blocks form the so called *grid*, whose size needs to be specified by the programmer upon invocation of the kernel from the host, i.e. the program running on the CPU. Both a thread’s position in its block as well as the blocks position in the grid can be identified through a one, two or three dimensional index. This allows for more convenient mapping of a thread to the element in a domain such as vector, matrix or volume.

As depicted in fig. 4 CUDA-compliant GPUs are composed of an array of multi-threaded Streaming Multiprocessors (SMs). Upon invocation of a kernel, the thread blocks are distributed between the available SMs for concurrent execution. All threads of a single block are executed concurrently on a single SM. A multiprocessor executes threads in groups of 32 parallel threads called *warps*.

Each thread in a warp has its own instruction address counter and register state and can branch and execute independently. However, the threads of a warp execute the instructions in lockstep, and thus to achieve maximal performance the threads should follow the same execution path. If threads of a single warp diverge (e.g. due to a conditional branch), the warp executes each branch path one after another, whilst disabling the threads that diverged to another branch. This only applies to a single warp, different warps always execute individually regardless of whether they have a common execution

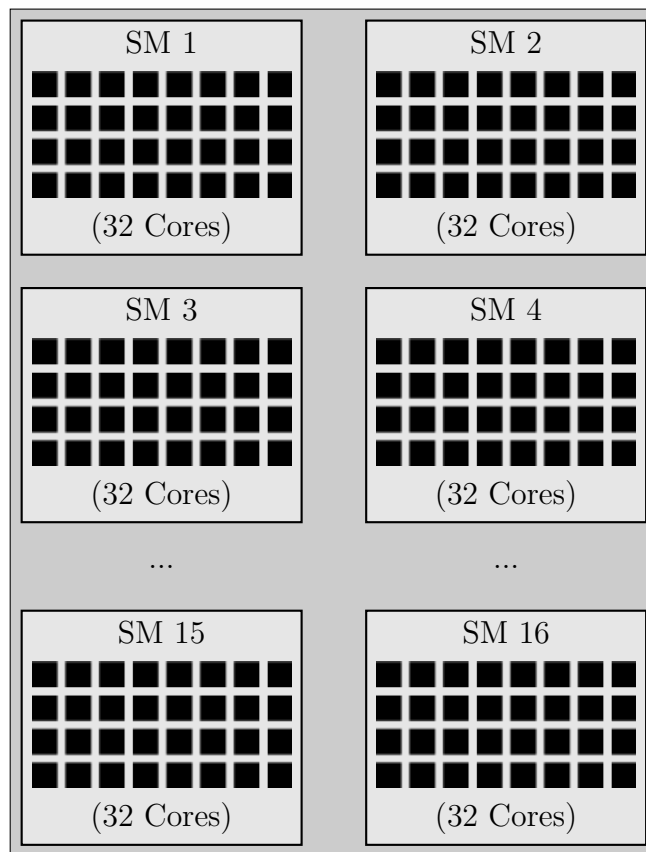


Figure 4: Schematic of the CUDA GPU architecture

path.

All threads executing a kernel have access to the off-chip *global memory* space. In addition to its fast on-chip registers, each thread has a private off-chip *local memory* space (see Lee et al. [15]). CUDA-compliant GPU devices have a separate memory space from the host CPU. The CUDA API supplies functions for explicit management of the graphics device's memory, which, amongst others, allow for transfer of data from the host's to the graphics device's memory and vice versa. Because transferring data between the respective memory spaces is time-expensive, the transferred data should be kept to minimal size.

3 Qualitative Model Checking

In this chapter, we describe the process of model checking qualitative PCTL formulae on MDPs and introduce the algorithms needed for the *precomputation* step.

3.1 Algorithms

Checking whether an MDP \mathcal{D} satisfies a PCTL formula $P = \mathbb{P}_J(\Phi\mathbf{U}\Psi)$ usually happens in two steps. In the first step, the states $s \in S_{\mathcal{D}}$ are partitioned into the following sets:

- $S_0 = \{ s \in S_{\mathcal{D}} \mid s \models \mathbb{P}_{=0}(\Phi\mathbf{U}\Psi) \}$, the set of all states from which no possible path satisfies $\Phi\mathbf{U}\Psi$,
- $S_1 = \{ s \in S_{\mathcal{D}} \mid s \models \mathbb{P}_{=1}(\Phi\mathbf{U}\Psi) \}$, the set of all states from which every possible path satisfies $\Phi\mathbf{U}\Psi$ (note that $\{ s \in S_{\mathcal{D}} \mid s \models \Psi \} \subseteq S_1$), and
- $S_? = S_{\mathcal{D}} \setminus (S_0 \cup S_1)$, the rest of $S_{\mathcal{D}}$.

At this point, we have achieved knowledge of the exact probabilities for the states in S_0 and S_1 as 0 and 1 respectively. Several options exist to compute the remaining values for the states from $S_?$ as the second step, among them value iteration by Bellman [4] and Shapley [20] and a reduction to a linear equation system (Howard [11], van Nunen [22], Puterman [18]). Note that performing the first step suffices for model checking of formulae from the qualitative fragment of PCTL. Additionally, calculation of S_0 is needed to achieve a unique solution in the second step, while calculating S_1 will reduce the size of the numerical computation problem to solve in the second step.

In this thesis we will focus solely on the first step, commonly referred to as the *precomputation* of PCTL model checking on MDPs. Taking into account minimal probabilities as well we arrive at the different cases depicted in table 1, which shows the names of the algorithms we use to calculate the state sets for the corresponding cases. For example, the **Prob0A** algorithm may be used to calculate the states s for which $Pr_{max}^{\mathcal{D}}(s \models \varphi) = 0$ holds. The resulting states satisfy $\mathbb{P}_{=0}(\varphi)$.

For the S_0 cases, it turned out to be easier to calculate the complementing $S \setminus S_0$ and complement it afterwards. To account for this, the **ProbGreater0E** and **ProbGreater0A** algorithms have been introduced. They can be used to determine the states that have a minimal (and maximal respectively) probability greater than 0 of satisfying the formula in question. The algorithms

	Pr_{min}	Pr_{max}
S_0	Prob0E	Prob0A
S_1	Prob1A	Prob1E

Table 1: Different precomputation cases and the corresponding algorithms

are related to each other in the following way:

$$\text{Prob0A}(\mathcal{D}, \text{Sat}(\Phi_1), \text{Sat}(\Phi_2)) = S \setminus \text{ProbGreater0E}(\mathcal{D}, \text{Sat}(\Phi_1), \text{Sat}(\Phi_2)) \quad (1)$$

$$\text{Prob0E}(\mathcal{D}, \text{Sat}(\Phi_1), \text{Sat}(\Phi_2)) = S \setminus \text{ProbGreater0A}(\mathcal{D}, \text{Sat}(\Phi_1), \text{Sat}(\Phi_2)) \quad (2)$$

In the following we describe the algorithms and illustrate their functionality using a small example.

3.1.1 ProbGreater0E

The computation of the states for which a scheduler exists s.t. the probability of satisfying a formula $\varphi = \Phi_1 \mathbf{U} \Phi_2$ is greater than 0 can be performed by means of iterative computations of state sets. The idea is to successively determine those states s for which $Pr_{max}(s \models \Phi_1 \mathbf{U}^{\leq k} \Phi_2) > 0$ for some $k \in \mathbb{N}$. Initially, the states R of \mathcal{D} which fulfill Φ_2 are determined. Obviously, all paths starting in those states fulfill $\varphi = \Phi_1 \mathbf{U} \Phi_2$ which means that for every scheduler the states $s \in R$ have a probability of 1 to fulfill φ . Let s be a state contained in R and s' a predecessor of s which fulfills Φ_1 , i.e. $s' \models \Phi_1 \wedge \exists \alpha \in \text{Act}. \mathbf{P}(s', \alpha, s) > 0$. Because $Pr_{max}(s \models \Phi_1 \mathbf{U} \Phi_2) = 1 > 0$ and $s' \models \Phi_1$, it holds that $Pr_{max}(s' \models \Phi_1 \mathbf{U} \Phi_2) > 0$, too. Thus, for each $s \in R$, we can add the predecessors of s which fulfill Φ_1 to R . At this point R consists of the states s for which $Pr_{max}(s \models \Phi_1 \mathbf{U}^{\leq 1} \Phi_2) > 0$, and as a consequence it holds that $R \subseteq \{s \in S \mid Pr_{max}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0\}$. We repeat this procedure and determine the sets of states $R_k = \{s \in S_{\mathcal{D}} \mid Pr_{max}(s \models \Phi_1 \mathbf{U}^{\leq k} \Phi_2) > 0\}$ for increasing values of $k \in \mathbb{N}$ until no new states can be added to R_k , i.e. $R_k = R_{k+1}$. These steps are summarized in algorithm 1. By making use of eq. (1) we can calculate $S_0^{max} = \{s \mid \mathbb{P}_{=0}(\varphi)\}$ by complementing the result of the ProbGreater0E algorithm:

$$S_0^{max} = S \setminus \text{ProbGreater0E}(\mathcal{D}, \text{Sat}(\Phi_1), \text{Sat}(\Phi_2)).$$

Example 3.1. For our examples we consider the MDP depicted in Figure 5. As the algorithms do not depend on the transition probabilities and rely

Algorithm 1 Computing the set of states s with $Pr_{max}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0$

```

1: procedure ProbGreater0E( $\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)$ )
2:    $R \leftarrow Sat(\Phi_2)$ 
3:    $done \leftarrow false$ 
4:   while  $done = false$  do
5:      $R' \leftarrow R \cup \{s \in Sat(\Phi_1) \mid \exists \alpha \in Act(s). \exists s' \in R. \mathbf{P}(s, \alpha, s') > 0\}$ 
6:     if  $R' = R$  then
7:        $done \leftarrow true$ 
8:     end if
9:      $R \leftarrow R'$ 
10:  end while
11:  return  $R$ 
12: end procedure

```

simply on whether a transition has a non-zero probability, the actual probabilities have been omitted from the figure to improve readability and all the plotted transitions have a probability > 0 . Obviously, if for one action in a state only a single transition exists, this transition has a probability of 1. Interested in the states which have a maximal probability of 0 to fulfill $\varphi = a\mathbf{U}b$, we make use of eq. (1) and apply algorithm 1 to determine the states which have a non-zero probability of fulfilling the formula in the minimal case, i.e. those from which at least one path starts that fulfills the formula.

Upon initialization of the algorithm, R is set to $Sat(b) = \{s_4\}$. In the first iteration of the loop, we add s_2 and s_5 to R , because both fulfill a and from both a transition to s_4 exists. In the next iteration s_0 and s_3 are added to R because they are predecessors of s_2 and fulfill a . At this point R contains s_0, s_2, s_3, s_4 and s_5 . Because $s_6 \not\models a$ and no transition from s_1 to a state of R exists, no additional states are added to R in the next iteration of the loop. Thus, the algorithm terminates with the result $\{s_0, s_2, s_3, s_4, s_5\}$.

Consequently, for each of the states s_0, s_2, s_3, s_4 and s_5 at least one scheduler exists s.t. the probability of fulfilling $a\mathbf{U}b$ is greater than 0. Additionally, due to eq. (1) we know that no path starting in a state from $S \setminus \{s_0, s_2, s_3, s_4, s_5\} = \{s_1, s_6\}$ fulfills φ regardless of the scheduling, i.e. $Pr_{max}(1 \models \varphi) = Pr_{max}(s_6 \models \varphi) = 0$.

3.1.2 ProbGreater0A

The states for which the probability of fulfilling some formula $\Phi_1 \mathbf{U} \Phi_2$ is non-zero for every possible scheduler can be determined in a similar fashion as

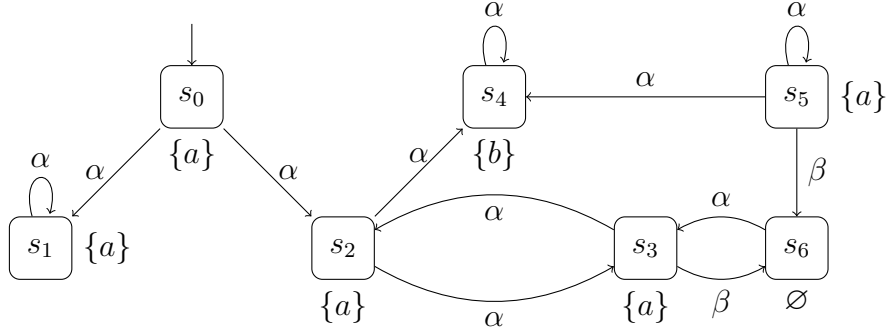


Figure 5: Example MDP

the `ProbGreater0E` algorithm. The algorithm now works as follows. We first determine the states $s \in R \subseteq S$ for which $s \models \Phi_2$ holds. Again, those states also satisfy $\mathbf{P}_{>0}(\Phi_1 \mathbf{U} \Phi_2)$ regardless of the resolution of non-determinism, because all paths starting in a state $s \in R$ start in a state which fulfills Φ_2 and thus the paths fulfill $\Phi_1 \mathbf{U} \Phi_2$. Let $s \in S$ be a state of \mathcal{D} s.t. s satisfies Φ_1 and for every action which is enabled in s the probability of moving to a state $s' \in R$ is greater than 0, i.e. $s \models \Phi_1 \wedge \forall \alpha \in Act(s). \exists s' \in R. \mathbf{P}(s, \alpha, s') > 0$. Then it holds that $Pr_{min}(s \models \Phi_1 \mathbf{U}^{\leq 1} \Phi_2) > 0$ and thus also $Pr_{min}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0$. Therefore we can add all such states s to R and it follows that $(R \subseteq \{s \in S \mid Pr_{min}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0\})$. The state sets $R_k = \{s \in S_{\mathcal{D}} \mid Pr_{min}(s \models \Phi_1 \mathbf{U}^{\leq k} \Phi_2) > 0\}$ for increasing values of $k \in \mathbb{N}$ are determined by repeating the step described above. The algorithm returns when no new states can be added to R_k , i.e. $R_k = R_{k+1}$ for some k . This procedure is depicted in algorithm 2. Analogously to `Prob0A` and `ProbGreater0E` we can determine the states for which the probability of fulfilling $\Phi_1 \mathbf{U} \Phi_2$ is 0 for every possible scheduler by making use of eq. (2):

$$\text{Prob0E}(\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)) = S \setminus \text{ProbGreater0A}(\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)).$$

Example 3.2. As for the `ProbGreater0E` algorithm, we use `ProbGreater0A` to determine the states of \mathcal{D} from fig. 5 which have a non-zero probability of fulfilling $a \mathbf{U} b$. Again, R is initialized to $Sat(b) = \{s_4\}$. In the first iteration, s_2 is added to R , because it fulfills a , only the action α is enabled and $\mathbf{P}(s_2, \alpha, s_4) > 0$. The state s_5 , on the contrary, is not added to R , because β is enabled in it and $\mathbf{P}(s_5, \beta, s_4) = 0$. In the next iteration s_0 is added to R at which point no other states can be added, because s_0 has no predecessors and β is enabled in s_3 but $\mathbf{P}(s_3, \beta, s_2) = 0$. Thus the algorithm returns $\{s_0, s_2, s_4\}$.

Algorithm 2 Computing the set of states s with $Pr_{min}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0$

```

1: procedure PROBGREATER0A( $\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)$ )
2:    $R \leftarrow Sat(\Phi_2)$ 
3:    $done \leftarrow false$ 
4:   while  $done = false$  do
5:      $R' \leftarrow R \cup \{s \in Sat(\Phi_1) \mid \forall \alpha \in Act(s). \exists s' \in R. \mathbf{P}(s, \alpha, s') > 0\}$ 
6:     if  $R' = R$  then
7:        $done \leftarrow true$ 
8:     end if
9:      $R \leftarrow R'$ 
10:  end while
11:  return  $R$ 
12: end procedure

```

This means that $Pr_{max}(s_0 \models \varphi) = Pr_{max}(s_2 \models \varphi) = Pr_{max}(s_4 \models \varphi) > 0$. From eq. (2) we can deduce that $Pr_{min}(s \models \varphi) = 0$ for all states $s \in S \setminus \{s_0, s_2, s_4\} = \{s_1, s_3, s_5, s_6\}$.

3.1.3 Prob1E

So far we have presented the algorithms which calculate the states for which the minimal (maximal) probability of satisfying a formula $\Phi_1 \mathbf{U} \Phi_2$ is non-zero. To determine S_1 we have to adopt a different approach. The states of an MDP \mathcal{M} for which a scheduler exists s.t. the probability of fulfilling $\Phi_1 \mathbf{U} \Phi_2$ equals 1 can be calculated by a double fixed point algorithm. The idea is to successively remove those states s for which $Pr_{max}(s \models \Phi_1 \mathbf{U} \Phi_2) < 1$, i.e. those which under no scheduler σ satisfy $\mathbf{P}(\Phi_1 \mathbf{U} \Phi_2) = 1$. Initially we consider the whole set of states $R = S$. Then we determine the states $s \in R$ which fulfill Φ_2 as well as the ones which fulfill Φ_1 and for which a scheduler σ exists s.t. all transitions stay within R and at least one path of \mathcal{M}^σ starting in s eventually reaches Φ_2 . Note that, because initially R contains every state, this first step is equivalent to performing **ProbGreater0E**. All other states, i.e. those which under all schedulers have a probability of 0 to fulfill the formula, are then removed from R . In order to make sure that for every state $s \in R$ it holds that either s satisfies Φ_2 or a scheduler exists s.t. all transitions from s stay in R , this procedure is repeated until no additional states can be removed from R . These steps are depicted in algorithm 3.

Example 3.3. Again we apply the **Prob1E** algorithm to the MDP from Figure 5, checking the formula $\varphi = a \mathbf{U} b$. R gets initialized to the state

Algorithm 3 Computing the set of states s with $Pr_{max}(s \models \Phi_1 \mathbf{U} \Phi_2) = 1$

```
1: procedure PROB1E( $\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)$ )
2:    $R \leftarrow S$ 
3:    $done \leftarrow false$ 
4:   while  $done = false$  do
5:      $R' \leftarrow Sat(\Phi_2)$ 
6:      $done' \leftarrow false$ 
7:     while  $done' = false$  do
8:        $R'' \leftarrow R' \cup \{s \in Sat(\Phi_1) \mid \exists \alpha \in Act(s). (\exists s' \in R'. \mathbf{P}(s, \alpha, s') > 0$ 
           $\wedge (\forall s' \in S. \mathbf{P}(s, \alpha, s') > 0 \rightarrow s' \in R))\}$ 
9:       if  $R'' = R'$  then
10:         $done' \leftarrow true$ 
11:       end if
12:        $R' \leftarrow R''$ 
13:     end while
14:     if  $R' = R$  then
15:        $done \leftarrow true$ 
16:     end if
17:      $R \leftarrow R'$ 
18:   end while
19:   return  $R$ 
20: end procedure
```

set of \mathcal{D} , i.e. $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, which is depicted in Figure 6(1) with a hatched background. In the first iteration of the outer loop, R' gets initialized to $\{s_2\}$. Because at this point R contains every state of \mathcal{D} , all predecessors of s_4 which are labeled with a , i.e. s_2 and s_5 , are added to R' in the first iteration of the inner loop. In the next inner iteration s_0 and s_3 are added to R' and thus no new predecessors fulfilling a exist. The final states of R' are depicted in Figure 6(1) with a black background. Next we set R set to $R' = \{s_0, s_2, s_3, s_4, s_5\}$, as can be seen in depicted in Figure 6(2) with a hatched background. In the next iteration of the outer loop R' is reset to $Sat(a) = \{s_4\}$. In the first iteration of the inner loop, s_2 and s_5 are added to R' again. Because $s_3 \models a$ and $\mathbf{P}(s_3, \alpha, s_2) = 1$, s_3 is added to R' in the next iteration. However, although s_0 is a predecessor of $s_2 \in R'$ for action α , it is not added to R' , because $s_1 \notin R$ and $\mathbf{P}(s_0, \alpha, s_1) > 0$. No other predecessors of R' , which now consists of the states shown with a black background in depicted in Figure 6(2), can be added, and thus R is set to $R' = \{s_2, s_3, s_4, s_5\}$. As shown in depicted in Figure 6(3) the next iteration of the outer loop does not lead to any changes and thus the algorithm returns $\{s_2, s_3, s_4, s_5\}$.

This shows that $Pr_{max}(s \models \varphi) = 1$ for all states $s \in \{s_2, s_3, s_4, s_5\}$.

3.1.4 Prob1A

Algorithm 4 summarizes the steps of determining the states s from an MDP \mathcal{M} which have a probability of 1 to fulfill a formula of the form $\Phi_1 \mathbf{U} \Phi_2$ under every scheduler. We start with the set of all states $R = S$ and successively remove the states s for which a scheduler σ exists s.t. $Pr^{\mathcal{M}\sigma}(\Phi_1 \mathbf{U} \Phi_2) < 1$. First, the algorithm determines the set $R' = Sat(\Phi_2)$ of states which satisfy Φ_2 . Successively we add those states $s \in Sat(\Phi_1)$ to R' for which all transitions from the enabled actions stay in R and for all schedulers σ at least one path of \mathcal{M}^σ starting in s eventually reaches Φ_2 . This is essentially the same as performing **ProbGreater0A**, due to R containing all states of \mathcal{M} at this point. The other states $s \in R \setminus R'$ are then removed from R . Because now R might contain states from which a transition leads to a state not contained in R , we repeat this procedure until no additional states can be removed from R .

Example 3.4. To show how the **Prob1A** algorithm works, we apply it to the MDP from Figure 5, checking the formula $a \mathbf{U} b$. Just like in **Prob1E**, initially R is set to the state set of \mathcal{D} , i.e. $\{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$. In the first iteration of the outer loop, R' gets initialized to $\{s_2\}$. Because at this point R contains every state of \mathcal{D} , all states which have a non-zero probability of

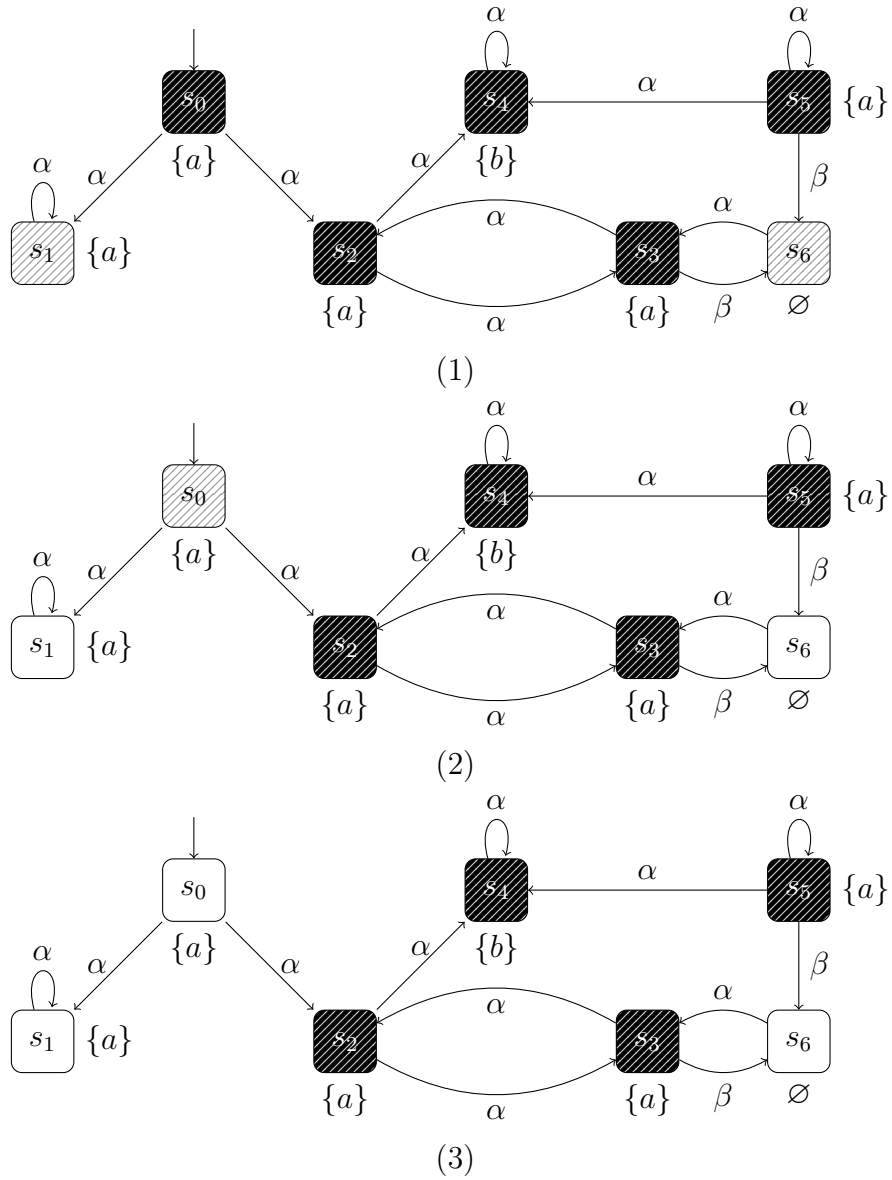


Figure 6: Initial R (hatched) and final R' (white on black) in the three iterations of the outer loop in example 3.3

Algorithm 4 Computing the set of states s with $Pr_{min}(s \models \Phi_1 \mathbf{U} \Phi_2) = 1$

```
1: procedure PROB1A( $\mathcal{D}, Sat(\Phi_1), Sat(\Phi_2)$ )
2:    $R \leftarrow S$ 
3:    $done \leftarrow false$ 
4:   while  $done = false$  do
5:      $R' \leftarrow Sat(\Phi_2)$ 
6:      $done' \leftarrow false$ 
7:     while  $done' = false$  do
8:        $R'' \leftarrow R' \cup \{s \in Sat(\Phi_1) \mid \forall \alpha \in Act(s). (\exists s' \in R'. \mathbf{P}(s, \alpha, s') > 0$ 
           $\wedge (\forall s' \in S. \mathbf{P}(s, \alpha, s') > 0 \rightarrow s' \in R))\}$ 
9:       if  $R'' = R'$  then
10:         $done' \leftarrow true$ 
11:       end if
12:        $R' \leftarrow R''$ 
13:     end while
14:     if  $R' = R$  then
15:        $done \leftarrow true$ 
16:     end if
17:      $R \leftarrow R'$ 
18:   end while
19:   return  $R$ 
20: end procedure
```

reaching s_4 for all actions and which fulfill a , in our case the single state s_2 , are added to R' in the first iteration of the inner loop. On the contrary, s_5 is not added to R' because $s_6 \notin R'$ and $\mathbf{P}(s_5, \beta, s_6) > 0$. In the next inner iteration s_0 is added to R' , however s_3 is not added, because β is enabled in s_3 and $\mathbf{P}(s_3, \beta, s_6) = 1$. No other predecessors of R' labeled with a exist and R is set to $R' = \{s_0, s_2, s_4\}$. In the next iteration of the outer loop R' is reset to $Sat(a) = \{s_4\}$. For the same reason as above s_5 can not be added to R' . Additionally, because $s_3 \notin R$ and $\mathbf{P}(s_2, \alpha, s_3) > 0$, s_2 is not added to R' again. No other predecessors of s_4 exist, and thus R is set to $R' = \{s_4\}$. The next iteration of the outer loop gives the same result and thus the algorithm returns $\{s_4\}$.

This shows that $Pr_{min}(s_4 \models aUb) = 1$.

3.2 Input

The algorithms we implemented take an MDP and a PCTL path formula of the form $\Phi_1 \mathbf{U} \Phi_2$ as their input. Remember that an MDP is a tuple $\mathcal{M} = (S, Act, \mathbf{P}, s_{init}, AP, L)$. We implement the model checking in a recursive fashion and thus pass $Sat(\Phi_1) \subseteq S$ and $Sat(\Phi_2) \subseteq S$ to the algorithms instead of the corresponding state formulae. In the StoRM framework \mathcal{M} is assumed to be finite and thus the state and action space is also finite and we can refer to the states and actions simply by numbering them. We represent state subsets $A \subseteq S$ by binary arrays of length $|S|$, i.e. $a \in \{0, 1\}^{|S|}$. An array a corresponds to set A through $\forall s_i \in S. a[i] = 1 \Leftrightarrow s_i \in A$. We refer to this kind of array as a *bit vector*.

Because we pass the sets of satisfying states, we don't need to pass the labeling function L to the algorithms. The initial state s_{init} is not of relevance either due to the need of checking the probability for all states of M . As shown earlier $\mathbf{P}_{\mathcal{M}}$ can be represented by the $n \cdot S \times S$ matrix

$$T = \begin{pmatrix} \mathbf{P}_{\mathcal{M}}(s_1, \alpha_1, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_1, \alpha_1, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_1, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_1, \alpha_n, s_n) \\ \mathbf{P}_{\mathcal{M}}(s_2, \alpha_1, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_2, \alpha_n, s_n) \\ \vdots & & \vdots \\ \mathbf{P}_{\mathcal{M}}(s_n, \alpha_n, s_1) & \dots & \mathbf{P}_{\mathcal{M}}(s_n, \alpha_n, s_n) \end{pmatrix}.$$

However, as the PCTL can not formulate statements about the actions and possibly a lot of actions are not enabled in each state from the MDP, we

only include columns for the enabled actions of each state. This reduces the representation of $\mathbf{P}_{\mathcal{M}}$ to the $|Choices(\mathcal{M})| \times S$ matrix

$$T' = (\mathbf{P}(s, \alpha, s'))_{(s, \alpha) \in Choices(\mathcal{M}), s' \in S}.$$

The sets of rows belonging to choices from the same state are called row groups. We need to specify the different row groups and to which states they belong. This can be done by a vector $v \in \mathbb{N}^{|S_{\mathcal{M}}+1|}$ of row indices, s.t. for each $s_i \in S_{\mathcal{M}}$ it holds that the choices (s_i, α) with $\alpha \in Act(s)$ are represented in the rows $T'_{v_i} \dots T'_{v_{i+1}-1}$, which form the row group belonging to s_i .

Example 3.5. Considering even distribution of the transition probabilities in each state, the transition probability function $\mathbf{P}_{\mathcal{M}}$ of the MDP \mathcal{M} from fig. 5 is represented by:

$$M^{\mathbf{P}} = \begin{array}{c} \begin{array}{c} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{array} \left(\begin{array}{c|c|c|c|c|c|c} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\ \hline 0 & 0.5 & 0.5 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0.5 & 0.5 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \right) \end{array}$$

To make it easier to understand the matrix, we have added the starting states of the transitions to the left and the target states to the top of the matrix. Additionally, vertical lines partition the matrix to illustrate which starting state the choices of the respective rows belong to. The belonging row group index vector of $M^{\mathbf{P}}$ is $v = (0 \ 1 \ 2 \ 3 \ 5 \ 6 \ 8)^T$.

To retrieve the transitions for state s_3 , we first look at $v_4 = 3$ and $v_5 = 5$. This tells us that there are two different actions enabled in s_3 , whose transitions are represented in the fourth and fifth row of the matrix. Thus we know that for the first action there's only one transition leading to s_2 , because $M_{4,3}^{\mathbf{P}}$ equals 1. And for the second action, a transition with probability 1.0 leads to s_6 .

The most common way to store matrices is in two-dimensional arrays where each element of the matrix is mapped to an element of the array. However, this so called dense matrix representation of an n -by- m matrix requires $n \cdot m$ memory cells of which each is able to hold a single value of the

matrix. Because this is an infeasible amount of required memory for many applications other matrix representations are being used for different kinds of special matrices like diagonal and symmetric matrices. The transition probability function matrix T contains 0-entries where the probability of the corresponding transition is zero. Experience shows that most real-world models have relatively few transitions in comparison to the amount of their states. For example, the *coin2_6* model from our case study (see section 4.2.1) has 784 states and only 1452 of the $784 \cdot 784 = 614.656$ possible transitions.

A matrix in which most elements are zero is called a *sparse matrix*. Various memory representations have been proposed for sparse matrices (see Bai et al. [2, pp. 315 ff.]), among them Compressed Row Storage (CRS), which was chosen for the StoRM framework. The CRS format stores an m -by- n matrix A using three one-dimensional arrays *values*, *col_indicators*, and *row_pointers*. The *values* array consists of the non-zero values of A in left-to-right top-to-bottom order. Let NNZ denote the amount of elements in *values*, i.e. the amount of non-zero elements of A . Like the *values* array, the *col_indicators* array is of length NNZ . Each element of it specifies the column index of the corresponding element in *values*. The *row_pointers* array is of length $m + 1$ and stores the indices of the *values* array that start a new row, i.e. $a_{i,j} = values[x] \Rightarrow row_pointers[i] \leq x < row_pointers[i + 1]$. The last element of *row_pointers* must equal NNZ . Formally, it holds that

$$a_{i,j} = values[x] \Leftrightarrow (row_pointers[i] \leq x < row_pointers[i + 1]) \wedge (col_indicators[x] = j)$$

for all $1 \leq i \leq m, 1 \leq j \leq n$. The CRS representation allows us to store matrices in memory of size $2 \cdot NNZ + m + 1$ instead of $m \cdot n$ like the dense matrix representation.

Example 3.6. Consider as an example the 4×4 matrix

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 2 & 0 \end{pmatrix}.$$

Using CRS the matrix can be represented by the following arrays:

$$values = \begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 2 \end{pmatrix}, col_indicators = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 2 \end{pmatrix}, row_pointers = \begin{pmatrix} 0 \\ 1 \\ 3 \\ 4 \\ 4 \end{pmatrix}.$$

Note that we used 0-based indexing for the column indicators and the row pointers. If we want to retrieve $A_{2,3}$ we first look at the row pointers of the second and third row. As $row_pointers_2 = 1$ and $row_pointers_3 = 3$, we know that we have to look at the second and third element of $col_indicators$. The value of $col_indicator_{s_3}$ equals the index of the third column and thus we find the value of $A_{2,3}$ at the third element of the $values$ array. If we wanted to retrieve the value of $A_{2,2}$ instead, we would find that neither the second nor the third element of $col_indicators$ equals the index of the second column and thus, we know that $A_{2,2} = 0$.

For model checking of qualitative properties we are only interested in whether a transition between two states is possible or not. Thus we can simplify the matrix representation to

$$T = \begin{pmatrix} E_{\mathcal{M}}(s_1, a_1, s_1) & \dots & E_{\mathcal{M}}(s_1, a_1, s_n) \\ \vdots & & \vdots \\ E_{\mathcal{M}}(s_1, a_n, s_1) & \dots & E_{\mathcal{M}}(s_1, a_n, s_n) \\ E_{\mathcal{M}}(s_2, a_1, s_1) & \dots & E_{\mathcal{M}}(s_2, a_n, s_n) \\ \vdots & & \vdots \\ E_{\mathcal{M}}(s_2, a_n, s_1) & \dots & E_{\mathcal{M}}(s_2, a_n, s_n) \\ \vdots & & \vdots \end{pmatrix}$$

where $E_{\mathcal{M}} : S \times Act \times S \rightarrow \{0, 1\}$ and

$$\forall s, s' \in S, a \in Act. E_{\mathcal{M}}(s, a, s') = 1 \Leftrightarrow \mathbf{P}_{\mathcal{M}}(s, a, s') > 0.$$

Converting A to CRS representation would lead to the $values$ array containing only elements of value 1. We can make use of this information by not actually storing the array in memory and simply interpreting it as full of 1s. This reduces the memory requirements for the probability function to $NNZ + m + 1$.

For parts of the algorithms it makes sense to iterate over the predecessors of a state. Because the predecessors of some state s can not be determined efficiently using the CRS representation of the transition matrix, we explicitly calculate a matrix representing the *backwards* transitions of the MDP. This matrix can be obtained by transposing the transition matrix $M^{\mathbf{P}}$. However we need to handle the row groups when transposing the matrix to get a $|S| \times |S|$ matrix. We achieve this by treating each row group of n rows as a single row of $[0, 1]^n$ vectors. For example, the backwards transition matrix of the MDP \mathcal{M} from fig. 5 (assuming even probability distribution)

3 QUALITATIVE MODEL CHECKING

is represented by:

$$B^{\mathbf{P}} = \begin{array}{c}
 \begin{array}{ccccccc}
 & s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 \\
 s_0 & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
 s_1 & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
 s_2 & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
 s_3 & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
 s_4 & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
 s_5 & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0.5 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
 s_6 & \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix}
 \end{array}
 \end{array}$$

For the CRS representation of $B^{\mathbf{P}}$ it is sufficient to have multiple elements of *col_indicators* of the corresponding row contain the same index and adjust *values* and *row_pointers* accordingly.

4 Implementation and evaluation

4.1 Implementation

For optimal performance we restrict our selves to pure C - with the CUDA extensions - and make no use of C++ features such as classes in the kernel functions, i.e. the programs that are executed on the GPU. Because the rest of the StoRM framework is implemented in C++ and follows the Object-oriented programming (OOP) paradigm in wide parts of the application, we have to prepare the input data for our kernels. Structs are the usual choice for implementing class-like behavior in C and thus we choose them to implement the data structure for the sparse matrices and bit vectors. The *bit_vector* struct simply holds the number of elements n and an array of m *chars* such that $n \leq m \cdot 8$. The *sparse_matrix* struct holds both the number of non-negative elements *NNZ* as well as the number of rows R and the *column_indicators* and *row_pointers* arrays. As explained earlier, we do not need to store the values of the matrix in our struct.

The CPU-based implementation of the *precomputation* algorithms uses a depth-first search (DFS) approach that manages a stack of states that are to be visited. Because we want to make use of the parallelization available on the GPU we have to take another approach, in which CUDA thread handles exactly one state of the MDP. This is where the massive parallelization features of GPUs come into play. For example, for the MDP of *csm4.4* (see section 4.2.2) we launch a total of $|S_{csm4.4}| \approx 6.24 \cdot 10^6$ CUDA threads. We launch 512 threads per block, which has shown to be the best performing setting, and as many blocks as necessary to have one thread per state. Note that this means that we may launch more threads than necessary. Those threads will not perform any work but don't slow the computation down either, because they are running in parallel with the threads actually performing work. We pass a bit vector to the kernel that specifies how many states exist and for which of them work has to be performed.

Each thread calculates the state it is supposed to work on using the following formula:

$$state_index = block_id \cdot block_size + thread_id$$

One-dimensional indexing is sufficient, because we map the threads to elements of the bit vector. Notice that the blocks and threads are 0-indexed. The needed information about the grid is passed to our kernels automatically by the extensions of CUDA to the C language.

A thread performs work if their state index is smaller than the number of states and the corresponding state is enabled in the input bit vector,

otherwise they will just idle and wait for the other threads to finish their work. As fig. 7 depicts, we manage the main loop of the algorithms from our host device, the CPU. This means that every kernel call performs exactly one iteration of the respective loop. After all threads of the kernel grid terminate, we check whether the algorithm is finished. If so, we return the result to the caller, otherwise we invoke the kernel again. Due to the lack of global synchronization (between the blocks) this can not be done in the kernels and instead needs to be done on the CPU.

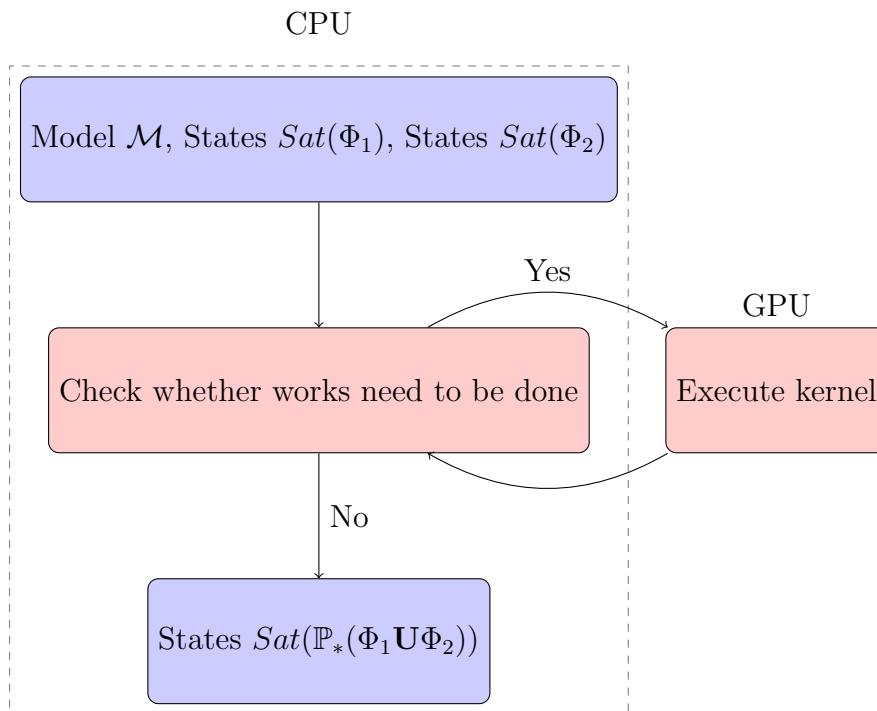


Figure 7: Schematic of the general algorithm execution

Because the memory accessed by the GPU and the CPU is not the same, we need to explicitly copy data from the former to the latter and vice versa. Before executing the first kernel of an algorithm, we need to copy the necessary data to the memory of the graphics card. After a kernel finished executing we usually copy the data that is needed to check the termination of the algorithm back to the CPU in order to perform the check. Remember that copying between the two memories is expensive and needs to be reduced to the absolute minimum.

The kernel which is used to perform `ProbGreater0E` is depicted in algorithm 5. Initially, the CPU prepares the input for the kernels. In particular, it creates a bit vector sat_{Φ_1} that represents the set of states $s \in Sat(\Phi_1)$.

The working set w , which holds the states for which work needs to be performed, and the result set res , which is sequentially expanded to hold $\{s \in S | Pr_{max}^{\mathcal{M}}(s \models \Phi_1 \mathbf{U} \Phi_2) > 0\}$, are both initialized to represent the states $s \in Sat(\Phi_2)$. This means that the trivial part of the work where parallelization is of little help, i.e. marking all states $s \in Sat(\Phi_2)$ as *true* in res , is performed on the CPU instead of the GPU. Additionally, this removes the need to separately copy the states which satisfy Φ_2 to the graphics device and thus improves the runtime of our GPGPU-aided approach. Note that the kernel for **ProbGreater0E** only needs the backwards transitions, i.e. the transposed transition matrix, and not the original transitions.

After the CPU has prepared the input data and copied it to the graphics device, it invokes the kernel for the first time. This means that at least $|S_{\mathcal{M}}|$ CUDA threads are created, each of which executes the kernel. As explained earlier, each thread first determines which state it is supposed to perform work on (cf. line 2). The threads idle if the corresponding state is not in the working set w . Otherwise, it removes the state from the working set, because the work will be performed by the thread itself. In lines 5 to 13, the threads iterate over the target states of the backward transitions, i.e. the predecessors, of their corresponding states. If a predecessor p satisfies Φ_1 , i.e. the corresponding value in sat_{Φ_1} is *true*, and is not yet included in the result set res , the thread sets the corresponding value of p to *true* in both res and the working set w .

When all threads have executed those steps, control is given back to the host CPU. The CPU copies back the working set w to its memory and checks whether at least one state's value in w is *true*. If so, the kernel is invoked again to continue working on the problem. Otherwise, the CPU copies back the result bit vector res and returns it to the caller as the result.

Algorithm 6 depicts the steps the kernel performs for **Prob1E**. The CPU manages the double fixed point loop of the algorithm. Just as before, the CPU prepares the input data for the kernel. In each iteration of the outer loop the working set w and the result set res are initialized as $Sat(\Phi_2)$, whereas the set of current states $curr$ is initially set to S . The CPU performs the inner loop by invoking the kernel until $w = \emptyset$. Each CUDA thread determines the state it needs to work on as explained earlier. If that state is enabled in w , it then iterates over the predecessors p of the state. If $p \in Sat(\Phi_1)$ and all successors of p are in $curr$, the thread adds p to the result res and to the working set w . After each iteration of the inner loop the $curr$ and res bit_vectors are copied to the host's memory and compared to each other. If they are equal, the algorithm has finished and the result can be returned to the caller. Otherwise, we set $curr$ to res and reset res to $Sat(\Phi_2)$. Then the next iteration of the inner loop gets started by invoking the kernel again.

Algorithm 5 Kernel for ProbGreaterOE

Input:

bit_vector w // working set
bit_vector res // intermediate result $res \subseteq Sat(\mathbb{P}_{>0}(\Phi_1 \mathbf{U} \Phi_2))$
bit_vector sat_{Φ_1} // $Sat(\Phi_1)$
sparse_matrix $backwardTransitions$ // transposed transition matrix

```
1: procedure PROBGREATEROEKERNEL
2:    $state \leftarrow block\_id \cdot block\_size + thread\_id$ 
3:   if  $state < w.length \wedge w[state] = true$  then
4:      $w[state] \leftarrow false$ 
5:      $i \leftarrow backwardTransitions.row\_pointers[state]$ 
6:     while  $i < backwardTransitions.row\_pointers[state + 1]$  do
7:        $predecessor \leftarrow backwardTransitions.col\_indicators[i]$ 
8:       if  $sat_{\Phi_1}[predecessor] = true$ 
9:          $\wedge res[predecessor] = false$  then
10:         $res[predecessor] \leftarrow true$ 
11:         $w[predecessor] \leftarrow true$ 
12:      end if
13:       $i \leftarrow i + 1$ 
14:    end while
15:  end if
16: end procedure
```

Algorithm 6 Kernel for Prob1E**Input:**

```

    bit_vector  $w$  // working set
    bit_vector  $curr$  // intermediate result of last iteration
    bit_vector  $res$  // intermediate result  $res \subseteq Sat(\mathbb{P}_{>0}(\Phi_1 \mathbf{U} \Phi_2))$ 
    bit_vector  $sat_{\Phi_1}$  //  $Sat(\Phi_1)$ 
    sparse_matrix  $transitions$  // transition matrix
    int_vector  $choices$  // indices of the row groups
    sparse_matrix  $backwardTransitions$  // transposed transition matrix
1: procedure PROB1EKERNEL
2:    $state \leftarrow block\_id \cdot block\_size + thread\_id$ 
3:   if  $state < w.length \wedge w[state] = true$  then
4:      $w[state] \leftarrow false$ 
5:      $i \leftarrow backwardTransitions.row\_pointers[state]$ 
6:     while  $i < backwardTransitions.row\_pointers[state + 1]$  do
7:        $predecessor \leftarrow backwardTransitions.col\_indicators[i]$ 
8:       if  $sat_{\Phi_1}[predecessor] = true$ 
9:          $\wedge res[predecessor] = false$  then
10:           $row \leftarrow choices[predecessor]$ 
11:          while  $row < choices[predecessor + 1]$  do
12:            if ALLSUCCESSORSINCURR( $predecessor$ ) then
13:               $res[predecessor] \leftarrow true$ 
14:               $w[predecessor] \leftarrow true$ 
15:              break
16:            end if
17:             $row \leftarrow row + 1$ 
18:          end while
19:          end if
20:           $i \leftarrow i + 1$ 
21:        end while
22:      end if
23: end procedure
24: procedure ALLSUCCESSORSINCURR( $s$ )
25:    $succ \leftarrow transitions.row\_pointers[s]$ 
26:   while  $succ < transitions.row\_pointers[s + 1]$  do
27:     if  $\neg curr[succ]$  then
28:       return  $false$ 
29:     end if
30:      $succ \leftarrow succ + 1$ 
31:   end while
32:   return  $true$ 
33: end procedure

```

Because they are quite similar to the presented kernels, we do not include code listings and descriptions of `ProbGreater0A` and `Prob1A`.

4.2 Case studies

In order to test the performance of our GPU-based implementation we applied it to 3 widely used case studies. We compared our results with the default CPU-based implementation of StoRM and visualize the results by plotting the relative runtime difference against the number of transitions in the models. The properties we checked, as well as tables showing the specifics of the models checked and the measured runtime spent in the function implementing the algorithms for qualitative model checking, can be found in Appendix A. For each case study we include a diagram of the time reduction factor, which illustrates the ratio of runtime needed in the qualitative model checking algorithms in the GPGPU-aided implementation to that in the CPU-based implementation. Depending on the formulas checked, not all of the four algorithms are executed in each case study and therefore included in the diagram. For easy identification of good and bad ratios between the parallel and the sequential version, a dashed line in each of the diagrams marks the break-even points. Factors greater than 1 indicate that our GPGPU-aided approach outperformed the sequential CPU variant. For case studies in which the time reduction factor varies greatly between the different algorithms, we have split the diagram into multiple ones, each of which shows time reduction factors for a different subset of the algorithms. Note that the model size does not necessarily increase from left to right, as we ordered the models on the horizontal axis according to the first parameter.

The benchmarks have been performed on a personal computer running Arch Linux with Kernel version 3.18.6. The machine features a AMD Phenom II X6 1045T processor, 8 Gigabytes of DDR2-RAM and an Nvidia GTX 750 TI graphics card with 2 Gigabyte graphic memory.

4.2.1 coin case study

For our first benchmark we used the shared coin protocol created by Aspnes and Herlihy [1] for randomized consensus. The protocol models a set of N asynchronous processes which are supposed to make a decision between 0 and 1. The processes communicate by incrementing and decrementing a shared global counter, which is initially set to 0. Each process repeatedly flips an unbiased coin and increments or decrements the global counter based on the result. The coin is shared between the process and the order in which the processes gain access to the coin is nondeterministic. After modification of

the global counter the process checks whether the counters value is at most $-N \cdot K$ or at least $N \cdot K$, where $K > 1$ is a parameter of the protocol, and if so, chooses 0 or 1 accordingly. Otherwise the process flips the coin again, until it is able to make a decision. We refer to the MDP that models the shared coin protocol with parameters N and K as *coin N_K* .

As is depicted in fig. 8, our CUDA implementation of the `ProbGreater0E` and `ProbGreater0A` algorithms is unable to outperform the sequential equivalents for all of the model sizes. While the GPGPU-aided version takes 1.4 up to 2.5 times as long as the default StoRM implementation for the model sizes $N = 6, K \in \{2, 4, 6\}$, it performs significantly worse for the other model sizes.

However, in fig. 9 we see that the `Prob1A` and `Prob1E` algorithms as implemented in CUDA result in huge runtime reductions for greater model sizes. In particular, `Prob1A` runs between 3 and 13 times faster on the GPU than on the CPU for some model sizes.

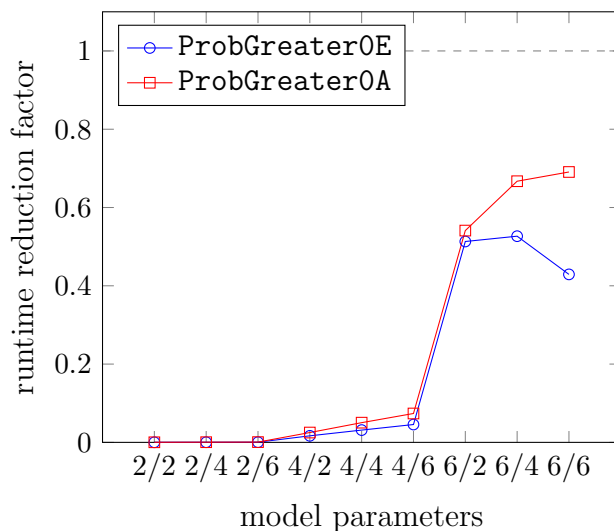


Figure 8: Shared coin protocol results for `ProbGreater0E` and `ProbGreater0A`

4.2.2 csma case study

This case study models the Carrier Sense Multiple Access/Collision Detection (CSMA/CD) protocol for asynchronous media access control which got standardized for ethernet in IEEE 802.3. The model describes N data stations which try to send a single message on a shared medium. Before a

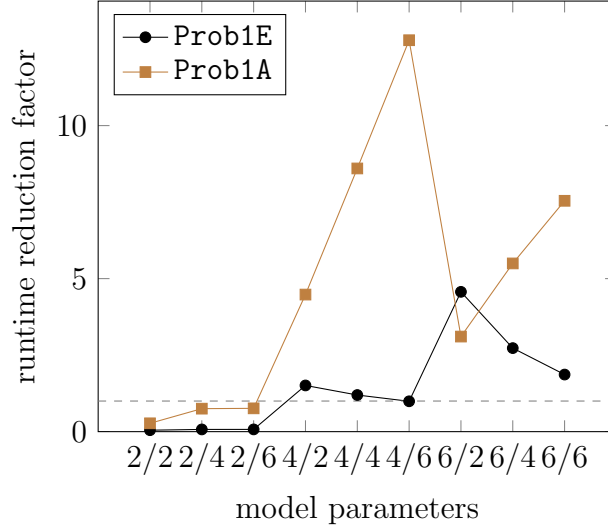


Figure 9: Shared coin protocol results for Prob1E and Prob1A

process actually sends the message to the medium, it checks whether the medium is free. If so, it starts sending the message. While sending the process constantly checks the medium for occurring collisions. If a collision happens, the process notifies the other processes of the collision using the so called *jam* signal, stops the transmission and waits for a random time interval called *backoff* before retrying the transmission. The range of possible backoff values B depends on the number C of occurred collisions and is given by $0 \leq B \leq K^C$ time slots, where $K > 0$ is another parameter of the model. We refer to the MDP that models the CSMA/CD protocol with parameters N and K as *csmaN_K*.

Figure 10 illustrates that with our approach to GPGPU-aided qualitative model checking the ProbGreater0A and Prob1A algorithms perform worse than the CPU-based for all the model sizes. Note that the ProbGreater0E and the Prob1E algorithms are not used for checking the formula we chose for our case study.

4.2.3 leader case study

The third case study models the asynchronous leader election protocol introduced by Itai and Rodeh [12]. The protocol models N processors which try to select a distinct processor called the *leader* by communicating over an asynchronous ring. The processors start as active and perform the following steps until they become inactive:

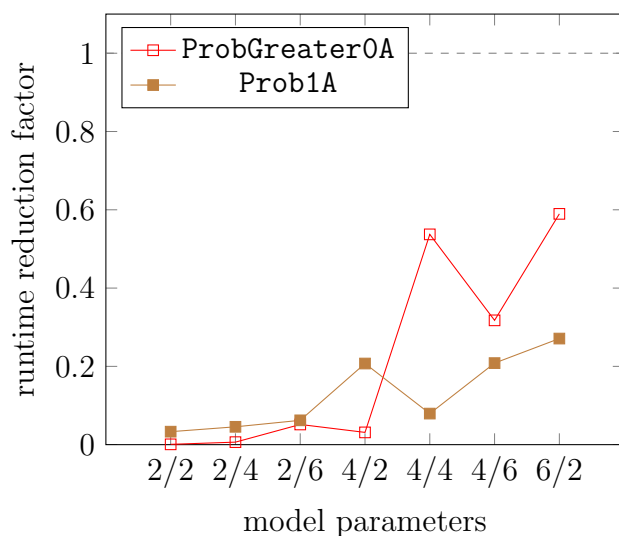


Figure 10: CSMA/CD protocol results for ProbGreater0A and Prob1A

- Make a probabilistic choice between 0 and 1, each with probability $\frac{1}{2}$, and send the result to the next processor in the ring,
- If the result was 0 and the choice of the preceding active processor was 1, become inactive.
- Determine whether the processor is the last active one in the ring by sending a counter via the ring. If so, the processor is the *leader* and the protocol is finished.

Because the ring is asynchronous, i.e. each processor has its own clock, the order in which the messages are sent via the ring is non-deterministic. We refer to the MDP that models the leader protocol with parameter N as *leaderN*. Again, it is to be noted that the formula we checked results in only performing the ProbGreater0A and Prob1A algorithms.

As can be seen in fig. 11, the GPGPU-aided implementation approaches the runtime of the CPU-based version quickly, and outperforms it in execution of Prob10A on *leader6* and of ProbGreater0A on *leader7*.

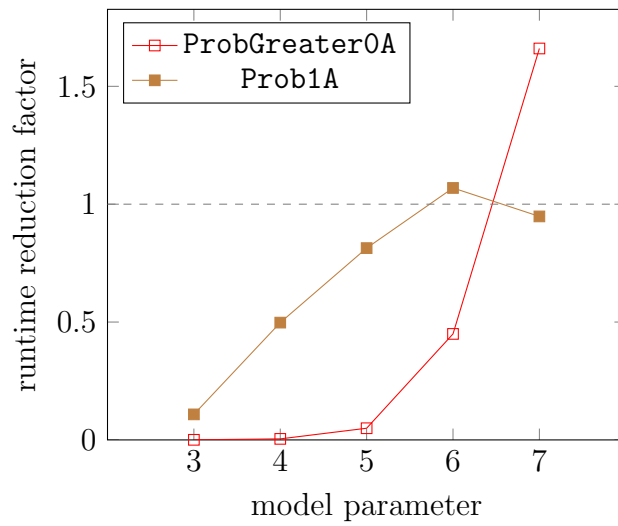


Figure 11: Asynchronous leader election protocol results for ProbGreater0A and Prob1A

5 Conclusion

5.1 Summary & Evaluation

In this thesis, we presented a parallel implementation of the qualitative model checking algorithms for GPUs. In particular, we integrated our CUDA-based implementation into the model checker tool StoRM. We pointed out differences between the massively parallel approach and a traditional sequential implementation. In order to evaluate the impact of our approach we conducted several case studies. Our experiments include a comparison with StoRM’s CPU-based implementation of the algorithms for qualitative model checking.

The experiments indicate that our GPGPU-aided implementation can, depending on the model to check, lead to a runtime reduction by more than a factor of 10 and consistently outperform the sequential implementation for big models. However, we found that the initial overhead of copying data to the graphics device and invoking kernels makes our method infeasible for smaller examples. This was to be expected though, as usually hundreds or thousands of CUDA threads are required to achieve noticeable runtime improvements. Due to the lack of a mechanism for global synchronisation between the threads we need to return control to the CPU multiple times during execution of a single algorithm, which inhibits us of exploiting the full potential of parallel computation on the GPU.

Overall, we have found that performing qualitative model checking on graphics devices may result in promising performance gains. However, the effectiveness of the approach varies greatly between the different algorithms and models.

5.2 Future Work

In this section, we present possible future work building on our approach. First of all, there are some possibilities to improve the applicability and performance of our prototype:

1. To be able to apply our approach to models which take up more memory than available on commonly used graphics devices, a decomposition of the model into strongly connected components (SCCs) could be implemented. The algorithms for qualitative model checking could then be executed on the SCCs separately. This has been previously done by Berger [5].

2. The kernels and the data could be restructured further to decrease the divergence between the threads of the warps. In particular, the threads in the working sets could be resorted such that states with similar numbers of predecessors will be handled by threads of the same warp. This could greatly improve the runtime of the algorithms.
3. The CUDA platform allows the host to manage multiple GPUs and execute kernels on the different devices concurrently. The algorithms could be adapted to be run on multiple devices at once, which would make it possible to model check greater models due to more graphics memory.

Additionally, more case studies could be conducted to evaluate the performance of the approach. A larger set of data points could make it possible to implement a heuristic which chooses between the sequential CPU-based and the GPGPU-aided implementation based on the model's attributes.

Appendix

A Case studies

A.1 General

- The models we checked are bundled with the StoRM framework.
- All times are given in milliseconds.

A.2 Properties

coin case study

The properties we verified are the following:

- What is the maximal probability that the protocol eventually finishes and at least two processes chose different values, i.e.

$$\mathbb{P}_{=?}^{max}(true\mathbf{U}finished \wedge \neg agree),$$

- What is the minimal probability that the protocol eventually finishes and all coins show the value 0, i.e.

$$\mathbb{P}_{=?}^{min}(true\mathbf{U}finished \wedge all_coins = 0),$$

- What is the minimal probability that the protocol eventually finishes and all coins show the value 1, i.e.

$$\mathbb{P}_{=?}^{min}(true\mathbf{U}finished \wedge all_coins = 1),$$

where

- *finished* specifies the states in which the protocol has finished,
- *agree* specifies the states in which all processes agree, and
- *all_coins = i* for $i \in \{0, 1\}$ specifies the states in which the value of all coins is 0 or 1, respectively.

csma case study

The properties we verified are the following:

- What is the minimal probability that all packets get delivered and no collision occurs at the maximal backoff K , i.e.

$$\mathbb{P}_{=?}^{min}(true\mathbf{U}\neg collision_max_backoff \wedge all_delivered).$$

leader case study

The properties we verified are the following:

- What is the maximal probability that a leader gets elected successfully, i.e.

$$\mathbb{P}_{=?}^{min}(true\mathbf{U}elected).$$

APPENDIX

A.3 Tables

coin case study

ProbGreater0E						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	272	492	0.19	1,130.05	$1.65 \cdot 10^{-4}$
2	4	528	972	0.17	557.75	$3.04 \cdot 10^{-4}$
2	6	784	1,452	0.25	548.88	$4.53 \cdot 10^{-4}$
4	2	22,656	75,232	9.35	558.97	$1.67 \cdot 10^{-2}$
4	4	43,136	$1.44 \cdot 10^5$	18.46	588.61	$3.14 \cdot 10^{-2}$
4	6	63,616	$2.13 \cdot 10^5$	27.73	608.24	$4.56 \cdot 10^{-2}$
6	2	$1.26 \cdot 10^6$	$6.24 \cdot 10^6$	687.26	1,339.25	0.51
6	4	$2.38 \cdot 10^6$	$1.18 \cdot 10^7$	1,316.31	2,499.06	0.53
6	6	$3.49 \cdot 10^6$	$1.74 \cdot 10^7$	1,965.68	4,580.51	0.43

Prob1E						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	272	492	0.29	6.21	$4.73 \cdot 10^{-2}$
2	4	528	972	0.96	13.23	$7.25 \cdot 10^{-2}$
2	6	784	1,452	2.05	27.86	$7.35 \cdot 10^{-2}$
4	2	22,656	75,232	369.04	244.45	1.51
4	4	43,136	$1.44 \cdot 10^5$	1,474.61	1,232.17	1.2
4	6	63,616	$2.13 \cdot 10^5$	3,135.51	3,160.54	0.99
6	2	$1.26 \cdot 10^6$	$6.24 \cdot 10^6$	59,561.29	13,037.67	4.57
6	4	$2.38 \cdot 10^6$	$1.18 \cdot 10^7$	$2.01 \cdot 10^5$	73,554.1	2.73
6	6	$3.49 \cdot 10^6$	$1.74 \cdot 10^7$	$4.19 \cdot 10^5$	$2.24 \cdot 10^5$	1.87

ProbGreater0A						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	272	492	0.26	1,091.88	$2.41 \cdot 10^{-4}$
2	4	528	972	0.52	1,116.92	$4.64 \cdot 10^{-4}$
2	6	784	1,452	0.78	1,099.51	$7.06 \cdot 10^{-4}$
4	2	22,656	75,232	28.51	1,134.97	$2.51 \cdot 10^{-2}$
4	4	43,136	$1.44 \cdot 10^5$	58.33	1,158.88	$5.03 \cdot 10^{-2}$
4	6	63,616	$2.13 \cdot 10^5$	89.26	1,209.92	$7.38 \cdot 10^{-2}$
6	2	$1.26 \cdot 10^6$	$6.24 \cdot 10^6$	1,784.26	3,297.6	0.54
6	4	$2.38 \cdot 10^6$	$1.18 \cdot 10^7$	3,611.45	5,412.97	0.67
6	6	$3.49 \cdot 10^6$	$1.74 \cdot 10^7$	5,303.06	7,677.12	0.69

Prob1A						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	272	492	1.02	3.75	0.27
2	4	528	972	3.45	4.6	0.75
2	6	784	1,452	4.16	5.46	0.76
4	2	22,656	75,232	188.34	42.05	4.48
4	4	43,136	$1.44 \cdot 10^5$	632.83	73.58	8.6
4	6	63,616	$2.13 \cdot 10^5$	1,345.2	105.17	12.79
6	2	$1.26 \cdot 10^6$	$6.24 \cdot 10^6$	15,621.19	5,027.9	3.11
6	4	$2.38 \cdot 10^6$	$1.18 \cdot 10^7$	50,385.47	9,166.43	5.5
6	6	$3.49 \cdot 10^6$	$1.74 \cdot 10^7$	$1.01 \cdot 10^5$	13,392.13	7.54

Table 2: Shared coin protocol

APPENDIX

csma case study

ProbGreater0A						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	1,038	1,282	0.92	1,135.15	$8.13 \cdot 10^{-4}$
2	4	7,958	10,594	3.57	555.29	$6.42 \cdot 10^{-3}$
2	6	66,718	93,072	30.65	594.5	$5.16 \cdot 10^{-2}$
3	2	36,850	55,862	18.21	581.41	$3.13 \cdot 10^{-2}$
3	4	$1.46 \cdot 10^6$	$2.4 \cdot 10^6$	794.75	1,479.36	0.54
4	2	$7.62 \cdot 10^5$	$1.33 \cdot 10^6$	416.89	1,312.71	0.32
4	4	$6.24 \cdot 10^6$	$1.21 \cdot 10^7$	3,731.07	6,328.33	0.59

Prob1A						
N	K	states	transitions	time (CPU)	time (GPU)	factor
2	2	1,038	1,282	0.46	13.91	$3.32 \cdot 10^{-2}$
2	4	7,958	10,594	3.73	81.96	$4.56 \cdot 10^{-2}$
2	6	66,718	93,072	31.05	499.72	$6.21 \cdot 10^{-2}$
3	2	36,850	55,862	18.22	87.88	0.21
3	4	$1.46 \cdot 10^6$	$2.4 \cdot 10^6$	830	10,458.95	$7.94 \cdot 10^{-2}$
4	2	$7.62 \cdot 10^5$	$1.33 \cdot 10^6$	415.46	1,994.18	0.21
4	4	$6.24 \cdot 10^6$	$1.21 \cdot 10^7$	15,755.61	58,131.35	0.27

Table 3: CSMA/CD protocol

leader case study

ProbGreater0A					
N	states	transitions	time (CPU)	time (GPU)	factor
3	364	654	0.24	565.68	$4.33 \cdot 10^{-4}$
4	3,172	7,144	2.5	565.14	$4.42 \cdot 10^{-3}$
5	27,299	74,365	28.39	573.09	$4.95 \cdot 10^{-2}$
6	$2.38 \cdot 10^5$	$7.61 \cdot 10^5$	314.93	700.42	0.45
7	$2.1 \cdot 10^6$	$7.71 \cdot 10^6$	3,362.51	2,025.01	1.66

Prob1A					
N	states	transitions	time (CPU)	time (GPU)	factor
3	364	654	0.18	1.68	0.11
4	3,172	7,144	1.91	3.84	0.5
5	27,299	74,365	19.24	23.65	0.81
6	$2.38 \cdot 10^5$	$7.61 \cdot 10^5$	200.7	187.77	1.07
7	$2.1 \cdot 10^6$	$7.71 \cdot 10^6$	1,836.54	1,937.07	0.95

Table 4: Asynchronous leader election protocol

Bibliography

- [1] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.
- [2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*, volume 11. Siam, 2000.
- [3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [4] Richard Bellman. A markovian decision process. Technical report, DTIC Document, 1957.
- [5] Philipp Berger. GPU-aided model checking of markov decision processes. Bachelor thesis, RWTH Aachen University, September 2014.
- [6] Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513. Springer, 1995.
- [7] Pedro R D’Argenio, Bertrand Jeannet, Henrik E Jensen, and Kim G Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, pages 39–56. Springer, 2001.
- [8] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.
- [9] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.
- [10] Tony Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
- [11] Ronald A Howard. Dynamic programming. *Management Science*, 12(5):317–348, 1966.
- [12] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

BIBLIOGRAPHY

- [13] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.
- [14] Joost-Pieter Katoen, Ivan S Zapreev, Ernst M Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker MRMC. In *Proceedings of the Sixth International Conference on the Quantitative Evaluation of Systems, QEST '09*, pages 167–176, Los Alamitos, September 2009. IEEE Computer Society Press.
- [15] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009.
- [16] Hung Q Ngo. CSE 694: Probabilistic analysis and randomized algorithms, 2011. URL <https://www.cse.buffalo.edu/~hungngo/classes/2011/Spring-694/lectures/dtmc.pdf>. [Online; accessed 16-April-2015].
- [17] Emanuel Parzen. *Stochastic processes*, volume 24. SIAM, 1999.
- [18] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.
- [19] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.
- [20] Lloyd S Shapley. Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095, 1953.
- [21] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program GPUs for general-purpose uses. *SIGARCH Comput. Archit. News*, 34(5):325–335, October 2006.
- [22] J. A. E. E. van Nunen. A set of successive approximation methods for discounted markovian decision problems. *Zeitschrift für Operations Research*, 20(5):203–208, 1976.