BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

GPU-ACCELERATED SYNTHESIS OF PROBABILISTIC PROGRAMS

GPU-AKCELEROVNÁ SYNTÉZA PRAVDĚPODOBNOSTNÍCH PROGRAMŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — Bc. VLADIMÍR MARCIN
AUTOR PRÁCE

SUPERVISOR — RNDr. MILAN ČEŠKA, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2021

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

24076

Student:	**Marcin Vladimír, Bc.**
Programme:	Information Technology
Field of study:	Software Verification and Testing
Title:	**GPU-Accelerated Synthesis of Probabilistic Programs**
Category:	Algorithms and Data Structures

Assignment:

1. Study the current methods for automated design and synthesis of probabilistic programs including methods based on MDP abstraction and counter-example guided inductive synthesis.
2. Evaluate these methods on practically relevant case-studies and identify their performance limitations.
3. Design an efficient parallelisation of these methods including a fine-grained parallelisation that is able to utilise modern massively parallel graphical-processing units.
4. Implement the improvements and extensions within an existing probabilistic model-checker (e.g. STORM or PRISM).
5. Carry out a detailed performance evaluation of the implemented methods including an extension of the existing benchmarks.

Recommended literature:

1. Milan Češka, Nils Jansen, Sebastian Junges, and Joost-Pieter Katoen. Shepherding hordes of Markov chains. In Proc. of TACAS'19. Springer, 2019.
2. Milan Češka, Christian Hensel, Sebastian Junges, and Joost-Pieter Katoen. Counterexample-Driven Synthesis for Probabilistic Program Sketches. In Proc. of FM'19. Springer, 2019.

Requirements for the semestral defence:

- Items 1, 2 and partially item 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Češka Milan, RNDr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	November 11, 2020

Abstract

This paper examines the problem of automatic synthesis of probabilistic programs: having a finite family of candidate programs, how can one efficiently identify a program that satisfies a given specification. Even the most straightforward synthesis problems prove to be $\mathcal{NP}$-hard. An improvement to this state of practice is brought by the PAYNT tool, which tackles this problem with a novel integrated technique for synthesising probabilistic programs. Even though it efficiently deals with the exponential growth of the family size, there is still a problem with the underlying state-space explosion. To solve this problem, we have implemented GPU-oriented model-checking algorithms that takes advantage of the GPU architecture and parallelise the task at a state level of a probabilistic model. The overall acceleration that we were able to achieve with this approach was, under certain conditions, close to the theoretically possible limit of the acceleration of the whole synthesis process.

Abstrakt

V tejto práci sa zoberáme problémom automatizovanej syntézy pravdepodobnostných programov: majme konečnú rodinu kandidátnych programov, v ktorej chceme efektívne identifikovať program spĺňajúci danú špecifikáciu. Aj riešenie tých najjednoduchších syntéznych problémov v praxi predstavuje $\mathcal{NP}$-ťažký problém. Pokrok v tejto oblasti prináša nástroj PAYNT, ktorý na riešenie tohto problému používa novú integrovanú metódu syntézy pravdepodobnostných programov. Aj keď sa tento prístup dokáže efektívne vysporiadať s exponenciálnym rastom rodín kandidátnych riešení, stále tu exituje problém spôsobený exponenciálnym rastom jednotlivých členov týchto rodín. S cieľom vysporiadať sa aj s týmto problémom, sme implementovali GPU orientované algoritmy slúžiace na overovanie kandidátnych programov (modelov), ktoré danú úlohu paralelizujú na stavovej úrovni pravdepodobnostých modelov. Celkové zrýchlenie doshiahnuté týmto prístupom za určitých podmienok potom prinieslo takmer teoretický limit možného zrýchlenia syntézneho procesu.

Keywords

Discrete-Time Markov Chains, Markov Decision Processes, Model Checking, Synthesis of Probabilistic Programs, CUDA, Parallelisation

Kľúčové slová

diskrétne Markovove reťazce, Markovove rozhodovacie procesy, overovanie modelov, syntéza pravdepodobnostných programov, CUDA, paralelizácia

Reference

MARCIN, Vladimír. *GPU-Accelerated Synthesis of Probabilistic Programs.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Milan Češka, Ph.D.

Rozšírený abstrakt

Pravdepodobnostné programy sú dôležitým modelovacím jazykom pre popis sytémov obsahujúcich nepredvídateľné alebo nespoľahlivé správanie. Ich aplikácia pokrýva široké spektrum výskumných domén. Pri analýze spoľahlivosti sú nevyhnutné na kvantifikáciu straty správ, zlyhaní systému atď. Ďalšiu oblasť ich aplikácie predstavuje náhodnosť, slúžiaca napríklad na prerušenie symetrie v komunikačných sieťach [11]. Náhodnosť je taktiež cenná aj v bezpečnostných protokoloch napríklad na zabezpečenie anonymity [32] alebo na vytvorenie stratégií riadenia spotreby energie [27]. Za účelom získania garancií o správnosti a efektívnosti pravdepodobnostných programov je možné použiť stochastickú kontrolu modelu [19] založenú na konštrukcii a overení pravdepodobnostných modelov, ako sú Markovove reťazce a Markovove rozhodovanie procesy. Vytvorený model potom overujeme voči danej špecifikácii, ktorá je často popísaná ako formula temporálnej logiky. Automatizované overovanie týchto vlastností podporuje niekoľko existujúcich nástrojov, ako sú Storm [14], Prism [20] alebo Modest [13].

Spomenuté nástroje však zvyčajne vyžadujú na svojom vstupe úplný (pevný) model, čo však v praxi nieje pravidlom. V počiatočných fázach vývoja systému máme k dispozícii iba jeho neúplný popis obsahujúci neznáme voľby. Tie môžu predstavovať nedefinovanú komponentu v sieťovej komunikácii alebo čiastočne implementovaný radič spotreby. Našim cieľom je potom inštanciovať tieto voľby tak, aby výsledný návrh (realizácia) zodpovedala danej špecifikácii – napr. minimalizovanie počtu stratených paketov alebo vybratie optimálnej stratégie na riadenia napájania. Na to aby sme mohli potvrdiť alebo vyvrátiť existenciu požadovanej realizácie, musíme preskúmať skupinu všetkých existujúcich realizácií. Za účelom automatizácie tohto procesu možno použiť tzv. *program sketch* [1, 33] – popis systému obsahujúci neznáme voľby – a necháme automatický syntetizátor vyplniť tento popis, s cieľom získať hľadané riešenie. Na rozhodnutie, či konkrétna realizácia vyhovuje špecifikácii, použije Markov reťazec ako operačný model.

Program sketch $\mathcal{P}$ potom odpovedá rodine Markovových reťazcov $\overline{\mathcal{R}}$, a cieľom automatizovaného syntetizátora je potom prehľadať túto rodinu a nájsť reťazec $\mathcal{M} \in \overline{\mathcal{R}}$ taký, že $\mathcal{M} \models \Phi$, kde Φ je daná špecifikácia. (Figure 1.1).

Automatická syntéza ako taká predstavuje obrovskú výzvu, najmä kvôli problému stavovej explózie, ktorá ovplyvňuje syntézu dvojakým spôsobom: nielenže počet kandidátnych riešení je exponenciálny voči počtu uvažovaných volieb, ale stavový priestor každého jednotlivého reťazca obvykle taktiež rastie exponenciálne voči dĺžke popisu programu. Za posledné roky došlo k významnému zlepšeniu v analýze pravdepodobnostných programov. Češka a kol. [8] aplikovali prístup *zjemňovania abstrakcie* (AR) na rodiny realizácií a doplnili to *protipríkladmi riadenou induktívnou syntézou* (CEGIS). Táto práca stavia na novom výsledku – konkrétne na algoritme predstavenom v [3], ktorý kombinuje dva vyššie spomenuté prístupy a poskytuje výrazné zrýchlenie oproti prístupu založenom na postupnom overovaní realizácií [9, 10], ktorý v tomto prípade predstavuje základnú metódu. Tento prístup však rieši iba jeden z dvoch vyššie spomenutých problémov.

Analýza Markovových modelov je základným kameňom procesu syntézy a predstavuje ortogonálny problém k syntéze pravdepodobnostných programov. Zatiaľ čo syntézny prístup predstavený v [3] sa zaoberá exponenciálnym rastom veľkosti rodiny, analýza každého konkrétneho modelu sa zoberá explóziou stavového priestoru reťazca. V tejto práci sa zameriame na druhý zo spomenutých problémov a na jeho vyriešenie sa snaží využiť potenciál moderného hardvéru v podobe grafických kariet (GPU). Zrýchlením kontroly jednotlivých modelov potom cielime aj na zrýchlenie celkového procesu syntézy pri práci s veľkými modelmi. Naša implementácia rozširuje existujúcu [4], ktorá je založená na nástroji Storm.

Konkrétne sa nám našou paralelnou implementáciou podarilo zrýchliť kontrolu modelov až 16 krát pri kontrole Markovových rozhodovacích procesov a až 716 krát pri kontrole Markovových reťazcov. Toto zrýchlenie následne viedlo až k takmer teoretickému limitu zrýchlenia celkového procesu syntézy. Pri experimentoch sme taktiež zistili, že naša paralelná varianta nieje vždy najlepšia zo širokej škály prístupov implementovaných v nástroji Storm. Našťastie sme navrhli metriku, ktorá je viac menej schopná ešte pred začatím overovania modelu identifikovať, ktorá implementácia by sa mala použiť. Výsledkom je teda adaptívny prístup ku kontrole modelu.

V neposlednom rade sme sa taktiež pokúsili vylepšiť škálovateľnosť aj pri práci s menšími modelmi. Za týmto účelom sme navrhli tzv. *family-based* paralelizáciu, ktorej cieľom je kontrola viacerých modelov súčasne. Tu sme však dospeli k záveru, že táto metóda nieje v aktuálnej verzii syntézneho procesu použiteľná kvôli rozdielnym vlastnostiam súčasne analyzovaných modelov.

GPU-Accelerated Synthesis of Probabilistic Programs

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of RNDr. Milan Češka, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Vladimír Marcin
May 25, 2021

Acknowledgements

I would like to thank my supervisor RNDr. Milan ČEŠKA, Ph.D. for his support both during designing the presented methods and in the critical time of writing this thesis.

Contents

Chapter 1

Introduction

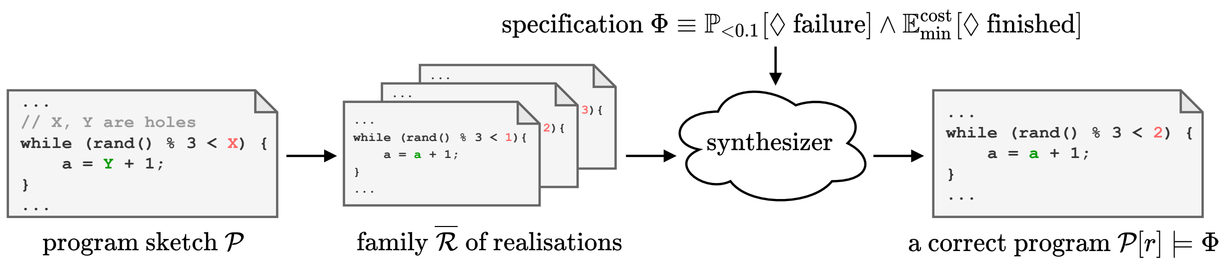
*"The first step is to establish that something is possible then **probability** will occur."*

Elon Musk

Probabilistic programs are an important modelling language to describe systems containing unpredictable or unreliable behaviour. Their application covers a wide range of research domains. In reliability analysis, they are essential to quantify message loss, system failures etc. Another area of stochastic behaviour is randomisation, serving, for example, to prevent flooding or break symmetry in communication networks [11]. Randomisation is also valuable in security protocols, for example, to ensure anonymity [32] or construct dynamic power management plans [27]. To obtain guarantees about the correctness and efficiency of a probabilistic program, one can use *stochastic model-checking* [19] based on the construction and verification of probabilistic models such as *Markov chains* (MC) and *Markov Decision Processes* (MDP). The constructed model is then analysed against the constraints specified by the probabilistic extension of temporal logic. Automated verification of these constraints is supported by several existing tools such as Storm [14], Prism [20] or Modest [13].

However, these tools typically require a fixed model, which is often not the case in practice. In the early stages of system development, we still have an incomplete description containing some *holes*. A hole may represent some undefined component for network communication or a partially implemented controller. The aim is then to complete these holes such that the resulting design (realisation) meets a given specification – e.g. minimise the number of lost messages or select the optimal power management strategy. In order to confirm or refute the existence of the desired realisation, we must examine a *family* of all possible designs. To automate this process, one usually starts with the so-called *sketch* [1, 33] – a system description with holes, representing a family of designs – and let the automatic *synthesizer* (a program that designs programs) fill in this description to obtain a program that satisfies a given specification. To decide whether a realisation satisfies a specification, we use a Markov chain as its operational model.

Program sketch $\mathcal{P}$ then corresponds to a family of Markov chains $\overline{\mathcal{R}}$, and the goal of an automated synthesizer is to explore this family and identify a chain $\mathcal{M} \in \overline{\mathcal{R}}$ such that $\mathcal{M} \models \Phi$, where Φ is the desired specification (see Figure 1.1).


specification $\Phi \equiv \mathbb{P}_{<0.1}[\Diamond\, \text{failure}] \wedge \mathbb{E}^{\text{cost}}_{\text{min}}[\Diamond\, \text{finished}]$
synthesizer
program sketch $\mathcal{P}$
family $\overline{\mathcal{R}}$ of realisations
a correct program $\mathcal{P}[r] \models \Phi$

```
...
// X, Y are holes
while (rand() % 3 < X) {
    a = Y + 1;
}
...
```

```
...
while (rand() % 3 < 1){
    a = a + 1;
}
...
```

```
...
while (rand() % 3 < 2) {
    a = a + 1;
}
...
```

Figure 1.1: The workflow of the synthesis process.

Automated synthesis itself represents a tremendous challenge, particularly due to the state-space explosion problem that affects the synthesis in a twofold manner: not only the amount of possible solutions is exponential wrt. the number of considered holes, but also the state-space of each chain, usually grows exponentially wrt. the length of the program's description. Over the last years, there has been significant improvement in analysing probabilistic program sketches and temporal logic constraints. Češka *et al.* [8] applied *abstraction refinement* (AR) on families of realisations and complemented this with a *counterexample-guided inductive synthesis* (CEGIS) approach [7]. This thesis builds on top of a more recent result – an algorithm presented in [3] that combines the latter two approaches and yield an acceleration of multiple orders of magnitude over the *one-by-one* approach [9, 10], representing the baseline.

The modern trend in hardware development appears to favour the number and the density of transistors instead of higher clock speed due to silicon-based hardware physical limitations. It leads to increasing the number of cores, resulting in many-core and multi-core architectures. Unless a significant breakthrough occurs, this trend will only continue. In the last few years, we have also seen the emergence of a special-purpose many-core *single instruction, multiple threads* (SIMT) hardware – a GPU – as a general-purpose processing device. This situation has also brought innovations in the field of analysis of probabilistic programs. This situation has also brought innovations in the field of analysis of probabilistic programs. Several research groups either implemented GPU-aided model-checking algorithms into existing tools or developed new tools based on many-core algorithms [6, 31]. In light of that, our goal was parallelisation of the synthesis algorithm, while we were only slightly inspired by the approaches presented in the [6] and tried to improve them further.

Key contributions. Analysis of individual Markov chains and Markov decision processes represents a cornerstone of the synthesis process and it is a difficulty orthogonal to the synthesis of probabilistic programs. While the synthesis approach presented in [3] deals with the exponential growth of the family size, analysis of each particular model deals with the explosion of the chain underlying state-space. This work will address this problem by catching the potential of modern GPUs for stochastic model-checking. By speeding up the model-checking, we also aim to speed up the synthesis itself when working with large models. Our implementation extends an existing one [4], which is based on the Storm model checker [14]. The first step was to speed up the model-checking of individual family members, i.e. MC model-checking. Here, in some cases, we achieved an acceleration of up to two orders of magnitude compared to the original approach. The next step was the parallelisation of the MDP model-checking, while the analysed MDP represents the above-mentioned abstraction of the given family. It resulted in many cases up to 16 times faster execution time compared to the method used previously. We have also found that our parallel variant is not always the best of the wide range of model-checking approaches implemented in STORM. Fortunately, we also came up with a metric that can identify these situations. Thus, a result is an adaptive approach to model-checking, where we decide based on the properties of the analysed model which approach will be used.

We have already successfully implemented our methods in STORM and experimentally show the effectiveness on multiple models from various areas such as: performance & reliability, security, planning & synthesis and communication, network and multimedia protocols. All successfully accelerated methods were made available through the Python API of the STORM tool and incorporated in the synthesis algorithm. Subsequent experiments have shown that in cases where we were able to accelerate both versions of model-checking, we accelerated the overall synthesis process up to 4 times, while the value of acceleration was quite close to the theoretical limit of possible acceleration.

Last but not least, we tried to improve scalability even when working with small models. For this purpose, we have proposed *family-based parallelisation*, which aims at model-checking multiple chains at once. However, here we conclude that this method is not applicable in the current version of the synthesis process due to the different properties of the models analysed simultaneously.

Structure of this thesis. The rest of this thesis is structured as follows. Chapter 2 introduces the reader to the necessary theory regarding Markov chains and Markov decision processes. In Chapter 3 we briefly present a probabilistic synthesis problem and give an overview of techniques for its solution. After that, we will look at the original sequential implementation of the core algorithms of the synthesis process (Chapter 4). Further, Chapter 5 introduces all of our parallel methods. Finally, in Chapter 6, the performance of the multiple implementations is compared on relevant models.

Chapter 2

Preliminaries

In this chapter, we provide the necessary theory, based on [12, 7, 8, 16], and introduce notation that will be used throughout the thesis. First we present the simplest probabilistic model – discrete-time Markov chains. We will also, in more detail, introduce the techniques to analyse these models. In the second part of this chapter, we will address a slightly more complex concept – Markov decision processes. It is an extension of Markov chains, which introduces non-deterministic choices between successor states. Both of these operational models will play an essential role in the synthesis of probabilistic programs. Finally, in the last part of this chapter, we will briefly introduce the basic architecture of modern GPUs.

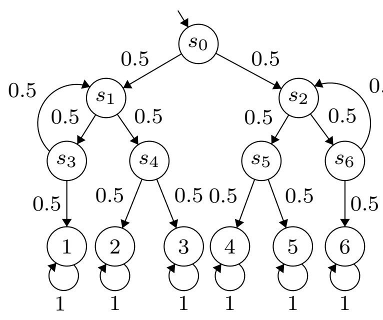
2.1 Discrete – Time Markov Chains

Definition 1 (Probability Distribution)**.** A *(discrete) probability distribution* over a finite set X is a function $\mu : X \to [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$. The set $Distr(X)$ denote the set of all probability distributions over X.

Definition 2 (Descrite-Time Markov Chain)**.** A *(discrete-time) Markov chain* (MC) is a tuple $\mathcal{M} = (S, s_{init}, \mathbf{P})$, where S is a finite set of states, $s_{init} \in S$ is an initial state, $\mathbf{P} : S \to Distr(S)$ is a transition probability matrix.

The transition probability matrix $\mathbf{P}$ specifies for each state s the probability $\mathbf{P}(s, s')$ of transitioning from s to s' in one step (i.e., by a single transition). This probability depends only on the current state s and not, for example, on the path leading to state s from an initial state. Simply put, the state of the system does not depend on history. This is known as the *Markov property* (memorylessness). A Markov chain also induces an *transition probability graph*, where states act as nodes and there is a transition from s to s' iff $\mathbf{P}(s, s') > 0$ (see Figure 2.1). We say that state s is *absorbing iff* $\mathbf{P}(s, s) = 1$.

A *path* π in Markov chain is a possibly infinite path in the underlying graph. It is defined as infinite state sequence $\pi = s_0 s_1 s_2 \dots \in S^\omega$, where $s_0 = s_{init}$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. By applying the Markov property we can quantify the probability of finite path using the transition probability matrix: $\mathbb{P}[s_0 s_1 \dots s_n] = \Pi_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$. However, this „naive“ multiplication yields a probability mass zero if we do it over infinite paths. This situation is addressed by introducing the so-called *cylinder sets*. Formally, the cylinder set $C(\omega)$, for a finite path ω is the set of infinite paths with the common finite prefix ω. The probability of a cylinder set C of $s_0 s_1 s_2 \dots s_n$, is simply $\mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot \dots \cdot \mathbf{P}(s_{n-1}, s_n)$. In this way, we can measure every set of paths that can be expressed as a complement and/or



$$\begin{pmatrix}
0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 1/2 & 0 \\
0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

Figure 2.1: Knuth-Yao algorithm [18] for simulating a six-sided die by repeatedly tossing a fair coin. The coin is flipped until finally the outcome is between one and six.

a countable union of cylinder sets. For example, the probability of $C_1 \cup \overline{C_2}$, with C_1 and C_2 being cylinder sets, is $\mathbb{P}(C_1) + (1 - \mathbb{P}(C_2))$.

Example 1. Consider a Markov chain depicted in Figure 2.1. We want to find out whether the usage of a fair coin to simulate a six-sided die is correct. In our example, we will check if the probability of reaching state 2 is exactly $\frac{1}{6}$ (one can analogously verify the probabilities of other states). In our example, we can write a set of paths that eventually end up in state 2 as: $\bigcup_{i\in\mathbb{N}} s_0(s_1s_3)^i s_1 s_4 2^\omega$ where $(s_1s_3)^i$ denotes, that the cycle between s_1 and s_3 is taken i times. Given this characterization, the desired probability can be expressed as:

$$\sum_{i=0}^{\infty} \mathbf{P}(s_0(s_1s_3)^i s_1 s_4 2) = \frac{1}{8}\sum_{i=0}^{\infty}\left(\frac{1}{4}\right)^i = \frac{1}{8}\cdot\frac{1}{1-\frac{1}{4}} = \frac{1}{6}$$

2.1.1 Model Checking MCs

Stochastic model-checking [19] is a method used to calculate the likelihood that a particular events will occur while a verified system is running. Standard model checkers take as input a description of a model (represented as a state transition system) along with specification (typically a formula in probabilistic temporal logic) and return a probability that the model meets the given property. The cornerstone technique in MC model-checking is the calculation of so-called *reachability probabilities*, as the deciding of more complex properties can, in most cases, be reduced to the calculation of reachability. Therefore, in this work we will primarily focus on properties of this type.

Definition 3 (Unbounded Reachibility)**.** For a set $G \subseteq S$ of target (goal) states, *reachibility property* $\varphi \equiv \mathbb{P}[s \models \Diamond G]$ denote the probability of eventually reaching any of the states in G from $s \in S$. An MC $\mathcal{M}$ satisfies φ iff it is satisfied in the initial state, i.e. $\mathcal{M} \models \varphi \Leftrightarrow s_{init} \models \varphi$. One can also express qualitative property as $\varphi \equiv \mathbb{P}_{\bowtie\lambda}[\Diamond G]$, where $\lambda \in \langle 0, 1\rangle$ and $\bowtie \in \{<, \leq, >, \geq\}$. It express that the probability to reach G relates to λ according to $\bowtie$, and it holds for a given state s iff $\mathbb{P}[s \models \Diamond G] \bowtie \lambda$.

Consider a Markov chain $\mathcal{M}$ with a finite number of states. Model checking this chain against the reachability property $\mathbb{P}_{\bowtie\lambda}[\Diamond G]$ means calculating exact probabilities $x(s) = \mathbb{P}[s \models \Diamond G]$ for each state $s \in S$, and then checking if $\mathbb{P}[s_{init} \models \Diamond G] \bowtie \lambda$. These $x(s)$ values are obtained as a unique solution of a system of linear equations as shown in Algorithm 1.

Algorithm 1: Computing unbounded reachability probabilities for MC.

```
    Input: An MC $\mathcal{M} = (S, s_{init}, \mathbf{P})$, target states $G \subseteq S$
    Output: A probability vector $\boldsymbol{x}(s) = \mathbb{P}[s \models \Diamond G]$ for each $s \in S$
1   $S_0 \leftarrow \{s \in S \,|\, \mathbb{P}[s \models \Diamond G] = 0\}$ // a graph problem
2   $S_1 \leftarrow G$
3   $S_? \leftarrow S \setminus (S_0 \cup S_1)$
4   Solve the following linear equation system:
        $$\boldsymbol{x}(s) = \begin{cases} 0, & \text{if } s \in S_0 \\ 1, & \text{if } s \in S_1 \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot \boldsymbol{x}(s') & \text{if } s \in S_? \end{cases}$$
5   return $\boldsymbol{x}$
```

Example 2. Previously, in Example 1, we derived that $\mathbb{P}[s_0 \models \Diamond\{2\}] = \frac{1}{6}$. As a next step, we will discuss an algorithmic way to obtain $\mathbb{P}[s_0 \models \Diamond 2]$, by applying the Algorithm 1. From Figure 2.1 we can see that $S_1 = \{2\}$ and $S_0 = \{1, 3, 4, 5, 6, s_2, s_5, s_6\}$. Using the characterization from Algorithm 1 we obtain:

$$\begin{aligned}
x(1) = x(3) = x(4) = x(5) = x(6) = x(s_2) = x(s_5) = x(s_6) = 0 \\
x(2) = 1 \\
x(s_0) &= 1/2\; x(s_1) + 1/2\; x(s_2) \\
x(s_1) &= 1/2\; x(s_3) + 1/2\; x(s_4) \\
x(s_3) &= 1/2\; x(s_1) + 1/2\; x(1) \\
x(s_4) &= 1/2\; x(2) + 1/2\; x(3)
\end{aligned}$$

Solution of this system of equations yields: $\boldsymbol{x} = (\boldsymbol{\frac{1}{6}}, \frac{1}{3}, 0, \frac{1}{6}, \frac{1}{2}, 0, 0, 0, 1, 0, 0, 0, 0)^T$. As we can see, by solving a system of linear equations, we obtain the same result as using cylinder sets and geometric series.

In practice, modern model-checkers reduce the derived system of linear equations to a system with $|S_?|$ unknowns instead of $|S|$. It is done in the following way. Let $|S_?|$ be a set of states that have a non-zero probability of reaching G, but at the same time need more than 0 steps to reach G. Let $\boldsymbol{A} = [\mathbf{P}(s, s')]_{s,s' \in S_?}$ be a matrix of transition probabilities between the states in $S_?$. Let the vector $\boldsymbol{b} = [\sum_{g \in G} \mathbf{P}(s, g)]_{s \in S_?}$ denotes the probabilities to reach state $g \in G$ in a single step. Then $\boldsymbol{x} = (x_s)_{s \in S_?}$ (where $x_s = \mathbb{P}[s \models \Diamond G]$) is the unique solution of:

$$\boldsymbol{x} = \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b} \equiv (\boldsymbol{I} - \boldsymbol{A}) \cdot \boldsymbol{x} = \boldsymbol{b}$$

where $\boldsymbol{I}$ is the identity matrix. For our example we have $S_? = \{s_0, s_1, s_3, s_4\}$, and we are interested in the solution of:

$$\begin{pmatrix} 1 & -1/2 & 0 & 0 \\ 0 & 1 & -1/2 & -1/2 \\ 0 & -1/2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_{s_0} \\ x_{s_1} \\ x_{s_3} \\ x_{s_4} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1/2 \end{pmatrix}$$

which yields: $\boldsymbol{x} = (\mathbf{\frac{1}{6}}, \frac{1}{3}, \frac{1}{6}, \frac{1}{2})^T$. Values for other states are 0 or 1. Here we would like to point out the difference in the size of the original matrix from Figure 2.1 and the matrix $\boldsymbol{A}$, which is ≈11 times smaller.

2.1.2 Iterative Methods

The system of linear equations can be solved by any common approach, which involves direct methods, such as Gaussian elimination, or iterative methods, such as Jacobi and Gauss-Seidel. The benefit of direct methods is that they compute exact solutions in a fixed number of steps. However, the main disadvantage is their scalability with increasing model size, and even the reduction mentioned above is not enough for direct methods to apply to large models.

An alternative is offered by iterative methods, which are preferred in practice due to their scalability, but at the expense of accuracy. Therefore, in this thesis we will consider only the iterative methods. The idea is that, starting with an initial estimate for the vector $\boldsymbol{x}$, each iteration produces an increasingly accurate approximation to the solution of the linear equation system. Let $\boldsymbol{x}^{(k)}$ denotes the approximation computed in the k-th iteration. Each estimate $\boldsymbol{x}^{(k)}$ than uses values of $\boldsymbol{x}^{(k-1)}$ and iterative process is terminated when the solution vector is judged to have converged sufficiently.

The specific methods then differ from each other in the way they calculate the vector $\boldsymbol{x}^{(k)}$. Below we will introduce the two most well-known iterative methods.

Jacobi Iteration Method. The Jacobi method is based on the fact that the i-th equation of the linear equation system $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$:

$$\sum_{j=0}^{|S_?|-1} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}(j) = \boldsymbol{b}(i) \quad \text{for } i \in \{0, \ldots, |S_?| - 1\}$$

can be rearranged as:

$$\boldsymbol{x}(i) = \left(\boldsymbol{b}(i) - \sum_{j \neq i} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}(j) \right) / \boldsymbol{A}(i,i)$$

which yields this update scheme:

$$\boldsymbol{x}^{(k)}(i) = \left(\boldsymbol{b}(i) - \sum_{j \neq i} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}^{(k-1)}(j) \right) / \boldsymbol{A}(i,i)$$

Note that for for probabilistic model-checking, the diagonal elements $\boldsymbol{A}(i,i)$ will always be non-zero.

Gauss-Seidel Method. The Jacobi method can be improved by using the most up-to-date values of $\boldsymbol{x}(j)$ that are available. This gives rise to the Gauss-Seidel method:

$$\boldsymbol{x}^{(k)}(i) = \left(\boldsymbol{b}(i) - \sum_{j<i} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}^{(k)}(j) - \sum_{j>i} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}^{(k-1)}(j) \right) / \boldsymbol{A}(i,i)$$

2.2 Markov Decision Processes

As mentioned above, each state of a Markov chain had a unique probability distribution over its successors. However, what if we want to model, for example, the abstraction of a modular system (e.g., with interchangeable implementations), or we simply do not have statistical information of probabilities. We can no longer model such requirements using the Markov chain, and therefore we present *Markov decision processes*, a concept first introduced by Bellman [5] in the 1950s.

Definition 4 (Markov Decision process)**.** A Markov decision process (MDP) is a tuple $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$, where S and s_{init} indicate the same as in MC, Act is a finite set of actions, and $\mathbf{P} : S \nrightarrow Distr(S)$ is a partial transition probability function. Let $Act(s)$ denotes the set of all available actions in state $s \in S$. In general we require $Act(s) \neq \emptyset$ for each $s \in S$.

Based on the above, we can now easily model properties containing non-determinism. Suppose the MDP is in state s, then the successor selection is made in two steps. First, we non-deterministically choose one of the actions in $Act(s)$, say α. In the second step, the successor is chosen using the probability distribution given by $\mathbf{P}(s)(\alpha)$. It should be obvious that if for all states $s \in S$, holds $|Act(s)| = 1$, this specific MDP is equivalent to an MC.

A path π in an MDP is defined as possibly infinite sequence of pairs of states and actions $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \cdots \in (S \times Act)^\omega$, where $\forall i \in \mathbb{N}_0 : \mathbf{P}(s_i, \alpha_i, s_{i+1}) > 0$ and $\alpha_i \in Act(s_i)$. Let $\text{Paths}^{\mathcal{M}}(s)$ and $\text{Paths}^{\mathcal{M}}_{fin}(s)$ denote the sets of all infinite and finite paths taken from state s in $\mathcal{M}$, respectively. Then we use $\text{Paths}^{\mathcal{M}}$, $\text{Paths}^{\mathcal{M}}_{fin}$ to denote the sets of all such paths in the MDP $\mathcal{M}$. For finite $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{n-1} s_n$, let $last(\pi) = s_n$ denote the last state of π. The probability of a finite path is evaluated in the same way as for MCs, using a transition probability matrix: $\mathbb{P}[\pi] = \Pi_{i=0}^{n-1}\mathbf{P}(s_i, \alpha_i, s_{i+1})$. However, in order to define the probability space over infinite paths (as in MCs), we must first completely resolve all the present non-determinism. To do so, one might use a *scheduler* responsible for deterministically choosing an action in each state of an MDP.

Definition 5 (Scheduler)**.** A *scheduler* for an MDP $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$ is a function $\sigma : \text{Paths}^{\mathcal{M}}_{fin} \to Act$ such that for all finite path fragments $\pi \in \text{Paths}^{\mathcal{M}}_{fin}$ it holds that $\sigma(\pi) \in Act(last(\pi))$. The set of all schedulers of $\mathcal{M}$ is denoted as $\Sigma^{\mathcal{M}}$.

Now a scheduler can be applied to an MDP such that if an MDP enters a state $last(\pi)$ via a path π, a scheduler deterministically chooses an action $\sigma(\pi)$. So, by using a scheduler, we get an MC called *induced Markov chain.*

Definition 6 (Induced Markov Chain)**.** For an MDP $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$ and scheduler $\sigma \in \Sigma^{\mathcal{M}}$, the *induced MC* is $\mathcal{M}^\sigma = (\text{Paths}^{\mathcal{M}}_{fin}, s_{init}, \mathbf{P}^\sigma)$ where for any π, $\pi' \in \text{Paths}^{\mathcal{M}}_{fin}$:

$$\mathbf{P}^\sigma(\pi, \pi') = \begin{cases} \mathbf{P}(last(\pi), \sigma(\pi))(s') & \text{if } \pi' = \pi \xrightarrow{\sigma(\pi)} s' \\ 0 & \text{otherwise} \end{cases}$$

Note that induced MC has possibly an infinitely large state-space depending on the used scheduler. However, in the case of *memoryless schedulers*, we can construct a finite-state MC.

Definition 7 (Memoryless scheduler)**.** A scheduler σ is *memoryless* if $\sigma(\pi)$ depends only on $last(\pi)$, i.e., for any $\pi, \pi' \in \text{Paths}^{\mathcal{M}}_{fin}$, $last(\pi) = last(\pi') \Rightarrow \sigma(\pi) = \sigma(\pi')$.

The input of the memoryless scheduler is thus a path of length 1, representing the current state $s \in S$ in a given MDP. Furthermore, if we reach state s, the scheduler will always return the same action $\alpha = \sigma(s)$. Thus, the MC obtained using a memoryless scheduler is fully consistent with Definition 2.

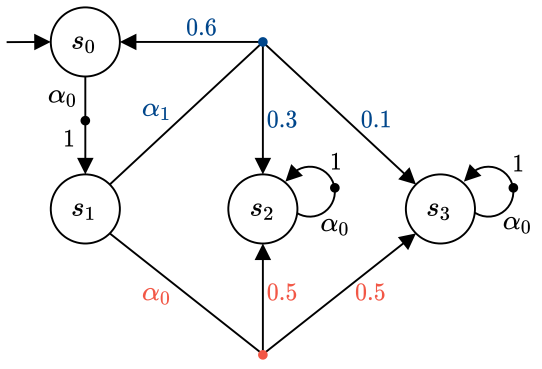
Example 3. Consider the MDP $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$ depicted in Figure 2.2. Here $S = \{s_0, s_1, s_2, s_3\}$, $s_{init} = s_0$, $Act = \{\alpha_0, \alpha_1\}$, and the transition probability matrix $\mathbf{P}$ is shown in Figure 2.2. We can see that the only state with non-deterministic choice is state s_1. In this state there are two possible probability distributions:

$$\mathbf{P}(s_1, \alpha_0) = [s_2 \mapsto 0.5, s_3 \mapsto 0.5]$$
$$\mathbf{P}(s_1, \alpha_1) = [s_2 \mapsto 0.6, s_2 \mapsto 0.3, s_3 \mapsto 0.1]$$

Example of the finite path is $\pi = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_0} s_2$ and example of the infinite path is $\pi' = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_0} s_2 \xrightarrow{\alpha_0} \dots$. The probability of finite path π can be calculated as $\mathbb{P}[\pi] = \mathbf{P}(s_0, \alpha_0, s_1) \cdot \mathbf{P}(s_1, \alpha_0, s_2) = 1 \cdot 0.5 = 0.5$. In our example, there are only two memoryless schedulers:

$$\sigma_0 = [s_1 \mapsto \alpha_0]$$
$$\sigma_1 = [s_1 \mapsto \alpha_1]$$

Applying scheduler σ_0 induces an MC, which omits the blue line from the transition probability matrix and applying the σ_1 scheduler on the contrary omits the red line.



$$\mathbf{P} = \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.3 & 0.1 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right)$$

Figure 2.2: An example of the MDP and its transition probability matrix $\mathbf{P}$ with row grouping.

2.2.1 Model Checking MDPs

As with MC model-checking, we will focus on the reachability properties. In MCs rechability probability could be obtained as the unique solution of a system of linear equations. In the case of MDP, the non-deterministic choices between probability distributions nullify the sense of the individual probability of reaching a particular state since there is no longer only one probability. Thus, in the case of MDPs, the verified properties will have slightly different semantics. We say that property φ holds for an MDP $\mathcal{M}$ iff it holds for the induced

MCs of all schedulers, i.e. $\mathcal{M} \models \varphi \Leftrightarrow \forall\sigma \in \Sigma^{\mathcal{M}} : \mathcal{M}^{\sigma} \models \varphi$. To avoid iterating through an infinite number of schedulers, instead of asking if an MDP meets the given specification, we will ask:

Does the property hold even in a worst/best case of non-deterministic choices?

This question, therefore, raises the idea of finding the maximum or minimum probability. For example, one could be interested in „the maximum probability of error occurrence“ or „the minimum probability of a message being delivered“. In order to find these probabilities, it is sufficient to look only at a special kind of schedulers, as illustrated in Proposition 1.

Proposition 1 (Maximizing/minimizing Scheduler)**.** Let $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$ be an MDP and φ be a property. Then $\sigma_{min}, \sigma_{max} \in \Sigma^{\mathcal{M}}$ denote the memoryless schedulers such that $\forall\sigma \in \Sigma^{\mathcal{M}} : \mathbb{P}[\mathcal{M}^{\sigma_{min}} \models \varphi] \leq \mathbb{P}[\mathcal{M}^{\sigma} \models \varphi] \leq \mathbb{P}[\mathcal{M}^{\sigma_{max}} \models \varphi]$.

So now, in order to decide reachability property $\varphi \equiv \mathbb{P}_{\leq\lambda}[\Diamond G]$ (safety property $\leq$), we compute a maximising scheduler σ_{max}. Using this scheduler we can assign to each state $s \in S$ its maximal probability for reaching a G: $\boldsymbol{x}(s) := \max_{\sigma\in\Sigma^{\mathcal{M}}} \mathbb{P}[\mathcal{M}^{\sigma}, s \models \Diamond G]$, and then asserting that $\boldsymbol{x}_{max}(s_{init}) \leq \lambda$. Similarly, deciding $\varphi \equiv \mathbb{P}_{\geq\lambda}[\Diamond G]$ (liveness property $\geq$), involves finding, for each $s \in S$, minimum probabilities $\boldsymbol{x}_{min}$ of reaching G and then checking whether $\boldsymbol{x}_{min}(s_{init}) \geq \lambda$. This upper/lower bound $\boldsymbol{x}_{max}/\boldsymbol{x}_{min}$ on the reachability probability can be obtained as a solution to a mixed-integer linear program (MILP). The benefit of this approach is that it computes exact answers. However, its drawback is poor scalability while working with large models. An alternative method is *Value iteration* algorithm, offering better scalability, but at the expense of accuracy.

2.2.2 Value Iteration.

The idea behind the *Value Iteration* (VI) method, is that instead of computing an exact solution, it computes a probability of reaching G within n steps. In practice, if n is large enough, the result is sufficiently accurate. Formally, we introduce $\boldsymbol{x}^n(s)$ for $s \in S$, $n \in \mathbb{N}$ and equations:

$$\boldsymbol{x}^n(s) = \begin{cases} 0, & \text{if } (s \in S_0) \vee (s \notin S_1 \wedge n = 0) \\ 1, & \text{if } s \in S_1 \\ \max_{\alpha\in Act(s)} \left(\sum_{s'\in S} \mathbf{P}(s,\alpha,s') \cdot x_{s'}^{n-1}\right) & \text{otherwise.} \end{cases}$$

It can be shown [29], that $\lim_{n\to\infty} \boldsymbol{x}_s^n = \max_{\sigma\in\Sigma^{\mathcal{M}}} \mathbb{P}[\mathcal{M}^{\sigma}, s \models \Diamond G]$. Thus, for sufficiently large n, we can approximate the actual probabilities by the values of $\boldsymbol{x}^n(s)$. Furthermore, the minimum probabilities can be computed in near identical fashion, by replacing „max“ with „min“ in the above.

Typically, a value of n is not set before the start of the computation, but rather determined on-the-fly according to the convergence of the values $\boldsymbol{x}^n(s)$. A simple but effective way is to terminate the calculation if $\max_{s\in S}(\boldsymbol{x}^n(s) - \boldsymbol{x}^{n-1}(s)) < \varepsilon$, where ε is a user specified threshold. In cases where the $\boldsymbol{x}^n(s)$ values are quite small, the more reliable criterion may be $\max_{s\in S}\left((\boldsymbol{x}^n(s) - \boldsymbol{x}^{n-1}(s))/\boldsymbol{x}^{n-1}(s)\right) < \varepsilon$, i.e. the maximum *relative* difference. It is important to note, that these criteria do <u>not</u> guarantee that the resulting values are within ε of the true results. An illustration of how to implement the VI algorithm is given in Algorithm 2. Note that there is no need to store all the $\boldsymbol{x}^n$ vectors (just the last two will suffice).

Algorithm 2: Computing the maximal reachability probability for an MDP, using the VI algorithm.

```
   Input: An MDP $\mathcal{M} = (S, s_{init}, Act, \mathbf{P})$, target states $G \subseteq S$
   Output: A probability vector $\boldsymbol{x}(s) = \mathbb{P}[s \models \Diamond G]$ for each $s \in S$
 1 $S_0 \leftarrow \{s \in S \mid \forall\sigma \in \Sigma^{\mathcal{M}} : \mathbb{P}[\mathcal{M}^\sigma, s \models \Diamond G] = 0\}$ // a graph problem
 2 $S_1 \leftarrow \{s \in S \mid \exists\sigma \in \Sigma^{\mathcal{M}} : \mathbb{P}[\mathcal{M}^\sigma, s \models \Diamond G] = 1\}$ // a graph problem
 3 $S_? \leftarrow S \setminus (S_0 \cup S_1)$
 4 $x(s)_{s\in S} \leftarrow (s \in S_1)\ ?\ 1\ :\ 0$
 5 do
 6     foreach $s \in \mathcal{S}_?$ do
 7         $x'_s \leftarrow \max_{\alpha\in Act(s)} \left(\sum_{s'\in S} \mathbf{P}(s,\alpha,s') \cdot x_{s'}\right)$
 8     end
 9     $swap(x_s, x'_s)$
10 while $\max\{|x_s - x'_s| \mid s \in S_?\} \leq \varepsilon$
11 return $\boldsymbol{x}$
```

Example 4 ([12])**.** To illustrate the VI method, assume an MDP $\mathcal{M}$ from Example 3, and safety property $\varphi \equiv \mathbb{P}_{\leq 0.6}[\Diamond s_2]$. In order to decide whether $\mathcal{M} \models \varphi$, we must compute, for each $s \in S$, an upper bound $\boldsymbol{x}_{max}$ on the reachability probability. Using Algorithm 2, we can see that $S_0 = \{s_3\}$ and $S_1 = \{s_2\}$, yielding the following equations for VI:

$$\begin{aligned}
x^n(s_2) &= 1 && \text{for } n \geq 0\\
x^n(s_3) &= 0 && \text{for } n \geq 0\\
x^0(s_i) &= 0 && \text{for } i \in \{0,1\}\\
x^n(s_0) &= x^{n-1}_{s_1} && \text{for } n > 0\\
x^n(s_1) &= \max\{0.5 \cdot x^{n-1}_{s_2} + 0.5 \cdot x^{n-1}_{s_3}, 0.6 \cdot x^{n-1}_{s_0} + 0.1 \cdot x^{n-1}_{s_3} + 0.3 \cdot x^{n-1}_{s_2}\} && \text{for } n > 0
\end{aligned}$$

Following this equations, the initial vector $x^0 = (0, 0, 1, 0)^T$. In order to get a new vector x^1, vector x^0 is multiplied by the transition matrix $\mathbf{P}$.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0.5 & 0.5 \\ 0.6 & 0 & 0.3 & 0.1 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ \hline 0.5 \\ 0.3 \\ \hline 1 \\ 0 \end{pmatrix}$$

The resulting vector now contains probabilities for all available states and actions and follows the same row grouping as the used matrix. We now look for maximum over the groups (actions) as we are interested in maximal probabilities. Since we have only one state with more than one action, the group reduction will be only one operation $x^1(s1) = \max\{0.5, 0.3\} = 0.5$. Below, we show the vector $\boldsymbol{x}^n = (x^n_{s_0}, x^n_{s_1}, x^n_{s_2}, x^n_{s_3})^T$, for an increasing n, terminating with $\varepsilon = 0.001$.

$$
\begin{aligned}
\boldsymbol{x}^0 &= (0,0,1,0)^T & \boldsymbol{x}^7 &= (0.66,0.696,1,0)^T\\
\boldsymbol{x}^1 &= (0,0.5,1,0)^T & \boldsymbol{x}^8 &= (0.696,0.696,1,0)^T\\
\boldsymbol{x}^2 &= (0.5,0.5,1,0)^T & \boldsymbol{x}^9 &= (0.696,0.7176,1,0)^T\\
\boldsymbol{x}^3 &= (0.5,0.6,1,0)^T & \boldsymbol{x}^{10} &= (0.7176,0.7176,1,0)^T\\
\boldsymbol{x}^4 &= (0.6,0.6,1,0)^T & &\dots\\
\boldsymbol{x}^5 &= (0.6,0.66,1,0)^T & \boldsymbol{x}^{22} &= (0.74849,0.74849,1,0)^T\\
\boldsymbol{x}^6 &= (0.66,0.66,1,0)^T & \boldsymbol{x}^{23} &= (0.74849,0.74909,1,0)^T
\end{aligned}
$$

The exact values are $\boldsymbol{x} = (0.75, 0.75, 1, 0)^T$, which differ from $\boldsymbol{x}^{23}$ by up to 0.00151. However, as we obtained upper bound probabilities, we can now conclude that $\mathcal{M} \not\models \varphi$ because $\boldsymbol{x}_{max}(s_0) = 0.74849 > 0.6$.

We would like to point out that, similarly to MC model-checking, we can reduce the $\mathbf{P}$ matrix, and consider only $S_?$ states inducing the submatrix $\boldsymbol{A} = [P(s, \alpha, s')]_{s,s' \in S_?, \alpha \in Act(s)}$. To compensate for the loss of S_1 states, we again use a vector $\boldsymbol{b}_{s \in S_?, \alpha \in Act(s)} := \sum_{g \in G} \mathbf{P}(s, \alpha, g)$, denoting the probabilities to reach state $g \in G$ in a single step from $s \in S_?$. Then the iterative equation will look like $\boldsymbol{x}' = \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{x}$ is also reduced to contain only the states in $S_?$. Thus, the first iteration for the MDP $\mathcal{M}$ and property φ from Example 4 will look like:

$$
\boldsymbol{x}^1 = reduce\left[\begin{pmatrix} 0 & 1 \\ \hline 0 & 0 \\ 0.6 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \hline 0.5 \\ 0.3 \end{pmatrix}\right] = \begin{pmatrix} 0 \\ 0.5 \end{pmatrix}
$$

2.3 General-Purpose Computing on GPUs

While modern mainstream CPUs are designed to handle a wide range of tasks quickly, they are limited in the concurrency of tasks running. GPUs use an entirely different approach. A GPU was designed as an accelerator for a particular task set, namely quickly render high-resolution images and video. To this end, GPUs use their massively parallel architecture where thousands of threads all perform the same actions over their respective data. Since GPUs can perform parallel operations on multiple data sets, they are commonly used for non-graphical tasks such as scientific computation or machine learning.

2.3.1 Programming and Memory Model.

Each GPU program consists of a *host* part that runs on CPU and a *device* part composed of so-called *kernels*. The kernels are parallel programs that are executed as sets of *threads* on GPU. The threads are organized into one, two or three dimension groups called thread *blocks*. Each kernel then represents a *grid* of one or more thread blocks (while it can also have up to three dimensions). The threads within a block can synchronize among themselves through light-weight synchronization barriers. Each thread within a grid has a unique identifier, which can be derived from its block id (position of its block within a grid), the block dimension and the thread id (its position within its block). Using thread id a running thread can decide which work it should do.

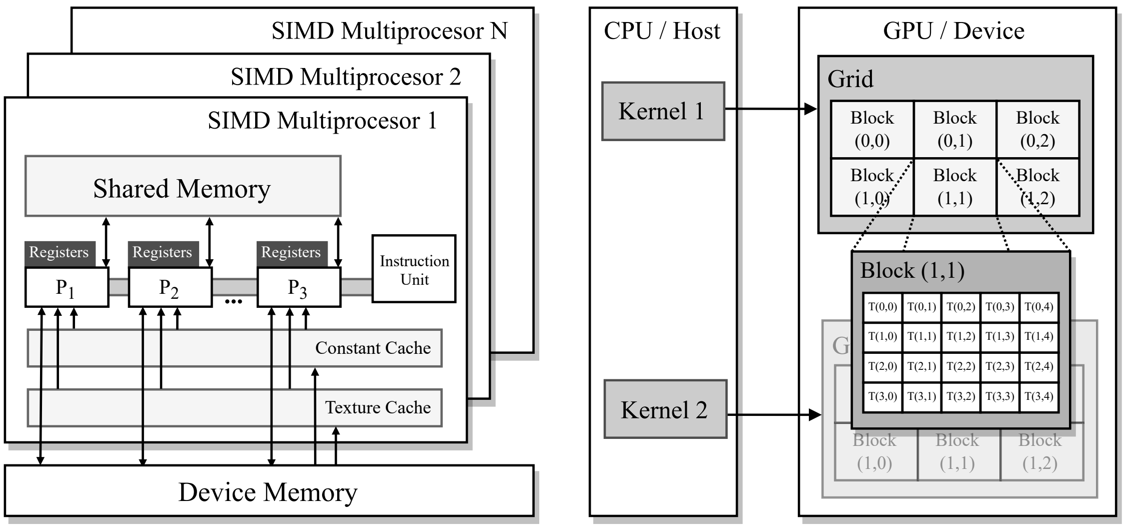


Figure 2.3: The CUDA Platform and Memory Model. Redrawn from the source: [17].

There are various kinds of memories in the CUDA memory model that significantly differ in address space, access latency, scope, and lifetime. The memory hierarchy loosely copies the program grid-block-thread hierarchy and is organized as follows:

(1) off-chip *device/global memory*: is the largest and has a very high latency, which is usually the main performance bottleneck. It is also the only memory that the host program can access.

(2) off-chip *local memory*: refers to „memory“ (bytes are stored in global memory) where registers and other thread-data is spilled.

(3) on-chip *shared memory*: is very fast and usually used for inter-thread communication within one block. If multiple blocks are executed in parallel, this memory is equally split between them (warning: Only a limited amount available, so it should not be overused.).

(4) off-chip *constant & texture memories*: are read-only regions in the global memory space with on-chip read-only caches.

(5) on-chip *registers*: are the fastest but also the smallest memory of the whole hierarchy. Each thread is allocated a set of registers when a kernel is started and generally stores frequently used variables which are private to each thread.

2.3.2 GPU Architecture and Execution Model.

A CUDA-compliant (*Compute Unified Device Architecture*) GPGPU device is composed of several *streaming multiprocessors* (SMs). Each SM has its own set of processor cores (CUDA cores) called *streaming processors* (SPs).

As we said before, single kernel execution is split into multiple blocks of threads. Each block is then assigned to one SM, whereas each SM can execute several blocks independently. The number of blocks that can be physically executed in parallel on the same SM is limited by the amount of shared memory and the number of registers. The GPUs computing architecture employs a *Single Instruction Multiple Threads* (SIMT) model of execution, which means that each thread is executed independently with its own instruction and local state. Moreover, threads of a single block are split into *warps*; each typically consists of 32 threads. The execution of several warps may be interleaved to cover waiting times for memory access to slow global memory. Threads in a warp execute the same instructions (even memory loads) in a lockstep fashion and therefore it is desirable that the memory accesses for these threads are physically coalesced and that their code paths do not diverge. These two mentioned conditions are also the most common mistakes when creating GPU programs. For example, suppose there is a condition in the kernel code that contains the thread id of one thread. In that case, its next instruction will be different from the next instruction of the other threads within its warp, and the whole warp holds its execution while this one thread performs its divergent instructions. The same philosophy applies to shared/global memory access. We said that the accesses of the threads within the warp should be physically coalesced, which means that the adjacent threads read from the adjacent locations within one so-called cache line. If it wasn't the case, and one thread would like to read with a non coalesced pattern, the other threads within its warp would have to wait, and so the overall performance is at most $1/32$ of the maximal one (see Figure 2.4). Threads/warps are not assigned to SPs arbitrarily, but IDs of threads executed in the same warp are sequential. Therefore a programmer can plan and use algorithms and data structures that take this into account.

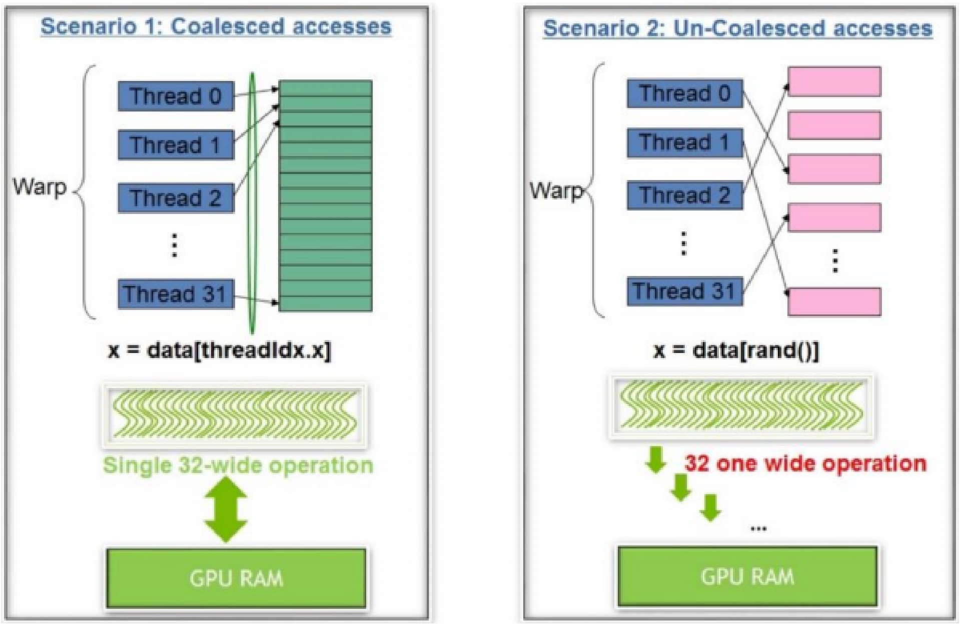


Figure 2.4: Correct vs Incorrect memory access by single warp threads [22].

Chapter 3

Synthesis of Probabilistic Programs

In the previous chapter, we introduced Markov chains and showed how to verify whether a given MC meets a required property. Now, we will address an opposite problem. Let us have a set of Markov chains; how can we find the one that meets a given specifications. This is the situation addressed by *synthesis methods*, and in this chapter, we will introduce the *hybrid dual-guided synthesis* approach, which was firstly presented in [2]. Name hybrid comes from using two oracles, which are based on the earlier presented synthesis methods, namely AR [8] and CEGIS [7]. It focuses on sets of Markov Chains, having different topologies of the state-space and in order to describe them, it uses a concept of so-called *families* [8].

3.1 Families of Markov Chains

Definition 8 (Family of MCs)**.** A *family* of MCs is a tuple $\mathcal{D} = (S, s_{init}, K, \mathcal{B})$ with S and s_{init} as before, K is a finite set of parameters with domains $V_k \in S$ for each $k \in K$, and $\mathcal{B} : S \to Distr(K)$ denotes a family of transition probability functions.

Function $\mathcal{B}$ maps each state to a distribution over parameters. It is these parameters that represent the unknown options (*holes*) of a system under design. Since parameter domains are subsets of S, function $\mathcal{B}$ maps the state to a distribution over states after substituting specific values into holes. Thus, it yields a concrete Markov chain representing one *realisation* (member) of a family. The following definition describes this more formally.

Definition 9 (Realisation)**.** A *realisation* of a family $\mathcal{D} = (S, s_{init}, K, \mathcal{B})$ of Markov chains is a function $r : K \to S$ such that $\forall k \in K : r(k) \in V_k$. We say that r induces MC $\mathcal{D}_r = (S, s_{init}, \mathcal{B}_r)$ iff $\mathcal{B}_r(s, s') = \sum_{k \in K, r(k)=s'} \mathcal{B}(s)(k)$ for all pairs of states $s, s' \in S$. Then, let $\mathcal{R}^{\mathcal{D}}$ denotes the set of all realisations.

Here we would like to point out that the size of the set of all realisations $\mathcal{R}^{\mathcal{D}} = \Pi_{k \in K} V_k$ is exponential in $|K|$, which rises the first state-space explosion problem.

Example 5. To better illustrate the concept of families, assume a family $\mathcal{D} = (S, s_0, K, \mathcal{B})$ of Markov chains with a set of states $S = \{s_0, s_1, s_2, s_3\}$, a set of parameters $K = \{X, Y, T, F\}$ with domains $T_X = \{s_1, s_2\}$, $T_Y = \{t, f\}$, $T_T = \{t\}$, $T_F = \{f\}$ (since T and F can each take only one value, actually they are not parameters), and a family of transition probability functions $\mathcal{B}$, which is shown in Figure 3.1 (together with all the family members). Notice the different topology of the underlying state-space of each of realisations resulting in a different sets of reachable states.

$$
\begin{aligned}
\mathcal{B}(s_0) &= [X \mapsto 1],\\
\mathcal{B}(s_1) &= [T \mapsto 0.6, Y \mapsto 0.2, F \mapsto 0.2],\\
\mathcal{B}(s_2) &= [T \mapsto 0.2, Y \mapsto 0.2, F \mapsto 0.6],\\
\mathcal{B}(t) &= [T \mapsto 1],\\
\mathcal{B}(f) &= [F \mapsto 1]
\end{aligned}
$$

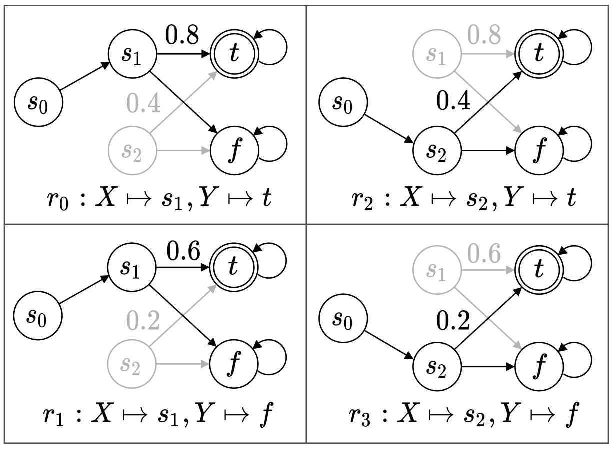

s_0
s_1
s_2
t
f
0.8
0.4
$r_0 : X \mapsto s_1, Y \mapsto t$
0.8
0.4
$r_2 : X \mapsto s_2, Y \mapsto t$
0.6
0.2
$r_1 : X \mapsto s_1, Y \mapsto f$
0.6
0.2
$r_3 : X \mapsto s_2, Y \mapsto f$

Figure 3.1: A family $\mathcal{D}$ of 4 Markov chains. For simplicity, some probabilities are omitted, but note that all probabilities must be added to one. The gray out states means that they are unreachable in the specific realisation (adapted from [3]).

We define two types of synthesis problems currently supported by the presented method. In case, when the state-space S and the set of parameters K are finite, the both of these problems are decidable and, to be more specific, $\mathcal{NP}$-hard [35].

Problem 1 (Feasibility synthesis)**.** Given a family of Markov chains $\mathcal{D}$ over parameters K and a specification $\varphi \equiv \mathbb{P}_{\bowtie\lambda}[\lozenge G]$, find a realisation $r : K \mapsto V$ such that $\mathcal{D}_r \models \varphi$.

Problem 2 (Maximum Synthesis)**.** Given a family of Markov chains $\mathcal{D}$ over parameters K and a maximising property $\varphi_{max} \equiv \mathbb{P}_{\bowtie\lambda}[\lozenge G]$, find a realisation $r : K \mapsto V$ such that:

$$r^* \in \ \arg\max_{r \in \mathcal{R}^{\mathcal{D}}} \mathbb{P}[\mathcal{D}_r \models \lozenge G]$$

The *minimal synthesis* problem is defined analogously. In the following, we will focus mainly on feasibility synthesis, as the basic idea for maximum synthesis remains the same, with the difference that if we obtain a feasible solution, we update the threshold λ of maximising property to exclude this solution. After exhaustion of the state-space of realisations, the actual solution is declared optimal (maximal). Variant of the minimal synthesis is defined analogously.

One possible solution to the synthesis problems is the so-called *one-by-one* approach, where one have to enumerate through each realisation $r \in \mathcal{R}^{\mathcal{D}}$. However, the parameter-space and state-space explosions render this approach unusable for large families, which brings the need for advanced techniques that accelerate the pruning of the state-space of candidate solutions.

3.2 Counterexample-Guided Inductive Synthesis

This section will introduce the *counterexample-guided inductive synthesis* (CEGIS) [7] synthesis method, which forms the basic building block of the presented hybrid method. Consider Figure 3.2. First, we assume a set of realisations $\overline{\mathcal{R}}$. We pick a realisation $r \in \overline{\mathcal{R}}$

(using a SAT solver), and check whether the particular r is a solution. If it is a solution, the synthesis is complete, and the realisation r is returned to the user. Otherwise, we compute a so-called *counterexample*; i.e. a fraction of the chain $\mathcal{D}_r$ that contains enough of paths that violate Φ (for more detailed description see [7]). Based on this counterexample, we then create a set $\mathcal{R}'$ which contains r and potentially more realisations that all violate the given property, and thus we can prune $\mathcal{R}'$ from $\overline{\mathcal{R}}$. This process repeats until we find a feasible realisation or until the set of realisations is empty.

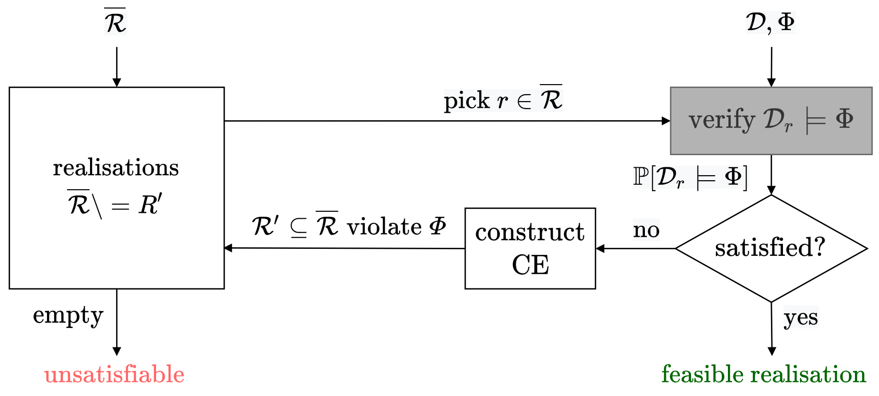

$\overline{\mathcal{R}}$
$\mathcal{D}, \Phi$
pick $r \in \overline{\mathcal{R}}$
verify $\mathcal{D}_r \models \Phi$
realisations
$\overline{\mathcal{R}} \setminus = R'$
$\mathbb{P}[\mathcal{D}_r \models \Phi]$
$\mathcal{R}' \subseteq \overline{\mathcal{R}}$ violate Φ
construct CE
no
satisfied?
empty
yes
unsatisfiable
feasible realisation

Figure 3.2: Counterexample-Guided inductive synthesis.

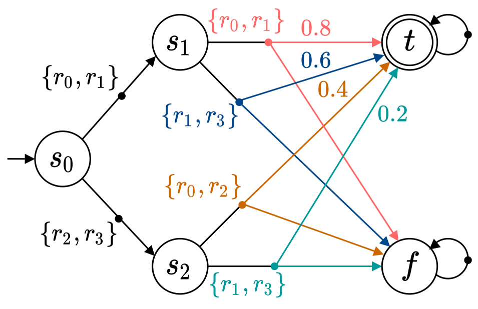
Example 6 (CEGIS illustration)**.** Consider running CEGIS, on a family from Example 5, and safety property $\varphi \equiv \mathbb{P}_{\leq 0.3}[\lozenge\{t\}]$. Let r_0 be the first selected realisation. Via MC model-checking, we obtain $\mathbb{P}[\mathcal{D}_r, s_0 \models \lozenge\{t\}] = 0.8 > 0.3$, therefore, we construct a counterexample that contains the (only) path to $\{t\}$: $s_0 \to s_1 \to t$ having probability $0.8 > 0.3$. It induces CE $C = \{s_0, s_1, t\}$. The next step is to create a set $\mathcal{R}'$ of realisations that also violate the property φ. In order to do so, we compute a so-called *conflict*. Shortly, a conflict is represented by a set of relevant parameters concerning a counterexample (for more details we again refer to [7]). Conflict for C is $K = \{X, Y\}$, i.e. the set of all parameters used in the definition of $\mathcal{B}(s)$ for $s \in C$. This implies that none of the parameters can be generalised, and thus, we reject only a single realisation r_0. Other realisations also do not allow any generalisation, so the process will be iterating through realisations until accidentally guessing r_3 (since it is only feasible solution).

3.3 Abstraction Refinement

Abstraction refinement (AR) [8] is the second synthesis method used by the hybrid method. The approach of this method is opposite to that of the CEGIS. Instead of analysing a single realisation, and then inferring statements about others, an AR argues about sets of realisations at once. In order to do so, it abstracts the set of realisations by constructing a stochastic process in which all realisations are possible at once. Formally, this process corresponds to an MDP, which in each state $s \in S$ has the possibility of a non-deterministic choice between specific realisations in $\mathcal{R}^{\mathcal{D}}$. Specifically, we will refer to this MDP as *quotient MDP*.

Definition 10 (Quotient MDP [8])**.** Assume a family of MCs $\mathcal{D} = (S, s_{init}, K, \mathcal{B})$. A *quotient MDP* of $\mathcal{D}$ is an MDP $\mathcal{M}^{\mathcal{D}} = (S, s_{init}, \mathcal{R}^{\mathcal{D}}, \mathbf{P})$, where $\mathbf{P}(\cdot)(r) \equiv \mathcal{B}_r$.

Thus, a quotient MDP is an abstraction of a given family, and over-approximates the behaviour of each family member. It means that it can simulate the behaviour of each realisation and can switch realisations directly at run-time. For example, let's take a look at Figure 3.3, which shows the quotient MDP of the family from Example 5. We can easily see that the resulting MDP also contains paths not included in any of the original family members.



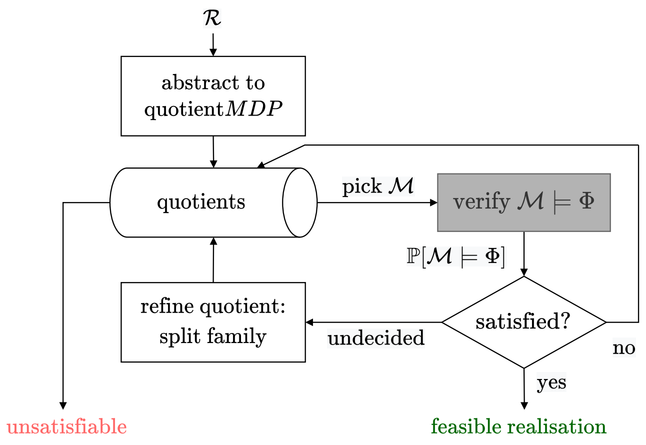
$$\begin{array}{c|ccccc}
 & s_0 & s_1 & s_2 & t & f \\
\hline
s_0 & 0 & 1 & 0 & 0 & 0 \\
 & 0 & 0 & 1 & 0 & 0 \\
\hline
s_1 & 0 & 0 & 0 & 0.8 & 0.2 \\
 & 0 & 0 & 0 & 0.6 & 0.4 \\
\hline
s_2 & 0 & 0 & 0 & 0.4 & 0.6 \\
 & 0 & 0 & 0 & 0.2 & 0.8 \\
\hline
t & 0 & 0 & 0 & 1 & 0 \\
f & 0 & 0 & 0 & 0 & 1
\end{array}$$

Figure 3.3: An example of quotient MDP of a family from Figure 3.1, together with its transition probability matrix **P** with row grouping.

Now let us look at how the synthesis itself works using this method. Consider Figure 3.4. The input of this method is a set of realisations $\mathcal{R}$, from which we create a quotient MDP that overapproximates the behaviour of all of them. Assume a safety property $\Phi \equiv \mathbb{P}_{\leq\lambda}[\Diamond G]$. During the model-checking, we compute minimizing and maximising schedulers σ_{min} and σ_{max}, respectively, together with the particular lower and upper bounds on the reachability probabilities $(\boldsymbol{x}_{min}, \boldsymbol{x}_{max})$. Now, if $\boldsymbol{x}_{max} \leq \lambda$, we know that each realisation for sure satisfies Φ and we can return any realisation as a solution. On the contrary, if $\boldsymbol{x}_{min} > \lambda$, then for each realisation $r \in \mathcal{R}^{\mathcal{D}}$, $\mathcal{D}_r \not\models \Phi$, and there is no feasible solution. Finally, if $\boldsymbol{x}_{min} \leq \lambda < \boldsymbol{x}_{max}$ then we cannot conclude anything unless $\mathcal{M}^{\mathcal{D}}_{\sigma_{min}}$ represents a valid family member, i.e. it is a solution. Otherwise, the problem remains undecided as the abstraction is too coarse. The solution to this problem is to refine this abstraction and split the family into two subfamilies, each of which will be analysed separately using the method described above. If the subfamilies are undecidable again, we continue to split them into even smaller ones until either we reject all family members or, find a feasible realisation.

Example 7 (AR illustration)**.** To demonstrate the AR method, we again use the family from Example 5 and the same property as in Example 6, i.e. $\varphi \equiv \mathbb{P}_{\leq 0.3}[\Diamond\{t\}]$. We begin with the set of all realisations and construct the corresponding quotient MDP depicted in Figure 3.3. Via MDP model-checking we obtain $\boldsymbol{x}_{min}(s_0) = 0.2 \leq 0.3 < 0.8 = \boldsymbol{x}_{max}(s_0)$. For a better demonstration of the method, let's pretend that the MC $\mathcal{M}^{\mathcal{D}}_{\sigma_{min}}$ is not a valid family member, so we got an undecidable case. Let us pick parameter X and split the set of all realisations into subsets $\mathcal{R}_1 = \{s_1\} \times \{t, f\}$ and $\mathcal{R}_2 = \{s_2\} \times \{t, f\}$[1]. For each set of realisations, the process is then repeated again: model-checking quotient MDP corresponding to the set $\mathcal{R}_1$ yields $\boldsymbol{x}_{min}(s_0) = 0.6 > 0.3$ allowing us to reject all realisations associated with this subfamily, and model-checking the second quotient MDP again results

[1]Note that the split operation is removing non-determinism in state s_0, which corresponds to the omission of a row in the transition probability matrix for individual subfamilies.



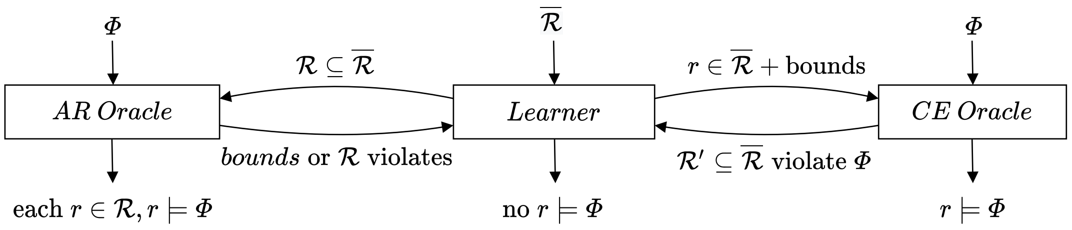
$\mathcal{R}$
abstract to quotient MDP
quotients
pick $\mathcal{M}$
verify $\mathcal{M} \models \Phi$
$\mathbb{P}[\mathcal{M} \models \Phi]$
refine quotient: split family
undecided
satisfied?
no
yes
unsatisfiable
feasible realisation

Figure 3.4: Abstraction refinement deductive synthesis.

in undecidable result. Splitting $\mathcal{R}_2$, this time at parameter Y, yields another two sub-cases: $\mathcal{R}_3 = \{s_2\} \times \{t\}$ and $\mathcal{R}_4 = \{s_2\} \times \{t\}$, each containing a single MC, from which we conclude that realisation r_3 represents the desired solution.

3.4 Hybrid Dual-Oracle Synthesis

As the two previous approaches are conceptually completely different, the hybrid method represents the middle ground between them, and seeks to take advantage of both methods. In particular, the presented hybrid method switches between CEGIS and AR methods. It estimates the efficiency of the two methods and allocate bigger time slot to the method which performs better (e.g. prunes more family members per time unit). In [3], they also found that the CE oracle can effectively use the bounds calculated during AR phase to create smaller counterexamples, which ultimately cause the faster pruning of the state-space of families in the CEGIS phase.



Φ
$\overline{\mathcal{R}}$
Φ
$\mathcal{R} \subseteq \overline{\mathcal{R}}$
$r \in \overline{\mathcal{R}}$ + bounds
AR Oracle
Learner
CE Oracle
bounds or $\mathcal{R}$ violates
$\mathcal{R}' \subseteq \overline{\mathcal{R}}$ violate Φ
each $r \in \mathcal{R}, r \models \Phi$
no $r \models \Phi$
$r \models \Phi$

Figure 3.5: Oracle-guided synthesis (adapted from [3]).

Chapter 4

Original Sequential Implementation of Paynt

In this chapter, we will focus on the original sequential implementation of the Paynt (Probabilistic progrAm sYNThesizer) tool [4], implementing the presented hybrid synthesis method. It is based on the probabilistic model checker Storm [14]. The focus will be on subroutines implementing MC and MDP model-checking, as they represent a core stage of the synthesis algorithm (to illustrate, see gray parts of figures 3.4 and 3.2). As it turns out, they are also bottlenecks of the whole process when working with large models because of the explosion of the underlying state-space of the chain. Note that some of the variables and function names were changed to better match the definition in this thesis.

4.1 Bottlenecks of Synthesis Process

To support our claim that model-checking is the main bottleneck when working with large models, we will show some profiling results. By running Paynt several times on families that included large models, we discovered that:

(1) The `model_checking` method really takes up the most time. From the obtained logs, we found that the time spent in this method is $\approx$ **68** % of the total time (note: it is called whenever it is necessary to verify a specific model, and only within this method, depending on the type of model, it is decided whether it will be MDP or MC model-checking.).

(2) About **22** % of the total time is taken by `construct_ce` method. It is called only in the CEGIS phase, if we find a family member who does not meet the verified property. Its task is to find the smallest counterexample violating this property (see Figure 3.2). Obviously, the percentage of the taken time depends on the number of (unsat) iterations of the CEGIS phase, and thus the value presented is only a rough estimate, valid for our test set.

(3) The remaining $\approx$ **10** % of the calculation will no longer be analysed in detail. It consists of several parts such as models building, SAT solving and overhead associated with controlling the synthesis process. From our perspective, these parts are uninteresting as most of them form the necessary overhead and therefore cannot be significantly accelerated.

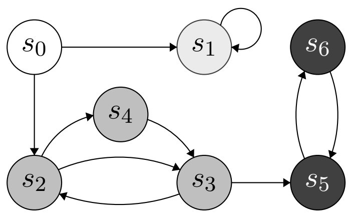


Figure 4.1: A directed graph with 4 maximal SCCs (distinguished by colours). The states s_0 and s_1 present a so-called trivial SCCs as they consist of only one state. The states s_2, s_3, and s_4 form a non-trivial SCC, just as states s_5 and s_6.

4.2 Implementation of the MC Model Checking

If one uses the STORM tool on the MC model-checking (as the PAYNT tool does), it is necessary to specify which solver will be used to solve the derived system of linear equations (see Algorithm 1, line 4) before calling the `model_checking` method. The original implementation uses a *topological solver* [36], which will be briefly introduced in this section.

4.2.1 Topological Solver

Instead of directly calculating the probability from initial states to targets, it divides the whole state-space into several partitions, and solve them individually. The idea behind this approach is that if there is a loop in the analysed system, where the convergence of associated probabilities take several steps, propagation of the intermediate changes to the outside present a waste. Therefore, the first step of this approach is finding *Strongly Connected Components* (SCCs). In graph theory, an SCC of a directed graph G is a subset of vertices, such that for each pair of vertices a and b, both a is reachable from b and b is reachable from a. Also, if the selected subset is maximal with respect to G, implying no vertices can be added without losing the property, the SCC is *maximal* (see Figure 4.1). The last important property of an SCC is to note that there exists no path that leaves the SCC and later returns, as this would contradict SCC's maximality. Based on the above, there are two possible cases for an SCC:

(1) only incoming transitions: The SCC is then called a *bottom* SCC (see Figure 4.1 which has two bottom SCCs, one consisting of states s_5 and s_6 and the other is just state s_1). Due to the fact that there is no way of leaving this SCC, the calculation of the reachability probabilities is independent of the remaining system. In addition, if the SCC does not contain a target state, the result is 0 for all its states.

(2) incoming *&* outgoing transitions: A topological ordering must be defined for these components, as the probability of one SCC directly affects the other through the outgoing transitions. Specifically, if there is an outgoing transition from a given SCC, the target SCC of that transition must be solved first. Looking at Figure 4.1, the order is given by the shade of gray, where a darker SCC must be resolved before a lighter one. Following the ordering, each SCC can be solved individually as a possibly smaller instance.

Each SCC is solved as a smaller system of linear equations, which ensures a faster convergence of this system. In order to solve them, Storm uses the GMRES[1] method implemented in the `Gmm++`[2] library.

Algorithm 3: Topological Linear Equation Solver used by Storm. Note that for simplicity, we present only the critical parts of the algorithm.

```
   Input: An MC $\mathcal{M} = (S, s_{init}, \mathbf{P})$, target states $G \subseteq S$
   Output: A probability vector $\boldsymbol{x}(s) = \mathbb{P}[s \models \Diamond G]$ for each $s \in S$
1  sortedSCCs ← createSortedSccDecomposition($\mathbf{P}$)
2  if sortedSCCs.size() == 1 then
3      solveFullyConnectedEquationSystem($\mathbf{P}$, $\boldsymbol{x}$)
4  else
5      foreach scc in sortedSCCs do
6          if scc.size() == 1 then
7              $\boldsymbol{x}$(scc) += $\mathbf{P}$.getRow(scc) · $\boldsymbol{x}$ // dot product
8              ...
9          else
               // solving the underlying equation system for the non-trivial SCC
10             solveSCCequationSystem($\mathbf{P}$, sccAsBitVector, $\boldsymbol{x}$)
11         end
12     end
13 end
14 return $\boldsymbol{x}$
```

4.2.2 Finding the Hotspots

As already mentioned, the benefit of the topological solver is a divide-and-conquer strategy, where the model is divided into smaller parts and each of them is solved as a smaller instance of the problem. Also, let us look at its implementation. It is clear that the topological solver brings a significant advantage in cases where the model contains many trivial SCCs, as they are solved by a simple dot product of two vectors (see Algorithm 3, line 7).

However, after trying this solver on a set of different models, we found that it does not scale at all, while working with models that contain a large number of non-trivial SCCs. Under these conditions, the SCC detection brings additional cost. Furthermore, each non-trivial component will be solved as a separate system of equations, which will adversely affect the total computation time with their large number.

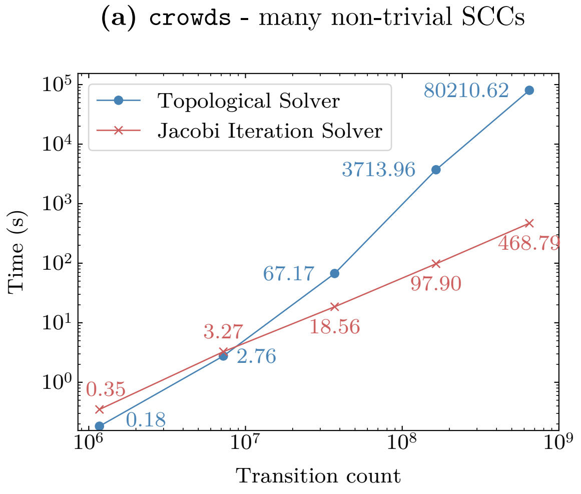
For illustration in Figure 4.2, we show preliminary results comparing the topological solver and the Jacobi method using the *crowds* [32] and the *nand* [26] models. To support our decision of choosing the Jacobi method for comparison, we state that it is a basic representative of iterative methods. It is also the default method of the state-of-the-art model-checker Prism [20] and will also play a vital role in the following chapters. As we can see in Figure 4.2a, even though the Jacobi method works with the whole state-space, it can outperform the topological solver by more than one order of magnitude while working with models containing a large number of non-trivial SCCs and fairly rapid convergence.

[1]Generalized Minimal Residual Method (GMRES) is an iterative method for the numerical solution of a nonsymmetric system of linear equations.

[2]`gmm++` – `http://getfem.org/gmm.html`

On the other side when there is no non-trivial SCC (or their number is reasonably small), the topological solver outperforms the Jacobi method (Figure 4.2b).

Because it is not easy to identify the number of non-trivial SCCs before starting model-checking, the user cannot tell which variant to use. So we came up with a metric that could at least roughly estimate which variant should be used. This metric is based on the idea that a greater number of transitions than states imply a more complex structure and a higher probability of non-trivial SCCs. For a more detailed evaluation, see Chapter 6.



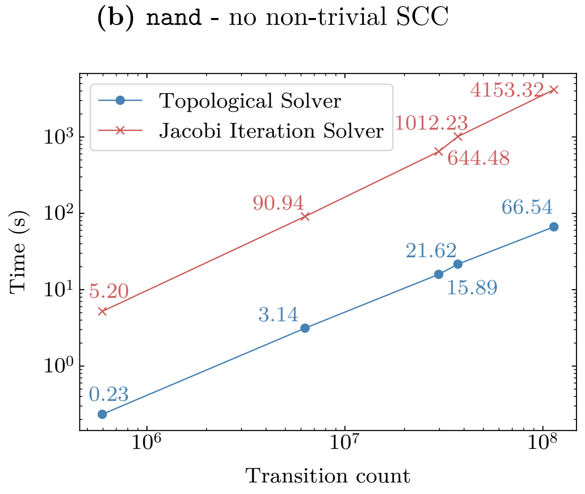


Figure 4.2: Comparison of topological solver and Jacobi iteration method on models with different number of SCCs. Notice the logarithmic scale on both axes of the graphs.

4.3 Implementation of the MDP Model Checking

For MDP model-checking, the original implementation uses the VI method. It is a straightforward translation of Algorithm 2 to `C++`. The pseudo-code of the `C++` implementation can be seen in Algorithm 4. From the algorithm, we can notice that before starting the calculation, Storm makes the reduction mentioned in Chapter 2, and thus considers only the states belonging to $S_?$ set.

4.3.1 Finding the Hotspots

In order to find limitations of the VI method, we ran it on a set of models, and, as expected, the method did not scale with the increasing model size and the increasing number of iterations. This means that the main problem was the loop shown in Algorithm 4.

In addition, since Storm also has a topological solver for MDP model-checking, we tried to compare it with the currently used VI algorithm. The idea of this topological solver is the same as described in Section 4.2.1, except that the VI algorithm is used over the individual SCCs. Then, matrix A in Algorithm 4 represents only one SCC. The solution of trivial SCCs will also change, and it will no longer be just a simple dot product as with an MDP transition matrix, more than one row of the matrix can belong to one state (depending on the number of non-deterministic actions in that state).

This comparison showed a similar trend as for MC model-checking. Thus, if the model contains a small number of non-trivial SCCs, the topological solver can overcome the cur-

Algorithm 4: Value Iteration algorithm as implemented in the STORM tool.

```
Input: Transition probability matrix $A$ over all states $s \in S_?$, static probability
       vector $b \in [0,1]^{|S_?|}$, initial state probability vector $x \in [0,1]^{|S_?|}$,
       row-grouping information $\mathcal{G}_A$ on $A$, reduce operator $\oplus \in \{min, max\}$,
       required precision $\varepsilon$
Output: A probability vector $\boldsymbol{x}(s)_{s \in S_?}$
1 do
2     multRes ← $A \cdot \boldsymbol{x} + b$
3     foreach $group \in \mathcal{G}_A$ do
4         $\boldsymbol{x'}(group) \leftarrow \oplus\{$multRes$(i) \mid i \in group\}$
5     end
6     swap($\boldsymbol{x}, \boldsymbol{x'}$)
7 while max$\{|x_s - x'_s| \mid s \in S_?\} \leq \varepsilon$
8 return $\boldsymbol{x}$
```

rently used VI algorithm. And again, as the number of non-trivial SCCs increases, the performance of the topological solver decreases faster. For more detail see Chapter 6.

4.4 Data Structures

In the previous sections, we have introduced methods that are currently used for model-checking. Nevertheless, we all know that not only the algorithms themselves but also the data structures are a crucial part of the implementation. The way the data is stored can often significantly affect the speed of the calculation. For this reason, in this section, we will introduce how the analysed models are stored. We can already see from Figure 2.1 that the transition matrix of a simple model simulating six-sided dice contains an great amount of zeroes, that play no role in the calculation. This trend is the same for most stochastic models and therefore, many ways that try to compress these *sparse* matrices have emerged.

The data representation in the STORM's sparse engine (which PAYNT is using) is based on *compressed sparse row* (CSR) matrix format, which will be briefly introduced in the rest of this section.

Definition 11 (The CSR Format)**.** Let $\mathbf{A} : \{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, n-1\} \to \mathbb{R}$ be a matrix with m rows and n columns, such that $\mathbf{A}(i,j) = a_{i,j}$ and $0 \leq i < m \wedge 0 \leq j < n$. Let $c = |\{\mathbf{A}(i,j) \mid \mathbf{A}(i,j) \neq 0\}|$ be the number of non-zero elements of $\mathbf{A}$. And let $rsize(i) = |\{\mathbf{A}(i,j) \mid \mathbf{A}(i,j) \neq 0\}|$ be the number of non-zero values in the i-th row of $\mathbf{A}$. Then $\mathbf{A}$ can be represented by the following three arrays:

- `matVal[`c`]`; `matVal[i]` is the `i`-th non-zero value of $\mathbf{A}$.
- `matCol[`c`]`; `matCol[i]` is the column of the `i`-th non-zero value of $\mathbf{A}$.
- `matRowStart[`$m+1$`]`; `matRowStart[j]` $= \sum_{i=0}^{i<j} rsize(i)$ and specifies the index in the `matVal` array where row `j` begins. `j` then ends at the index `matRowStart[j+1]-1`.

Example 8. Consider the MDP transition probability matrix depicted below.

$$\left(\begin{array}{ccc} 0.7 & 0.3 & 0 \\ \hline 0.5 & 0 & 0.5 \\ 0 & 1 & 0 \\ \hline 1 & 0 & 0 \end{array}\right)$$

In order to create a CSR representation, we first (line by line) extract all non-zero values, and store them in the `matVal` array. Then, for each value from the `matVal` array, we find its column and store it in the `matCol` array. The last array is simply created according to the above formula.

```
   matVal[] =  0.7  0.3  0.5  0.5  1  1
   matCol[] =    0    1    0    2  1  0
matStartRow[] =  0    2    4    5  6
```

In addition, in the case of an MDP transition matrix (as in our example), STORM defines an `matGroupStart` array, that groups the rows corresponding to the actions in one state. The semantics of this array is similar to `matRowStart` except that `mathGroupStart[j]` returns the row number on which the group `j` begins, and `mathGroupStart[j+1]-1` is then the last row of group `j`. In case of our example, this array would look like this:

```
matGroupStart[] =  0  1  3  4
```

The individual elements of matrices are of the `double` type, as the original sequential implementation requires accuracy of up to 11 significant digits. It follows that our parallel implementation presented in the next chapter will have to work with this accuracy, even though it is more advantageous to use single-precision when working on GPU.

Chapter 5

Parallelisation

In the last few years, the current trends in hardware development favour multicore and manycore architectures such as GPUs. To address the limitations mentioned in the previous chapters, this chapter will focus on finding those parts of synthesis algorithm that may be executed efficiently on GPU and take advantage of their highly parallel architecture. The aim is to increase the scalability of the presented synthesis method for both large and small models (if there are many). At the beginning of this chapter, we will describe which specific parts of the synthesis algorithm we have parallelized. Then we will say a few words about choosing the used GPU framework, and the last, the most crucial part of this chapter will be the introduction of our parallel methods.

5.1 Problem Decomposition

For a better idea of the computational distribution within the synthesis, Figure 5.1 shows the overall architecture of the PAYNT tool. As mentioned in Chapter 4, the most significant part of the calculation is occupied by model-checking when working with large models (see the green parts of Figure 5.1). Therefore, we decided to target this problem and we present a parallelisation on the state level of a given stochastic model. It is actually a parallelisation of the model-checking methods presented in the previous chapters, while all these methods work with one model (see Section 5.3). However, this parallelisation is not suitable for small models, as the overhead associated with GPU execution is higher than useful work.

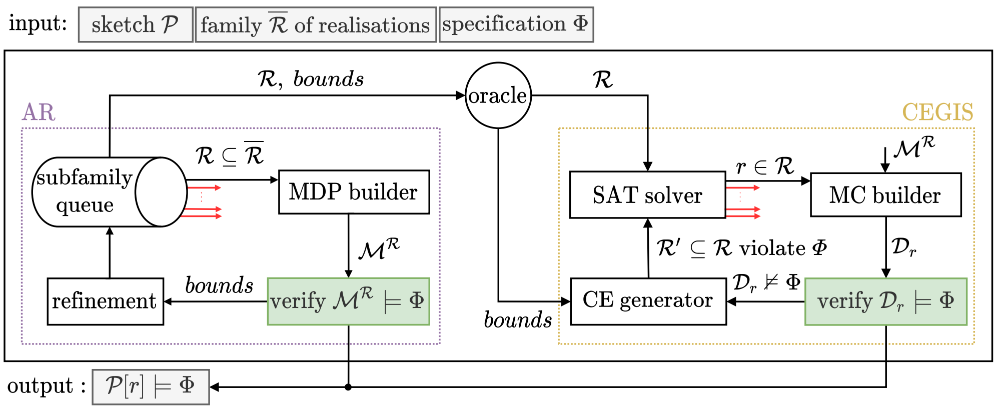

input:
sketch $\mathcal{P}$
family $\overline{\mathcal{R}}$ of realisations
specification Φ
$\mathcal{R}$, *bounds*
oracle
$\mathcal{R}$
AR
CEGIS
subfamily queue
$\mathcal{R} \subseteq \overline{\mathcal{R}}$
MDP builder
$\mathcal{M}^{\mathcal{R}}$
refinement
bounds
verify $\mathcal{M}^{\mathcal{R}} \models \Phi$
SAT solver
$r \in \mathcal{R}$
$\mathcal{M}^{\mathcal{R}}$
MC builder
$\mathcal{R}' \subseteq \mathcal{R}$ violate Φ
$\mathcal{D}_r$
$\mathcal{D}_r \not\models \Phi$
CE generator
verify $\mathcal{D}_r \models \Phi$
bounds
output :
$\mathcal{P}[r] \models \Phi$

Figure 5.1: The PAYNT tool architecture.

For this reason, we have proposed the so-called *family-based parallelisation*, where we try to increase the amount of work done on GPU by analysing several smaller models simultaneously. This situation is illustrated in Figure 5.1 by red arrows, which indicate that either multiple subfamilies (AR loop) or multiple realisations of a single family (CEGIS loop) are analysed in a single iteration. This approach is discussed in more detail in section 5.4.

5.2 Analysis CUDA/OpenCL

Ever since the field of *general-purpose computing on graphics processing units* (GPGPU) took off, the two main contenders on the market are AMD[1] and NVIDIA[2]. Both of them have been trying to attract as many customers as possible, not only in terms of the hardware offered, but also in terms of the tools and particular languages that allow optimal performance on their devices. Furthermore, since we directly influence the target hardware by choosing the framework, we present a list of advantages and disadvantages.

5.2.1 OpenCL

Open Computing Language (OpenCL) [25] serves as an independent, open standard for cross-platform parallel programming. OpenCL is not just for GPUs (like CUDA) but also for CPUs, FPGAs, and DSPs. It is maintained by a non-profit organization, and has been adopted by several large companies such as AMD, Intel, and NVIDIA. However, the heterogeneity of platforms brings the most significant disadvantage, which is that OpenCL cannot be very device-specific. This means that it is not possible to use the full potential of highly specialized hardware and low-level programming. The fact that OpenCL applications can run even on Android devices is irrelevant to the goals of this thesis.

5.2.2 CUDA

Compute Unified Device Architecture (CUDA) [24] serves as a platform for parallel computing, as well as a programming model. It was developed by NVIDIA for general-purpose computing on NVIDIA's GPU hardware. As a programming model, it is very similar to OpenCL, though the hardware abstraction layers differ in terminology and coarseness. In contrast to OpenCL, CUDA is specifically GPU-aimed, allowing it to focus more on available resources, which results in a better performance. Another advantage is that NVIDIA targets mainly the scientific community, and therefore there are many ready-made optimized libraries for different types of scientific computations. Based on these facts, we've decided on the CUDA programming model.

5.3 State-Based Parallelisation

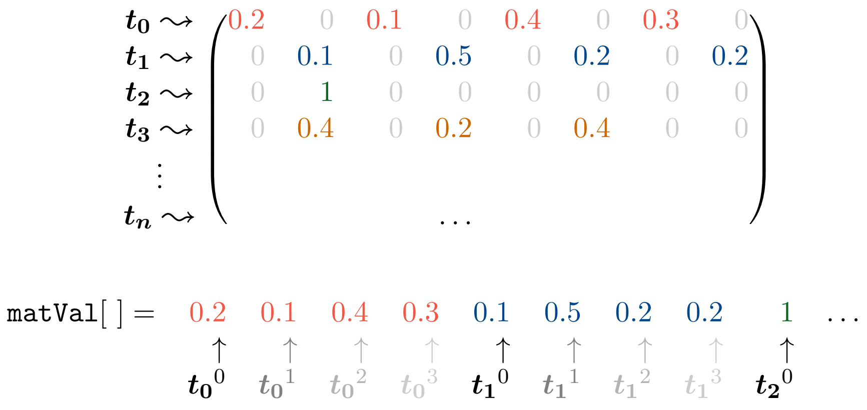
The first goal of our work is to use a large number of GPU threads to speed up the model-checking on huge models. To this end, we want to implement existing, sequential model-checking algorithms in CUDA, while the resulting implementation is parallel on the state level of a stochastic model.

[1] `https://www.amd.com/en`
[2] `https://www.nvidia.com/en-us/`

5.3.1 Matrix-Vector Multiplication

As shown in Chapter 2, matrix-vector multiplication plays a crucial part in the model-checking of stochastic models, and therefore it was the first step of our parallelisation. Despite the fact that it is a very well-known problem addressed by the vast community, we decided to parallelize it (ignoring the existing approaches) to better understand the CUDA programming model, and possibly improve the situation in this area.

Assume the matrix in Figure 5.2 and its CSR representation as presented in Section 4.4. For simplicity, let us also consider that one warp contains only four threads. The simplest, most straightforward variant of how to parallelize this calculation is to have each thread compute one element of the resulting vector. Thus, one row of an input matrix belongs to one thread (see Figure 5.2). The implementation of such an approach then looks like in Algorithm 5.



t_0
t_1
t_2
t_3
t_n
0.2 0 0.1 0 0.4 0 0.3 0
0 0.1 0 0.5 0 0.2 0 0.2
0 1 0 0 0 0 0 0
0 0.4 0 0.2 0 0.4 0 0
matVal[ ] = 0.2 0.1 0.4 0.3 0.1 0.5 0.2 0.2 1 ...
t_0^0 t_0^1 t_0^2 t_0^3 t_1^0 t_1^1 t_1^2 t_1^3 t_2^0

Figure 5.2: CSR thread per row - memory access pattern.

Algorithm 5: CSR Naive (thread per row) matrix-vector multiplication kernel.

```
row = blockIdx.x * blockDim.x + threadIdx.x; // thread's global ID
sum = 0.0;
foreach i = matStartRow[row]; i < matStartRow[row+1]; i++ do
    sum += matVal[i] * x[matCol[i]];
end
result[row] = sum;
```

But let's go back to Figure 5.2 showing also the biggest problem with this implementation, which is non coalesced memory access. We see that threads in the same warp read values with a (row size) stride. Can we improve it somehow? The answer to the previous question is shown in Figure 5.3. We see that, unlike the first approach, one line is now processed by the threads of one warp. Each thread then performs `matVal[tid] * x[matCol[tid]]`, and stores the result in its local register. The resulting reduction is performed using warp-level primitives (working directly with the registers, and therefore no additional memory is required). This approach solved the problem of non coalesced memory access, but if we look at the picture below, we see that a third row contains only one non-zero value, which results in unwanted divergence of threads within one warp (all threads except first are idle).

$$\texttt{matVal}[\,] = \begin{array}{cccc||cccc||cccc||c} 0.2 & 0.1 & 0.4 & 0.3 & 0.1 & 0.5 & 0.2 & 0.2 & 1 & - & - & - & \dots \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \\ \boldsymbol{t_0}^0 & \boldsymbol{t_1}^0 & \boldsymbol{t_2}^0 & \boldsymbol{t_3}^0 & \boldsymbol{t_4}^0 & \boldsymbol{t_5}^0 & \boldsymbol{t_6}^0 & \boldsymbol{t_7}^0 & \boldsymbol{t_8}^0 & t_9{}^0 & t_{10}{}^0 & t_{11}{}^0 & \end{array}$$

Figure 5.3: CSR warp per row - memory access pattern.

In order to eliminate this problem as well, we present our final method. In this approach, we also used the `matGroupStart` array, which Storm uses to define MDPs. However, we have slightly changed its semantics. Instead of grouping rows belonging to non-deterministic actions in one state, we will now group rows to ensure enough work for all threads in one block. Assume the matrix in Figure 5.2 and the condition of four non-zero elements per block `NNZ_PER_BLOCK = 4`. Then the `matGroupStart` array will look like: `[0, 1, 2, 4, ..., n]`. We see that the third and fourth row have merged into one group, removing the divergence problem from the previous approach (see Figure 5.4). The value of `NNZ_PER_BLOCK` is optional and also specifies the size of one thread block. Each thread block then gets a maximum of `NNZ_PER_BLOCK` non-zero elements, which are processed by `NNZ_PER_BLOCK` threads. Blocks are created in groups of rows, so the block either contains the entire row or no row element at all (if the `NNZ_PER_BLOCK` value would be exceeded by adding the row). Therefore, it is clear that our modification can also lead to a situation where some threads in the block will be idle. However, their number will be significantly smaller than in the previous approach.

The last thing that remains is the reduction of the resulting values. There is no longer a situation where one thread or one warp processes the entire row. There may be situations where multiple warps can process one row, and even threads in one warp do not have to process the same row (as in Figure 5.4). It follows that the reduction can no longer be done by the only one thread or warp. More precisely, it can, but we can do better. Therefore, our approach decides at run-time which method will be chosen based on the size of the individual rows in the block (hence the name *adaptable*). Specific situations that may occur:

- One block processes only one row of a matrix. It is exactly the case with the first two rows of our example, and we reduce it using one or more warps, depending on the number of non-zero values in the row. If one row contains more than `2 * warpSize` values, we cannot simply reduce them using the mentioned warp level

$$\texttt{matVal}[\,] = \begin{array}{cccc||cccc||cccc||c} 0.2 & 0.1 & 0.4 & 0.3 & 0.1 & 0.5 & 0.2 & 0.2 & 1 & 0.4 & 0.2 & 0.4 & \dots \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \\ \boldsymbol{t_0}^0 & \boldsymbol{t_1}^0 & \boldsymbol{t_2}^0 & \boldsymbol{t_3}^0 & \boldsymbol{t_4}^0 & \boldsymbol{t_5}^0 & \boldsymbol{t_6}^0 & \boldsymbol{t_7}^0 & \boldsymbol{t_8}^0 & \boldsymbol{t_9}^0 & \boldsymbol{t_{10}}^0 & \boldsymbol{t_{11}}^0 & \end{array}$$

Figure 5.4: CSR adaptable - memory access pattern.

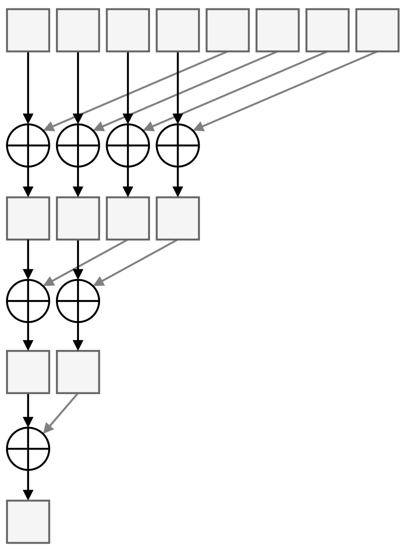
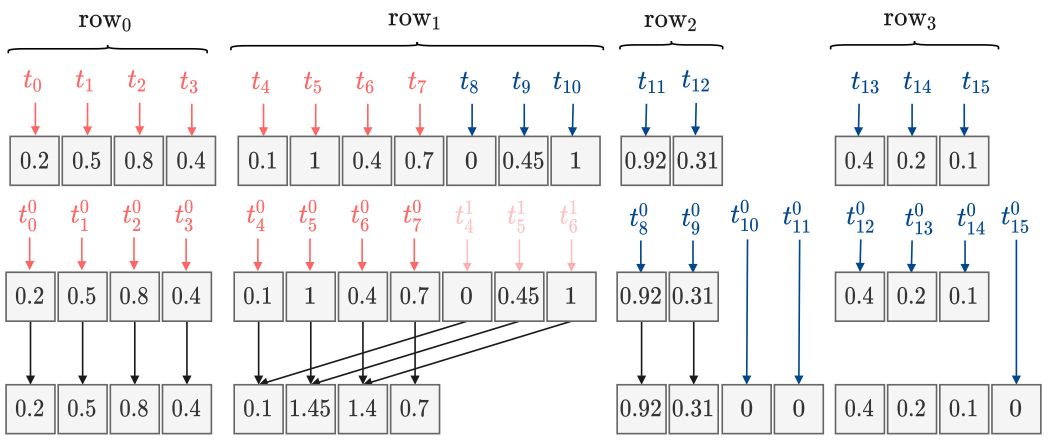


Figure 5.5: An example of parallel reduction using the pyramid-summation scheme. Gray squares represent memory cells, and a $\oplus$ represents a commutative binary operation performed by a single thread. Note that with GPU, we must use the memory access pattern as shown; otherwise, thread divergence will occur in a single warp.

primitives and so more warps will be used for reduction. Each of them reduces an assigned part of a row, and the results are reduced in shared memory applying the pyramid-summation scheme (see Figure 5.5).

- The second case is that one block contains several rows (see the third block in our example). Here the reduction is more complicated as one block computes several values of the resulting vector. Based on the number of rows in the block, we decide how many threads will be used to reduce one row: `prevPowerOf2(blockSize/rowsInBlock)`. For a better understanding, let us show it again with an example. This time consider the situation shown in Figure 5.6. The figure shows one block consisting of 4 rows, and the colours of the threads correspond to the warp membership (this time, eight threads per warp). The first line shows the situation just after multiplication. Each thread in a block writes the calculated value to shared memory on an index corresponding to its `threadId.x`. In the second step, the role of the threads changes. From the above formula, we get that up to 4 threads will reduce one row. If the given row has more than four values, it is necessary that the threads first process all values of the given row locally (in the registers). Threads with nothing to do in this step will write 0 (neutral element for sum operation) to their register. After this operation, each thread writes the value of its register to the shared memory, and there will be precisely four values for each row. Finally, we use the pyramid scheme over the shared memory, as shown in Figure 5.5. In the end, the threads 0, 4, 8, 12 write the results to the global memory. If `prevPowerOf2(blockSize/rowsInBlock)=1`, each row is assigned one thread, and the situation is pretty straightforward. We see that with this approach, an inevitable divergence occurs in the reduction, but a significantly larger part of the threads does valuable work.



row$_0$
row$_1$
row$_2$
row$_3$
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12} t_{13} t_{14} t_{15}
0.2 0.5 0.8 0.4 0.1 1 0.4 0.7 0 0.45 1 0.92 0.31 0.4 0.2 0.1
t_0^0 t_1^0 t_2^0 t_3^0 t_4^0 t_5^0 t_6^0 t_7^0 t_4^1 t_5^1 t_6^1 t_8^0 t_9^0 t_{10}^0 t_{11}^0 t_{12}^0 t_{13}^0 t_{14}^0 t_{15}^0
0.2 0.5 0.8 0.4 0.1 1 0.4 0.7 0 0.45 1 0.92 0.31 0.4 0.2 0.1
0.2 0.5 0.8 0.4 0.1 1.45 1.4 0.7 0.92 0.31 0 0 0.4 0.2 0.1 0

Figure 5.6: Example of reduction used in matrix-vector multiplication. This example demonstrates a situation where one thread block reduces multiple rows of the input matrix. The colours represent the association of the thread to the warp.

In order to evaluate the performance of our implementation, we compared it with the implementation from the CUSPARSE[3] library. Specifically, it was the implementation from the CUDA Toolkit ver. 11 [24]. For evaluation, we used a set of sparse matrices from the benchmark used in Chapter 6, and the comparison can be seen in Figure 5.7. From the results, we can see that we were able to outperform the CUSPARSE library on most selected matrices. However, despite the favourable results of our implementation, we used the CUSPARSE library to continue our work. The overhead associated with rows grouping unnecessarily brings more sequential work to the algorithm, as the CUSPARSE library does not need this information, and it was this overhead that caused smaller speed-ups when verifying large models using our multiplication.

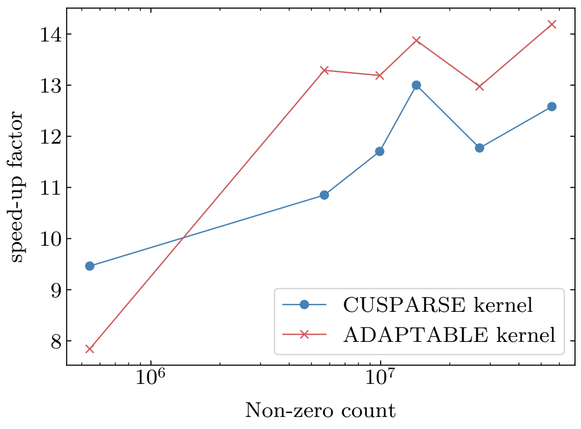


Figure 5.7: Comparison of our implementation of matrix vector multiplication with the CUSPARSE library.

[3]CUSPARSE is the CUDA sparse matrix library. It contains a set of basic linear algebra subroutines used for handling sparse matrices.

5.3.2 Model Checking MCs in parallel

In previous chapters, we have shown that MC model-checking is an essential part of the synthesis algorithm, and we have also introduced ways to solve this problem. In order to parallelize this problem, let us take a brief look at the individually presented methods, focusing on their parallel potential:

(1) *Topological Solver*: First, we will look at probably the most sophisticated approach, which is also set as STORM's default – the topological solver. The first idea that might strike someone is that it solves each SCC separately; couldn't it do it in parallel? However, this idea is rejected by the fact that the individual SCCs must be solved in the order given by the topological ordering (see Section 4.2.1). The possibility of parallelism is thus represented only by SCCs, which do not affect each other in any way. However, the identification of such SCCs would bring additional overhead to the calculation, which is not the biggest problem. Much more significant obstacles are their different properties and the fact that they are not stored contiguously in memory. In addition, if the number of these independent components is small, much of the calculation will still be serialized. These problems would result in unwanted thread divergence and poor memory access from the GPU's perspective. From the above, we can say that GPU parallelisation of this solver is not very suitable, as with GPU, we benefit from the fact that many threads do the same job.

(2) *Gauss-Seidel Method*: The second, much better known, is the Gauss-Seidel method. If we compare it with the previous approach, this method will no longer have a problem with memory access, as it works with the whole state-space at once. And as we have shown in the previous section, we can ensure that adjacent threads read from adjacent locations. This approach can even converge quite quickly. So, where is the problem? The problem, or rather the difficulty of parallelisation, of this method, is due to the data dependency. If we look at the update scheme of this method, we see that it already uses the currently calculated values in the current iteration. As a result, this method converges relatively quickly, but in a parallel environment, it will result in unwanted synchronization of threads.

(3) *Jacobi Method*: The last one and at the same time the one we have decided to parallelize is the Jacobi method. We chose this method even though it is one of the slowest in a sequential environment. However, unlike the Gauss-Seidel method, it has no data dependencies within a single iteration and is therefore very suitable for parallelisation.

Jacobi Iterations. Each iteration in the Jacobi method involves a matrix-vector multiplication. Let s be the size of the state-space, implying the dimension $s \times s$ of matrix $\boldsymbol{A}$ to be iterated. Now let us recall the update scheme of the Jacobi method, and identify the parts that can be done in parallel. For simplicity, we will adjust it a bit compared to the version given in Chapter 2. Specifically, we extract the diagonal elements of matrix $\boldsymbol{A}$ and store their inverted values in array $D_i = {}^{1}\!/\!{}_{A(i,i)}$, for $i \in \{0, \dots, s-1\}$ before applying the update scheme. The modification then yields:

$$\boldsymbol{x}^{(k)}(i) = D_i \cdot \left(\boldsymbol{b}(i) - \sum_{j \neq i} \boldsymbol{A}(i,j) \cdot \boldsymbol{x}^{(k-1)}(j) \right)$$

The pseudo code of the STORM's sequential implementation, which already uses the decomposition given above is shown in Algorithm 6. This algorithm already gives us a nice guide to our parallelisation. Let us look at the first step of the iterative calculation (line 3). We see that this is exactly the problem presented in the previous section, namely the multiplication of the sparse matrix and a dense vector. As already mentioned, our implementation uses the CUSPARSE library for this. Lines 4 and 5 of the sequential algorithm essentially do the operation:

$$\boldsymbol{x'}_i = (b_i - \boldsymbol{x'}_i) * D_i$$

In order to perform this operation, we created a kernel containing s threads. Each thread then calculates one entry of the resulting vector $\boldsymbol{x'}$ based on its `tid`. The final part of Algorithm 6 is checking convergence, which can be also done in parallel. In order to do so, we have defined a custom CUDA binary transformation operator for both relative and absolute comparison. It takes two arguments and returns a single positive value. This transformation is then applied to each pair $(\boldsymbol{x}_i, \boldsymbol{x'}_i)$ for $i \in \{0, \dots, s\}$. Now, it remains to find the maximum difference in the resulting vector. In the case of a simple vector-wide reduction, there is a possibility to use the Thrust library [15], implementing an optimized version of pyramid-scheme reduction mentioned earlier. Computed maximum is compared against the given precision value, and either the algorithm terminates or the next iteration begins.

The satisfactory condition for the Jacobi method to converge is that the magnitude of spectral radius (the largest eigenvalue) of matrix $D^{-1}(A - D)$ is bounded by value one. Luckily, the Perron-Frobenius theorem declares that spectral radius of a stochastic matrix is equal to 1, and all other eigenvalues are smaller than one, so that $lim_{k \to \infty} A^k$ exists. The worst case scenario is that the number of iterations is exponential in the size of the state-space, but in practice the number of iterations k is often moderate for some sufficiently small ε [34].

Algorithm 6: Jacobi method as implemented in the STORM tool.

```
   Input: CSR representation of transition probability matrix $A$ over all states
          $s \in S_?$, static probability vector $b \in [0,1]^{|S_?|}$, initial state probability vector
          $x \in [0,1]^{|S_?|}$
   Output: A probability vector $\boldsymbol{x}(s)_{s \in S_?}$
   // $LU$ represents matrix $A$ without diagonal component.
1  $D, LU \leftarrow$ getJacobiDecompostion($A$)
2  do
3      $\boldsymbol{x'} \leftarrow$ multiply($LU, \boldsymbol{x}$)
4      $\boldsymbol{x'} \leftarrow$ subtractVectors($\boldsymbol{x'}, b$)
5      $\boldsymbol{x'} \leftarrow$ multiplyVectorsPointwise($\boldsymbol{x'}, D$)
6      swap($\boldsymbol{x}, \boldsymbol{x'}$)
7  while max$\{|x_s - x'_s| \mid s \in S_?\} \leq \varepsilon$
8  return $\boldsymbol{x}$
```

5.3.3 Model Checking MDPs in parallel

Another building block of the synthesis algorithm that has a problem with large models is MDP model-checking. In previous chapters, we have introduced two approaches to this model-checking. One was the VI algorithm used by the current version of PAYNT. The second was a more sophisticated topological approach used as the default of the Storm tool. When deciding which approach is more suitable for parallelisation, we came to the same conclusions as in MC model-checking. The disadvantages of the topological approach remain precisely the same. On the other hand, an iterative calculation using matrix multiplication, which we have already successfully parallelized, has proved to be a clear choice. Therefore, we paralleled the value iteration algorithm.

Value Iterations. We have already introduced a sequential variant of the VI algorithm earlier in Algorithm 4. If we compare it with the algorithm of the Jacobi method, we see that the iterative parts of the algorithms do not differ so much from each other. In the beginning, we do a multiplication operation between a sparse matrix and a vector. We will use the CUSPARSE library again. Another part is the addition of the vector b. Since this is a simple vector operation, we will use the Thrust library again (vector operation with *plus* operator).

The implementation up to this point was more or less the same. So far, we have found the probability value for each action in each state. The next step is to select an action that maximizes or minimizes the value in each state based on the verified property (see Algorithm 4 lines 3-5). We will refer to this operation as *segmented reduce*, where one segment will represent actions in one state. First, we tried to find existing solutions. While searching, we came across the CUB library [28]. Specifically, it was a function `cub::DeviceSegmentedReduce`, whose interface exactly corresponded to our needs, and therefore we did not have to modify the used data structures. However, in our case, the segments representing non-deterministic choices are quite small, and the function just mentioned is not optimized for such cases. Using this function even led to an overall slowdown compared to the sequential variant, so we proposed our solution to target this problem.

To illustrate our approach, we will use the example in Figure 5.8. For simplicity, consider that the size of a thread block is 16 threads and that one warp consists of 8 threads (the condition is `blockSize % warpSize == 0`). In the example, we see the situation after the first two steps of the algorithm when we have calculated the probabilities for each action in each state. The first step is to decide how many threads will reduce the values belonging to one state. We determine this number using the formula: `n = prevPowerOf2(numEntries/numGroups)`. However, the condition is that the number of threads processing the actions of one state never exceeds the `warpSize` value (in our case `warpSize = 8`), and at the same time, it will never be less than 2. The next step is the calculation itself. Let us go back to our figure. If the number of elements in a segment is greater than the number of its threads, each thread first reduces the remaining values locally to its register in the manner shown in the figure. In our example, the *max* operation is used for reduction. The last step is the resulting reduction. We can use fast warp-level primitives for this reduction, as one state will never reduce more than `warpSize` threads. As we can see, warp level primitives also use the pyramid scheme from Figure 5.5, but their advantage is that they work directly with registers. Threads that have the resulting values in their registers will eventually write them to global memory. To summarize, based on the finding that one state rarely has more than `warpSize` actions (warp size is typically 32 or 64), we have proposed a segmented reduction method that can effectively utilize fast warp-level primitives.

The last part of the VI algorithm is the convergence detection. However, this part has not changed compared to one used in the Jacobi method, and thus the implementation of this part is exactly the same.

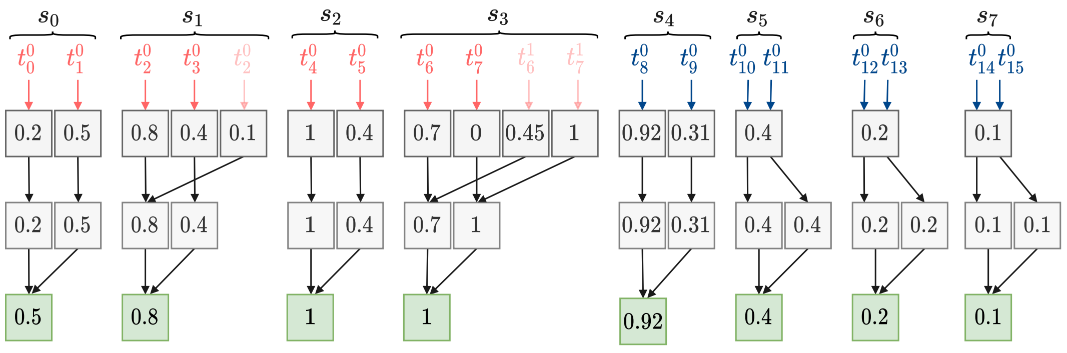


Figure 5.8: Example of segmented reduction used in the Value Iteration algorithm. s_i represents one state of an analysed MDP, and the colours again represent the association of the thread to the warp.

5.4 Family-Based Parallelisation

So far, we have talked about parallelisation at the state level of the stochastic model, which has brought acceleration when working with large models. The second part of our work deals with the idea of whether we cannot take advantage of a large number of threads even when working with small models (small matrices). Specifically, is there a way to analyse multiple smaller models at once?

5.4.1 Model Checking Multiple MDPs in parallel

First, we take a look at the possibility of analysing multiple MDPs simultaneously and for this purpose, let us recall Example 7. In this example, we have shown the operation of the AR synthesis method, which initially creates an abstraction of the whole family in the form of a quotient MDP and gradually refines this MDP during the synthesis. We also present an intuition that family refinement is actually omitting rows from a super-family transition matrix. And since we can describe all subfamilies with one transition matrix, this is why the idea arose to analyse several such families at the same time. The intention is to select as many MDPs from the quotients queue (see Figure 3.4) as can fit on the GPU and analyse them at once.

Consider the situation of Example 7, after the first refinement of the super-family, we have created two families, one of which omitted the first row of the original matrix and the other omitted the second row. Let us now look at how our method will analyse these families simultaneously. If you remember, when analysing one MDP, it was possible to reduce the state-space by considering only $S_?$ states. For this purpose, we then encoded the information about the states S_1 into vector b. However, in the case of the analysis of several families simultaneously, there may be a situation where a state that is not relevant for one family may be a relevant for another. Therefore this reduction is not possible in our approach. However, this is what comes in handy as we can use the vector b to encodes information about rows of the original matrix that are irrelevant for a given subfamily. For our two subfamilies, the vectors b will then look like this:

$$\begin{array}{l} b_{\text{subfamily1}} \\ b_{\text{subfamily2}} \end{array} \begin{pmatrix} \pm\infty & 0 & \vline & 0 & 0 & \vline & 0 & 0 & \vline & 0 & 0 \\ 0 & \pm\infty & \vline & 0 & 0 & \vline & 0 & 0 & \vline & 0 & 0 \end{pmatrix}$$

Specifically, we set the infinity value to the locations of irrelevant rows of the matrix for the given subfamily, which will play a vital role in the resulting reduction. However, by changing the semantics of vector b, we lost information about the target states. Thus, according to the original approach presented in Chapter 2, we encode this information into vector $\boldsymbol{x}$. Recall the property of Example 7: $\varphi \equiv \mathbb{P}_{\leq 0.3}[\lozenge\{t\}]$. So the initialization vectors $\boldsymbol{x}$ for our families will be:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}$$

After the introduction of these data structures, the VI algorithm itself will not require any significant changes. The first change is that the matrix vector multiplication changes to the multiplication of a sparse matrix and a dense matrix. The dense matrix will represent vectors $\boldsymbol{x}$ for individual families (the more families analysed simultaneously, the more columns this matrix will have). To implement this operation, we again used the CUSPARSE library and specifically the `cusparseSpMM` method. The result of this multiplication is a dense matrix, the columns of which represent the result for each family. This matrix is then stored in memory using the column-major order (since we are working with the GPU), which means that results of individual families are stored in memory as successive vectors, creating one long vector. As a result, the second step of algorithm VI remains the same, except that the added vector b represents the concatenation of vectors b for all families. The calculation for the families from our example would look like this (suppose we are looking for the maximum probability):

$$\begin{array}{c} \\ s_0 \\ \\ s_1 \\ \\ s_2 \\ \\ t \\ f \end{array}
\begin{array}{c} \begin{array}{ccccc} s_0 & s_1 & s_2 & t & f \end{array} \\
\left(\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\ \hline
0 & 0 & 0 & 0.8 & 0.2 \\
0 & 0 & 0 & 0.6 & 0.4 \\ \hline
0 & 0 & 0 & 0.4 & 0.6 \\
0 & 0 & 0 & 0.2 & 0.8 \\ \hline
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{array}\right) \end{array}
\cdot
\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \end{pmatrix}
=
\left(\begin{array}{cc}
0 & 0 \\
0 & 0 \\ \hline
0.8 & 0.8 \\
0.6 & 0.4 \\ \hline
0.4 & 0.6 \\
0.2 & 0.8 \\ \hline
1 & 1 \\
0 & 0
\end{array}\right)$$

$$\begin{array}{rl}
\texttt{mulRes} & (\ 0 \quad 0 \mid 0.8 \quad 0.6 \mid 0.4 \quad 0.2 \mid 1 \quad 0 \parallel 0 \quad 0 \mid 0.8 \quad 0.6 \mid 0.4 \quad 0.2 \mid 1 \quad 0) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\quad + \\
b & (-\infty \quad 0 \mid 0 \quad 0 \mid 0 \quad 0 \mid 0 \quad 0 \parallel 0 \quad -\infty \mid 0 \quad 0 \mid 0 \quad 0 \mid 0 \quad 0) \\ \hline
 & (-\infty \quad 0 \mid 0.8 \quad 0.6 \mid 0.4 \quad 0.2 \mid 1 \quad 0 \parallel 0 \quad -\infty \mid 0.8 \quad 0.6 \mid 0.4 \quad 0.2 \mid 1 \quad 0)
\end{array}$$

We can now apply a segmented reduction algorithm to the resulting vector. Note that as we look for maximum probability, setting irrelevant lines to minus infinity will ensure that they are always „ignored“ during reduction. The resulting vector is actually a matrix of vectors $\boldsymbol{x}$, serving as input for the next iteration. The last step is to check convergence. Here again, we benefit from the situation that the results for all families are in one „vector“, and so we can use exactly the same way as in the previous algorithms. This will check the convergence of all families at once, and the calculation will continue until all analysed families converge.

5.4.2 Model Checking Multiple MCs in parallel

It follows from Definition 10 that the MDP in the previous section over-approximates the behaviour of all members of the analysed family. It means that by omitting the correct rows of the transition matrix of this MDP, we can get all the correct members of this family. Thus, we could analyse multiple MCs simultaneously using a similar approach. However, as we will show in Chapter 6, this approach does not prove to be very advantageous, so we no longer tried to apply it to MC model-checking.

Chapter 6

Experimental Evaluation

All of the presented methods discussed in Chapter 5 were implemented in the Storm tool. Afterwards, they were made available through the Storm Python API and subsequently integrated into the Paynt tool. In the following set of experiments, we will demonstrate a comparison of our GPU aided implementation with the approaches used so far. We will also investigate the impact of the new methods on the overall performance of the synthesis. All of the experiments were run on a machine running Ubuntu 20.04 with an Intel XeonE5-2620 processor (6 cores up to 3.20 GHz) and using up to 64 GB RAM. Parallel model-checking was run on the NVIDIA GeForce GTX 1080 graphics card with 8 GB video memory.

6.1 Model Checking MCs

First, let us evaluate the MC model-checking. Here we will investigate how our GPU-aided implementation performed against the originally used topological solver, and also against the sequential version of the Jacobi method, as it is the method we've been accelerating, and in some cases, it is able to outperform the currently used topological solver (see Section 4.2.2). For each benchmark, we present two tables giving an overview over states and transitions count, average measured time of a CPU solver and our GPU based implementation and the factor between the two aforementioned times. In addition, if we compare our solver against the Jacobi method, the table also contains the number of iterations required for the method to converge. In comparison with the topological solver, we also state the number of non-trivial SCCs, as their number significantly affects the resulting factor. We also present a plot of the speed-up factor and the percentage of time spent in the GPU solver against the number of transitions to illustrate better the relationship between the size of the model and possible speed-up. A speed-up factor >1 means that our implementation performs better than a sequential one. This boundary is then shown in plots by a red line.

***crowds* case study.** Firstly we used the *crowds* protocol by Reiter and Rubin [32]. From our perspective, we are not so interested in the semantics of the investigated model, as in its structural properties. We found out, that these are the ones that have a significant impact on the overall performance of model-checking for different types of solvers. Thus, the crowds model is representative of models with a large number of non-trivial SCCs. The results for the crowds protocol are presented in Tables 6.1 and 6.2 and also in Figure 6.1.

Table 6.1: Experimental results using the *crowds* models, comparing our implementation against the sequential Jacobi algorithm.

states	transitions	iterations	CPU time	GPU time	factor
321 k	762 k	242	0.575 s	0.764 s	0.75
2.3 M	6.9 M	243	4.414 s	1.915 s	2.31
10 M	37 M	243	18.563 s	5.952 s	3.12
45 M	164 M	263	1.6 min	27.001 s	3.63
178 M	647 M	282	7.8 h	112.594 s	4.16

Table 6.2: Experimental results using the *crowds* models, comparing our implementation against the sequential topological algorithm.

states	transitions	SCCs*	$\frac{transitions}{states}$	CPU time	GPU time	factor
321 k	762 k	2.0 k	2.37	0.564 s	0.764 s	0.73
2.3 M	6.9 M	7.6 k	2.98	7.982 s	1.915 s	4.17
10 M	37 M	21 k	3.60	67.172 s	5.952 s	11.29
45 M	164 M	106 k	3.62	1 h	27.001 s	137.54
178 M	647 M	460 k	3.64	22.3 h	112.594 s	716.16

* Indicates the number of non-trivial SCCs.

(a) factor against topological solver **(b)** factor against seq. Jacobi

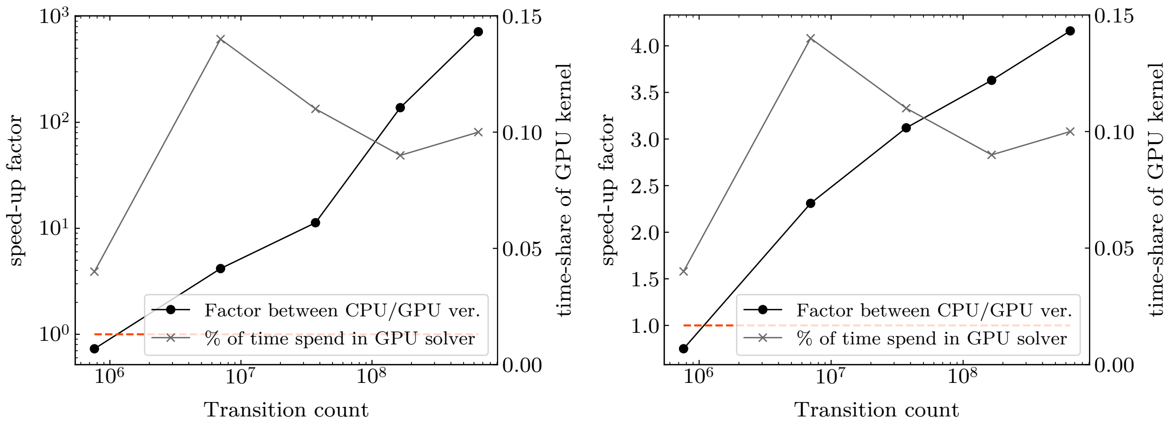


Figure 6.1: Speed-up factor and the time spent within the GPU solver plotted against the transition count of the *crowds* model.

The obtained results directly reveal several conclusions. Firstly, it is clear that a more preferred sequential variant, in this case, is the Jacobi iteration solver. The reason is that the convergence problem, that the topological solver targets, is not found in this model, and even in large instances, the Jacobi solver can converge up to hundreds of iterations. Furthermore, we can notice that the time of the topological solver snowballs with an increasing number of non-trivial SCCs, and thus our implementation is up to 700 times faster. The rapid convergence that caused this significant acceleration, damaged the acceleration compared to the sequential version of the Jacobi method. The small number of iterations meant that the time spent in our solver was only ≈10 % of the total computational time, which results in a maximum quadruple acceleration, although the parallel Jacobi method was much faster than the sequential one (see Amdhals' law[1]).

***maze* case study.** The second representative in our benchmark is the *maze* model [23]. In terms of structural properties, this model is representative of models with no or very few non-trivial SCCs. The results for the *maze* protocol are presented in Tables 6.3 and 6.4 and also in Figure 6.2.

At first, we want to point out the significantly higher number of iterations compared to the *crowds* model, which corresponds to the increase of time spent in our solver up to 95%. Under these conditions, we can notice that our implementation outperform its sequential variant up to twenty times. However, if we look at the comparison with the topological solver, we see a significant slowdown. From the topological solver's point of view, the model-checking is more or less a one matrix-vector multiplication, as the model contains only one non-trivial component (see Algorithm 3). Compared to the 20,000 multiplications (albeit performed in parallel) and the overhead associated with executing on the GPU, it has brought a significant speed-up against our implementation.

These results show that in cases where the model does not contain any non-trivial components (or their number is reasonably small), the most advantageous variant is the topological solver. We can identify this with our metric ($\frac{transitions}{states}$), the value of which implies that the number of transitions and states is close, and thus the formation of non-trivial SCCs is less likely.

Table 6.3: Experimental results using the *maze* models, comparing our implementation against the sequential Jacobi algorithm.

states	transitions	iterations	CPU time	GPU time	factor
809	1.2 k	18 k	0.227 s	1.279 s	0.18
7.4 k	11.1 k	19 k	1.630 s	1.353 s	1.20
74 k	110 k	19 k	16.084 s	1.958 s	8.21
740 k	1.1 M	20 k	3 min	13.451 s	13.19
7.4 M	11 M	20 k	30.2 min	80.675 s	22.49

Table 6.4: Experimental results using the *maze* models, comparing our implementation against the sequential topological algorithm.

states	transitions	SCCs*	$\frac{transitions}{states}$	CPU time	GPU time	factor
809	1.2 k	1	0.66	0.001 s	1.279 s	7.8e−4
7.4 k	11.1 k	1	1.48	0.004 s	1.353 s	2.9e−3
74 k	110 k	1	1.48	0.044 s	1.958 s	0.022
740 k	1.1 M	1	1.48	0.400 s	13.451 s	0.033
7.4 M	11 M	1	1.48	3.920 s	80.675 s	0.049

* Indicates the number of non-trivial SCCs.

[1] Amdhal's law – https://en.wikipedia.org/wiki/Amdahl%27s_law

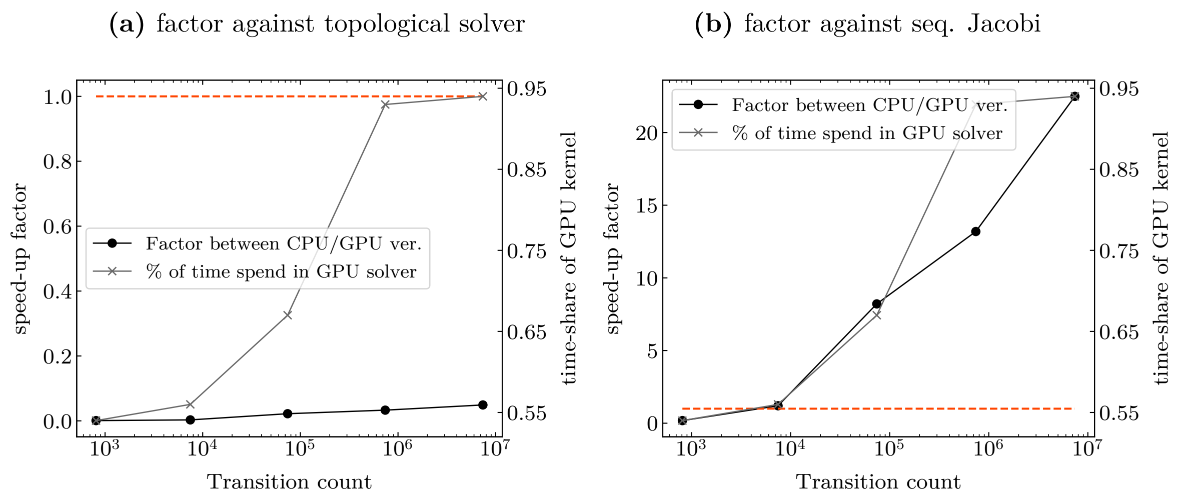


Figure 6.2: Speed-up factor and the time spent within the GPU solver plotted against the transition count of the *maze* model.

***dpm* case study.** The last of this set of benchmarks is the *dpm* model [27]. At the beginning, we would like to point out that during the evaluation of this model, we used the Jacobi method as an underlying solver for the topological approach. The reason is, that the originally used GMRES method was not able to converge for large instances of this model. It is representative of models where a topological solver addresses the problem of slow convergence. The results for the *dpm* protocol are presented in Tables 6.5 and 6.6 and also in Figure 6.3.

We can notice that Jacobi's method, in this case, needed about 90k iterations. The results show that the division into SCC brought its advantages, and thus the topological solver was faster than the Jacobi method. Another important observation is that with increasing model size (which also implies an increase in the number of non-trivial SCCs), the computational time for a topological solver increases much faster than with the Jacobi method (see Figure 6.3). Due to many iterations, we were able to outperform the sequential variant of our implementation as about 90% of the computational time was spent in our solver. At the same time, slow convergence has not prevented our approach from accelerating over a topological solver, and the speedup over this solver is increasing rapidly with an increasing number of non-trivial SCCs.

Table 6.5: Experimental results using the *dpm* models, comparing our implementation against the sequential Jacobi algorithm.

states	transitions	iterations	CPU time	GPU time	factor
127 k	450 k	92 k	2.5 min	9.926 s	14.99
491 k	1.7 M	93 k	10 min	55.561 s	10.82
1.9 M	6.7 M	93 k	40 min	2.2 min	19.00
7.6 M	27 M	94 k	2.7 h	6.5 min	24.80
17 M	62 M	95 k	6 h	12.3 min	29.68

Table 6.6: Experimental results using the *dpm* models, comparing our implementation against the sequential topological algorithm.

states	transitions	SCCs*	$\frac{transitions}{states}$	CPU time	GPU time	factor
127 k	450 k	3.9 k	3.53	5.614 s	9.926 s	0.56
491 k	1.7 M	16 k	3.59	1.4 min	55.561 s	1.51
1.9 M	6.9 M	63 k	3.61	8.1 min	2.2 min	3.78
7.6 M	27 M	252 k	3.62	1.4 h	6.5 min	12.51
17 M	62 M	568 k	3.63	5.9 h	12.3 min	28.57

* Indicates the number of non-trivial SCCs.

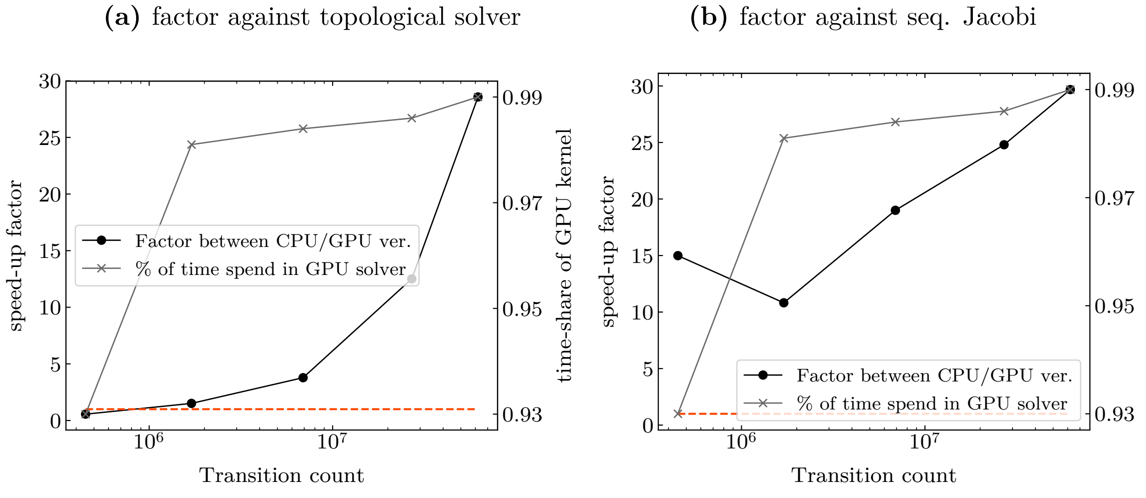


Figure 6.3: Speed-up factor and the time spent within the GPU solver plotted against the transition count of the *dpm* model.

6.2 Model Checking MDPs

6.2.1 Single MDP Model Checking

In this section, we will evaluate the second part of our parallel implementation. Tables and graphs in this section will have the same format as those in the previous one, as the number of SCCs and the number of iterations still have a crucial impact on the performance of compared algorithms. We will again compare our GPU-aided implementation with the organically used VI algorithm and the topological solver, which is used as Storm's default algorithm.

***wlan* case study.** In order to demonstrate performance on models with a small number of non-trivial SCCs, we selected a case study *wlan* by Kwatkowska et al [21]. If we compare the two sequential versions, we see that, as expected, a topological solver is a more preferred variant. Now let's take a look at how our implementation performed against these sequential versions. It is important to note that despite the small number of non-trivial SCCs, we were able to overcome the topological solver. Although the acceleration is not significant, we see that with the growing size of the model and the growing time spent in our solver, the factor gradually increases. The presented acceleration was mainly due to the fact that the solution of trivial SCCs is no longer as trivial as in MC model-checking. Another thing we want to point out with this model is the value of the presented metric. We can see that under the same conditions (SCCs = 1), its value was smaller for MCs. However, this is not unexpected as the structure of MDP is more complex due to non-determinism.

In comparison with the VI algorithm, we can see that with increasing time spent in our solver, the acceleration also increases proportionally. For this particular benchmark, we overcame the time of sequential solution up to 11 times.

Table 6.7: Experimental results using the *wlan* models, comparing our implementation against the sequential topological algorithm.

states	transitions	SCCs*	$\frac{transitions}{states}$	CPU time	GPU time	factor
28 k	57 k	1	2.01	0.046 s	0.749 s	0.061
96 k	205 k	1	2.12	0.231 s	0.748 s	0.31
345 k	762 k	1	2.21	0.911 s	1.357 s	0.671
1.3 M	2.9 M	1	2.26	4.423 s	3.862 s	1.145
5.0 M	11 M	1	2.29	26.745 s	19.738 s	1.355

* Indicates the number of non-trivial SCCs.

Table 6.8: Experimental results using the *wlan* models, comparing our implementation against the sequential value iteration algorithm.

states	transitions	iterations	CPU time	GPU time	factor
28 k	57 k	163	0.092 s	0.749 s	0.123
96 k	205 k	384	0.558 s	0.748 s	0.745
345 k	762 k	696	3.704 s	1.357 s	2.73
1.3 M	2.9 M	1 470	35.819 s	3.862 s	9.28
5.0 M	11 M	2 001	3.6 min	19.738 s	11.08

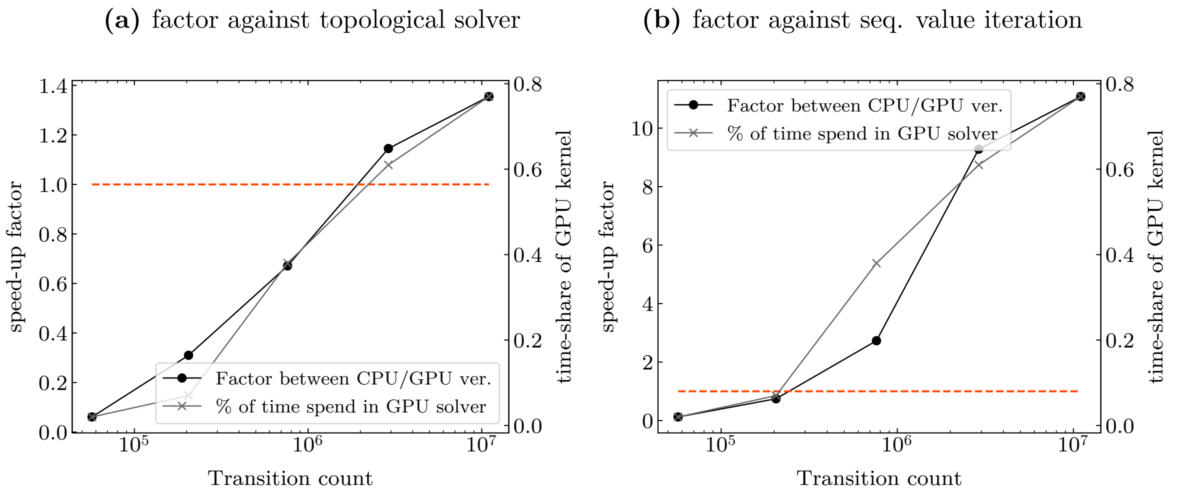


Figure 6.4: Speed-up factor and the time spent within the GPU solver plotted against the transition count of the *wlan* model.

***dpm* case study.** To show a comparison of solvers on models with a large number of non-trivial SCCs, we used a sketch of the *dpm* model and let the PAYNT tool to make a family out of it. We then analysed the quotient MDP of this family.

In this case, it is interesting to observe that both sequential variants had approximately the same performance. Let us compare this with the *dpm* case study in the previous section. We can notice that in MC, the number of iterations was more or less constant, and the number of non-trivial SCCs increased much more significantly with the growing model size. Nevertheless, in the latter case, we see that VI surpasses the topological solver. However, the most important thing is the fact that our solver was able to overcome both variants, and with the increasing time spent on a GPU, the acceleration also increases proportionally.

Table 6.9: Experimental results using the *dpm* models, comparing our implementation against the sequential topological algorithm.

states	transitions	SCCs*	$\frac{transitions}{states}$	CPU time	GPU time	factor
18 k	70 k	9	3.82	0.453 s	0.671 s	0.68
39 k	155 k	19	3.98	1.547 s	0.795 s	1.95
99 k	407 k	67	4.08	9.041 s	1.387 s	6.52
371 k	1.5 M	337	4.17	3.5 min	13.660 s	15.26
5.7 M	24 M	5842	4.24	5.3 h	18.8 min	16.81

* Indicates the number of non-trivial SCCs.

Table 6.10: Experimental results using the *dpm* models, comparing our implementation against the sequential value iteration algorithm.

states	transitions	iterations	CPU time	GPU time	factor
18 k	70 k	2.3 k	0.454 s	0.671 s	0.68
39 k	155 k	3.5 k	1.530 s	0.795 s	1.92
99 k	407 k	7.9 k	8.814 s	1.387 s	6.35
371 k	1.5 M	47 k	3.5 min	13.660 s	15.55
5.7 M	24 M	278 k	5 h	18.8 min	16.21

6.2.2 Multiple MDPs Model Checking

In order to evaluate the model-checking of several MDPs at once, we used a synthesis loop with the difference that only the AR method was used. Subsequently, we selected existing families from the queue, and first analysed each of them separately using a sequential VI algorithm and then all at once by using our GPU implementation. For evaluation, we select two separate benchmarks. For each of them, we present a table containing the number of currently analysed families, the average number of states and transitions of these families, the percentage of profitable work on the GPU[2], average measured time of CPU solver and our GPU-aided implementation, and the factor between the two times. Also, to better

[2]As follows from the explanation of the method, not all the work done is useful and for some families, which, e.g. need fewer iterations to converge, the calculation is unnecessarily prolonged.

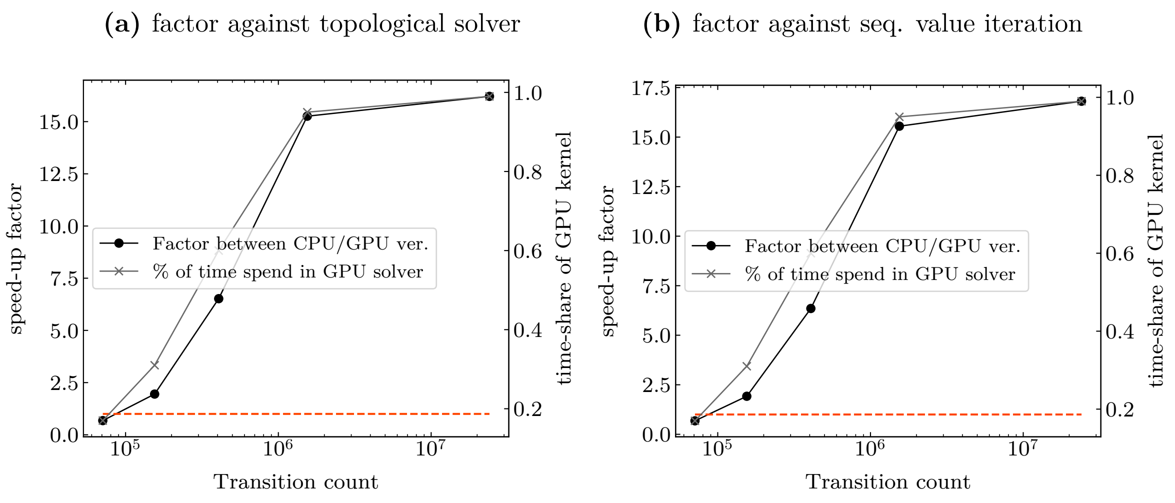


Figure 6.5: Speed-up factor and the time spent within the GPU solver plotted against the transition count of the *dpm* model.

illustrate the differences in the number of iterations and sizes across the groups of families analysed simultaneously, we present box plots showing these differences.

***dpm* case study.** The first model in our set is the *dpm*. From the first row of Table 6.11, we see that the analysis of relatively small models on the GPU is not advantageous, which resulted in the motivation to analyse several smaller models simultaneously. One would expect that the more families we process simultaneously, the higher the speedup will be. However, from the results, we see the exact opposite behaviour. The reason is that with the growing number of families, the amount of profitable work is decreasing. To find the causes, let us take a look at Figure 6.6. From the left part showing the variation of iterations across families, we see that the more families we combine into one group, the more their characteristics are different, and thus the variability of needed iterations in these groups is more significant. Since we still have to consider the highest number of iterations (to get the correct result even for the slowest converging family), the further away from the others, the more unnecessary work will be done. It reflects significantly in the last two groups, where we can already see the so-called outliers[3] in the number of iterations.

Another factor that adversely affects the performance of the evaluated method is the variability of the sizes of individual families. The synthesis process implies that the deeper we are, the smaller the size of families will be. Furthermore, as we showed in Chapter 2, if we analyse each family separately, we can further reduce its state-space by considering only the $S_?$ states (see the first line of Table 6.11). However, since our implementation does not further examine the analysed group of families, it cannot afford to reduce the state-space (as the omitted state for one family can be $S_?$ state for the other) and, therefore, still works with the quotient MDP. The fact that we analyse several families at the same time is thus overshadowed by the above-mentioned disadvantages, which make most of the done work unnecessary.

[3]An **outlier** is an observation that lies an abnormal distance from other values in a random sample from a population.

Table 6.11: Experimental results of multi-level parallelisation using the *dpm* model, comparing our implementation against the sequential VI algorithm.

#families	$\overline{\text{states}}$	$\overline{\text{transitions}}$	profitable	CPU time	GPU time	factor
1*	11730	38197	100 %	0.454 s	0.671 s	0.68
16	10021	30127	22.11 %	14.921 s	4.645 s	3.21
32	9743	28750	17.46 %	29.092 s	16.084 s	1.81
64	9460	27325	12.14 %	57.679 s	80.482 s	0.72
256	6089	16397	1.39 %	2.4 min	11.3 min	0.21
508	5670	14825	1.09 %	4.7 min	1.3 h	0.06

* The family (quotient MDP) size before reduction was 18 k states and 70 k transitions.

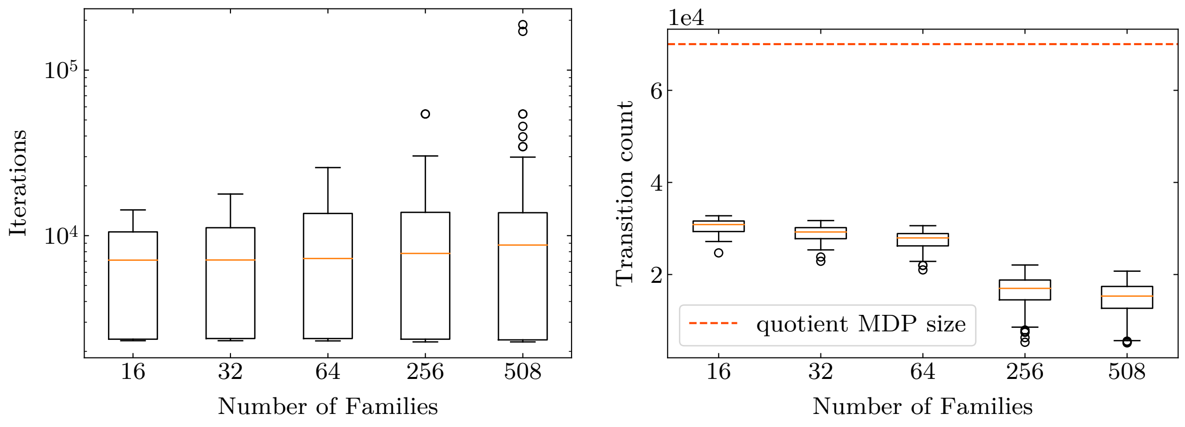


Figure 6.6: Box plots for the *dpm* model, showing the variability of iterations and sizes in groups of families analysed simultaneously.

***maze* case study.** Another model on which we want to demonstrate the multilevel parallelisation is the *maze*. Here we would like to point out that an outlier cannot necessarily be found only within a large group of families. In this case, we can see it already in the first group (see Figure 6.7). This fact brought a slowdown compared to the sequential variant even in the first group. Then the speedup increases for a while, but with the growing size of the group, the amount of profitable work declines due to the reasons given above, and the speedup starts to decrease again.

Table 6.12: Experimental results of multi-level parallelisation using the *maze* model, comparing our implementation against the sequential VI algorithm.

#families	$\overline{\text{states}}$	$\overline{\text{transitions}}$	profitable	CPU time	GPU time	factor
1*	147	333	100 %	0.818 ms	0.543 s	1.5e−3
64	140	274	17.90 %	1.187 s	3.919 s	0.30
164	139	268	23.36 %	4.246 s	4.389 s	0.97
328	136	247	22.63 %	8.261 s	5.619 s	1.47
656	135	244	18.32 %	16.852 s	9.648 s	1.75
1264	135	243	13.66 %	32.217 s	40.307 s	0.80

* The family (quotient MDP) size before reduction was 169 states and 377 transitions.

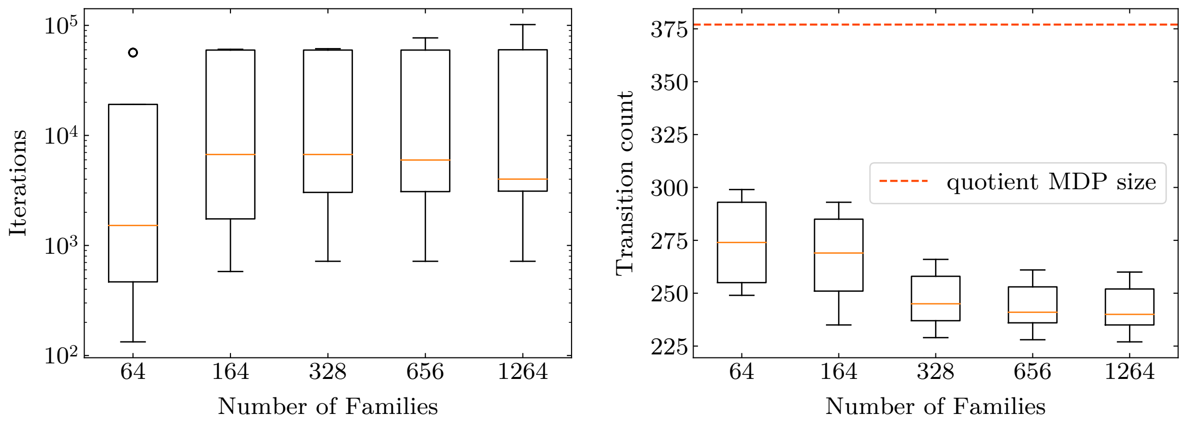


Figure 6.7: Box plots for the *maze* model, showing the variability of iterations and sizes in groups of families analysed simultaneously.

So to summarize, in cases where the number of iterations within the group did not differ much, we were able to speed up the calculation up to three times. However, there is still the problem of over-sizing associated with using the quotient MDP of a super-family (i.e. describing all input realisations). Due to the above, the current version does not apply to the synthesis process as it does not bring the expected benefits.

6.3 Synthesis of Probabilistic Programs

Finally, let us look at how the acceleration of the model-checking affects the overall synthesis process. In the previous sections, we have shown that our model-checking is not a general solution, i.e. that in some cases, a sequential topological solver is more advantageous. For this reason, we will examine the influence of our methods on synthesis only in cases where we can accelerate model-checking. Obviously, in cases where the model-checking cannot be accelerated, the total synthesis time would not change (or would be worse) compared to the sequential version.

To that end, we selected a case study *dpm*, as we showed that our methods could overcome all the presented sequential variants in the case of this model. From this model, we have created four groups of families in which the size of their members gradually increases. A more detailed description of these families is shown in Table 6.13. For each family, this table shows the average size of its members (MCs) as well as the average size of the analysed subfamilies (MDPs) in the AR phase of the synthesis process. We then also present the average factor between sequential and parallel model-checking performed during the synthesis for each type of model. It is important to note that as the size of models describing family members or the family itself increases, the number of family members decreases. The reason for this is quite simple. In order to be able to get the most out of our parallel model-checking, we want to work with the largest possible models (as this is where the benefit of parallelisation shows the most). However, with the increasing size of individual members, the size of respective MDP describing the family is also rapidly increasing. Because of this, we had to keep the number of members to a minimum for large models; otherwise, we would run out of memory when building the MDP. It is partly due to the building of support structures for efficient work with the family, but we assume a potential error in implementation, which has not yet been detected. To give a better idea, building the MDP, which describes the family in test case 4, took up to 16 GB of RAM. While using the same sketch and family size 81, it was up to 25 GB, and this trend gradually continued. However, the above is not an obstacle for us, as the problem of growing family size is addressed by the synthesis method itself, and we focus on the problem of large family members. During the evaluation of the synthesis, we observed that the speed-ups of our methods were more or less the same across the iterations. Thus with the increasing number of iterations (resulting from bigger families), the amount of work would increase proportionally, but the benefit of our implementation should remain the same. We can also see from the table that the first two families contained relatively small models. Therefore in these cases, we could not accelerate MC model-checking. However, since these families contained enough members, the abstraction of these families was already large enough to accelerate MDP model-checking up to 14 times.

Table 6.13: Characteristics of families created from the *dpm* model used to evaluate our methods in conjunction with the synthesis loop. The results of the synthesis for these families can be found in Table 6.14.

		MCs			MDPs		
test case	family size	average states	average transitions	average factor	average states	average transitions	average factor
1	729	13 k	19 k	0.04	28 k	114 k	4.56
2	729	114 k	319 k	0.37	205 k	829 k	14.78
3	81	1.6 M	5.7 M	1.81	3.0 M	9.6 M	14.11
4	9	4.1 M	14.6 M	4.99	5.5 M	23.2 M	12.39

We will now present the results for individual families from our benchmark, which we can see in Table 6.14. The first part provides information about synthesis using sequential model-checking (CPU), and the second part shows information about synthesis using our methods (GPU). For each variant, we show the number of performed CEGIS and AR iterations and the time of the synthesis itself. The last, most crucial part of this table is the resulting acceleration of the whole synthesis process. In addition to this information, we also present the maximum possible speedup resulting from *Amdahl's law.* It is derived from the amount of parallelizable part of an algorithm; thus, in our case, from the total

Table 6.14: Experimental results using the families from Table 6.13, comparing the synthesis using sequential model-checking with the synthesis using our parallel model-checking. In the experiments, we looked for the optimal solution to the synthesis problem, and specifically, it was a minimal synthesis problem.

	CPU			GPU				
test case	CEGIS	AR	time	CEGIS	AR	time	factor	max factor
1	211	3	9.4 min	43	11	11.2 min	**0.84**	4,57
2	379	5	3.1 h	274	65	3.2 h	**0.95**	3,05
3	54	3	20.2 h	49	21	11.4 h	**1.78**	2,45
4	5	1	9.4 h	11	7	7.4 h	**1.28**	3,92

time taken by model-checking (e.g., if the algorithm contains 75% of the work that can be parallelized, regardless of the number of threads, the acceleration will never be more than $\frac{1}{1-0.75} = 4$).

The results show that we could not speed up the overall synthesis process in cases where we were not able to speed up the MC model-checking (test case 1 and 2). In the other two families, we see that if we accelerated all the model-checkings, we managed to speed up the synthesis as well. However, let us compare the ratio of CEGIS and AR iterations of individual approaches. We see that the parallel variant did more work than sequential one and still managed to be faster. The reason is the already mentioned non-deterministic oracle, used by the hybrid method. It uses different metrics to switch between the CEGIS and the AR loops, one of which is the time spent in each of them. Its prolonged duration and the fact that it did not significantly contribute to the pruning of the state-space of families caused the oracle to prefer the CEGIS loop, which proved to be the better of the two variants for the verified model. Up to 14-fold acceleration of MDP model-checking caused a more balanced number of CEGIS and AR iterations, and thus CEGIS was given less time than with the sequential variant. Paradoxically, the acceleration brought more or less unnecessary work into the calculation.

Therefore, in order to be able to compare both approaches under the same conditions, we adapted the oracle to prefer the CEGIS loop more. By doing so, we were able to get the same amount of work for both variants. Under these conditions, we tested the synthesis on families where we could accelerate all model-checkings (test cases 3 and 4), and the results can be seen in Table 6.15. We see that in this case, we are already significantly closer to the theoretical limit of acceleration and thus that our parallelisation has successfully contributed to the process of synthesis on huge models.

Table 6.15: Synthesis results with adjusted work distribution, compared to the results in Table 6.14.

test case	CEGIS	AR	CPU time	GPU time	factor	max factor
3	25	1	8.4 h	4.0 h	**2.07**	2,45
4	5	1	9.4 h	2.5 h	**3.80**	3,92

The last thing we would like to demonstrate when evaluating the synthesis is to use a combined approach to model-checking. We find this in situations where family members are small, but many of them will cause the MDP describing this family to be large. We see exactly this situation in test cases 1 and 2, and in such a situation, we would like to use a parallel version of MDP model-checking. However, on MC model-checking, we would like to use a more advantageous topological solver. It is in this situation that our metrics presented in the previous sections will come in handy. Based on experiments over the benchmark presented in this work as well as employing several other models from the *Quantitative Verification Benchmark Set*[4], we determined the threshold values, indicating whether in a given situation it is more appropriate to use a sequential or parallel version of model-checking. The results of our observations are shown in Figure 6.8. Specifically, we show how the value of our metric affects the resulting acceleration compared to the topological solvers of both variants of model-checking. The topological approach was chosen because, under certain conditions, it can bring significant benefits, as shown in the previ-

[4]Quantitative Verification Benchmark Set - https://qcomp.org/benchmarks/

ous sections. It is nice to see from the figures that both with MC model-checking and with MDP model-checking, there is a limit value above which our approach is still faster than the topological one. Measurement inaccuracies can cause slight fluctuations, but as we want to be conservative, for both approaches, we have chosen a value above which our approach is always faster. As expected, the threshold value for MC model-checking is lower, and we specifically chose 3.6. For MDP model-checking, we have set the value to 4. The higher value is caused by the fact that MDP has more transitions related to non-determinism in its states. Before performing the model-checking, we then compare the current value with the corresponding threshold and choose the „right“ approach accordingly.

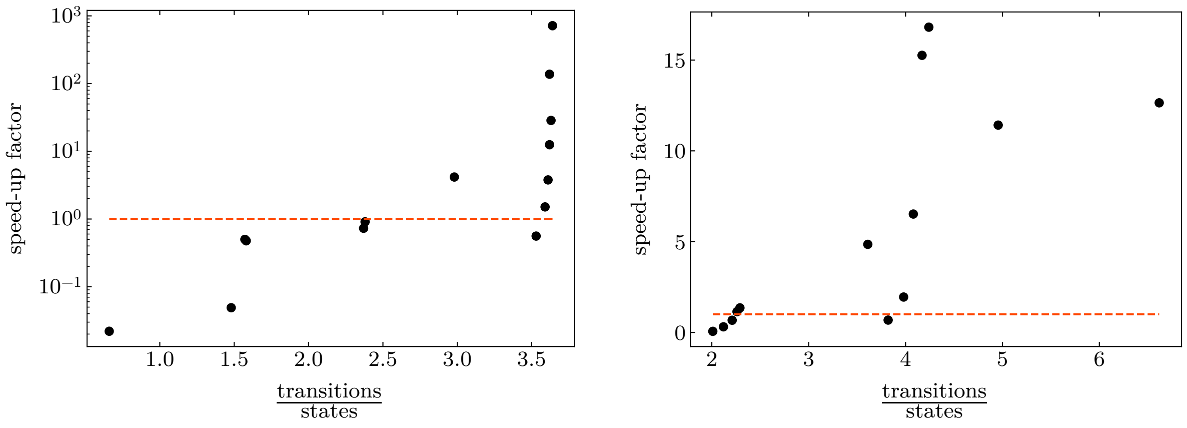


Figure 6.8: Comparing the $\frac{transitions}{states}$ metric against the achieved acceleration with respect to STORM's topological solver, for both MC (left) and MDP (right) model-checking.

After applying the metric, we tried to run test case 2 from our benchmark again. Based on our metric, a more advantageous topological solver was used for MC model-checking, and since parallel MDP model-checking was up to 14 times faster than the sequential variant, our metric correctly preferred this approach. With this combined approach, we were able to speed up the original, purely sequential approach by one hour, representing a factor of 1.47 out of a possible 1.75 (since MDP model-checking took up 43 % of the total calculation).

Chapter 7

Conclusion

In this thesis, we examined the problem of synthesis of probabilistic programs. After identifying the key shortcomings: the explosion of the underlying state-space and subsequent verification of these huge models, we solved this problem by parallelizing the original implementation of model-checking methods.

Specifically, we have speeded up the existing implementations of MC and MDP model-checking. With a sufficiently large number of iterations, we were able to accelerate sequential iteration methods by a factor of up to 30 in the Jacobi method and by a factor of more than 16 in the case of the VI method. In cases when the number of iterations is small, we have found that the preprocessing steps and other parts of the calculation performed by Storm do not scale as well as our GPU implementation. Furthermore, in cases when model contains a large number of non-trivial SCCs, we were able to significantly overcome the Storm's default topological approach for both MC and MDP model-checking. In terms of speed-up, we have accelerated MDP model-checking up to 16 times, and in MC model-checking, we reached a maximum value of 716-fold acceleration. On the other hand, if the number of non-trivial SCCs is too small, our implementation got the shorter end of the stick. A common finding in both comparisons is that when verifying small models, the overhead associated with execution on GPU exceeds the effective computation time. Therefore, we infer that while obtained gains are promising, the use of our GPU-aided implementation is no universal approach for performance problems but rather a promising addition to the available model-checking tools, each tailored for particular fields of application. Fortunately, we have found a possible solution to identify these situations, and the metric that was evaluated in Figure 6.8. The last thing we have done was to compare the possible peak performance of the GPU device with the achieved performance results, and we fall short of expectations. The main reason for this gap is the lack of hardware concerning the usage of (required) double precision. Current GPU devices suffer a significant drop in performance while working with doubles instead of floats, as they contain fewer double precision units (see Ryoo et al. [30]). Specifically, in the case of our GPU, the double precision instruction throughput is $^1/_{32}$ the rate of single precision one.

By involving all parallel methods, we have reached in some cases, almost theoretical limit of accelerating the overall synthesis process, but only in the case of working with sufficiently large models. Specifically, it was a factor of up to 3.8 if model-checking took $\approx$74% of the total synthesis time and thus, according to Amdahl's law, it was possible to achieve a maximum speed-up factor of 3.92. In order to use GPUs even on small models, we have proposed family-based parallelisation. The aim was to analyse several smaller families simultaneously, but the different characteristics of these families sentenced this approach to failure. The more subfamilies we tried to analyse simultaneously, the less benefit our method brought, which directly conflicted with the original idea. The maximum speed-up we were able to achieve was a factor of 3, compared to a sequential CPU implementation. However, since it was not a permanent trend but only one case on one specific benchmark, in the end, we did not include our method in the synthesis process implementation.

Future work Despite the rather favourable results of parallel model-checking, there is still space for an improvement in this area. One possibility is the parallelisation of the graph algorithms used to calculate the sets S_0 and S_1 as the time spent in this calculation increases with the increasing size of the model. In addition, if the number of required iterations is small, this calculation makes up a significant part of the total time, which worsens the overall acceleration. Another possibility is the parallelisation of the topological approach, but based on the reasons explained in this thesis, it would rather make sense to consider CPU parallelisation.

Acceleration of the topological approach is also directly related to the second-largest bottleneck of the synthesis process, which is the construction of counterexamples (for more details, we refer to [3]). One of the steps in this process is also MC model-checking. The topological approach has shown to be the most suitable for this model-checking and therefore we would be able to directly influence the speed of creating counterexamples by accelerating it.

Another thing worth trying is to return to the proposed family-based parallelisation. The results show that this approach could become applicable by reducing the amount of unnecessary work. Intuitively, if we look at Figures 6.6 and 6.7, we want to move the red line as close as possible to the maximum size of the model in a given group. This can be achieved by modifying the splitting algorithm; as with the current algorithm, we cannot be sure which parameter will be splitted. Therefore, we cannot omit any row from the matrix of the original quotient MDP of a super-family without further analysis. Such splitting could potentially reduce differences in the number of iterations, as the resulting groups of subfamilies could share more of their state-space.

Last but not least, we want to examine the problem revealed at a later stage of our work. Specifically, it is excessive memory consumption when compiling quotient MDPs describing the whole family.

Bibliography

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M. et al. Syntax-guided synthesis. In: *2013 Formal Methods in Computer-Aided Design*. 2013, p. 1–8. DOI: 10.1109/FMCAD.2013.6679385.

[2] Andriushchenko, R. *Computer-Aided Synthesis of Probabilistic Models*. Brno, CZ, 2020. Master's thesis. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/22997/.

[3] Andriushchenko, R., Ceska, M., Junges, S. and Katoen, J. Inductive Synthesis for Probabilistic Programs Reaches New Horizons. *CoRR*. 2021, abs/2101.12683. Available at: https://arxiv.org/abs/2101.12683.

[4] Andriushchenko, R. and Stupinsky, S. *PAYNT*. GitHub, February 2021. Available at: https://github.com/gargantophob/synthesis/commit/af79ef08690008c5c9e783e4cc6e717ac00406d2.

[5] Bellman, R. A Markovian decision process. *Journal of Mathematics and Mechanics*. 1957, vol. 6, no. 5, p. 679–684. Available at: http://www.jstor.org/stable/24900506.

[6] Bosnacki, D., Edelkamp, S., Sulewski, D. and Wijs, A. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Technology Transfer*. january 2011, vol. 13, p. 21–35. DOI: 10.1007/s10009-010-0176-4.

[7] Ceska, M., Hensel, C., Junges, S. and Katoen, J. Counterexample-Driven Synthesis for Probabilistic Program Sketches. *CoRR*. 2019, abs/1904.12371. Available at: http://arxiv.org/abs/1904.12371.

[8] Ceska, M., Jansen, N., Junges, S. and Katoen, J. Shepherding Hordes of Markov Chains. *CoRR*. 2019, abs/1902.05727. Available at: http://arxiv.org/abs/1902.05727.

[9] Chrszon, P., Dubslaff, C., Klüppelholz, S. and Baier, C. ProFeat: feature-oriented engineering for family-based probabilistic model checking. *Formal Aspects of Computing*. august 2017, vol. 30. DOI: 10.1007/s00165-017-0432-4.

[10] Classen, A., Cordy, M., Heymans, P., Legay, A. and Schobbens, P. Y. Model Checking for Software Product Lines with SNIP. *International Journal on Software Tools for Technology Transfer*. october 2012, vol. 14. DOI: 10.1007/s10009-012-0234-1.

[11] Duflot, M., Kwiatkowska, M., Norman, G. and Parker, D. A Formal Analysis of Bluetooth Device Discovery. In: *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*. 2004.

[12] Forejt, V., Kwiatkowska, M., Norman, G. and Parker, D. Automated Verification Techniques for Probabilistic Systems. In:. January 2011, p. 53–113. DOI: 10.1007/978-3-642-21455-3. ISBN 978-3-642-21454-7.

[13] Hartmanns, A., Klauck, M., Parker, D., Quatmann, T. and Ruijters, E. The Quantitative Verification Benchmark Set. In: Vojnar, T. and Zhang, L., ed. *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*. Springer, 2019, vol. 11427, p. 344–350. Lecture Notes in Computer Science. DOI: 10.1007/978-3-030-17462-0_20. Available at: https://doi.org/10.1007/978-3-030-17462-0_20.

[14] Hensel, C., Junges, S., Katoen, J.-P., Quatmann, T. and Volk, M. The Probabilistic Model Checker Storm. *ArXiv e-prints*. february 2020, p. arXiv:2002.07080.

[15] Hoberock, J. and Bell, N. *Thrust: A Parallel Template Library*. 2010. Http://code.google.com/p/thrust/. Available at: http://code.google.com/p/thrust/.

[16] Katoen, J.-P. Model Checking Meets Probability: A Gentle Introduction. In:. June 2013. DOI: 10.3233/978-1-61499-207-3-177.

[17] Klapka, O. and Slaby, A. NVidia CUDA Platform in Graph Visualization. In: Kunifuji, S., Papadopoulos, G. A., Skulimowski, A. M. and Kacprzyk, J., ed. *Knowledge, Information and Creativity Support Systems*. Cham: Springer International Publishing, 2016, p. 511–520.

[18] Knuth, D. and Yao, A. Algorithms and Complexity: New Directions and Recent Results. In:. Academic Press, 1976, chap. The complexity of nonuniform random number generation.

[19] Kwiatkowska, M., Norman, G. and Parker, D. Stochastic Model Checking. In:. May 2007, 4486 of LNCS, p. 220–270. DOI: 10.1007/978-3-540-72522-0-6. ISBN 9783540724827.

[20] Kwiatkowska, M., Norman, G. and Parker, D. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G. and Qadeer, S., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 585–591. ISBN 978-3-642-22110-1.

[21] Kwiatkowska, M., Norman, G. and Sproston, J. Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. november 2003, vol. 2399. DOI: 10.1007/3-540-45605-8-11.

[22] Lee, D. *PL00, Parallel computing*. Accessed: 2021-04-29. Available at: https://userdyk-github.github.io/pl00/PL00-Parallel-computing.html.

[23] Littman, M. L., Cassandra, A. R. and Kaelbling, L. P. *Learning policies for partially observable environments: Scaling up.* 1995.

[24] Merrill, D. *CUDA UnBound (CUB) Library.* 2011. Available at: https://nvlabs.github.io/cub/.

[25] Munshi, A., Gaster, B., Mattson, T. G., Fung, J. and Ginsburg, D. *OpenCL Programming Guide.* 1stth ed. Addison-Wesley Professional, 2011. ISBN 0321749642.

[26] Norman, G., Parker, D., Kwiatkowska, M. and Shukla, S. Evaluating the Reliability of NAND Multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* 2005, vol. 24, no. 10, p. 1629–1637.

[27] Norman, G., Parker, D., Kwiatkowska, M., Shukla, S. and Gupta, R. Formal Analysis and Validation of Continuous Time Markov Chain Based System Level Power Management Strategies. In: Rosenstiel, W., ed. *Proc. 7th Annual IEEE International Workshop on High Level Design Validation and Test (HLDVT'02).* IEEE Computer Society Press, 2002, p. 45–50.

[28] NVIDIA, Vingelmann, P. and Fitzek, F. H. *CUDA, release: 11.3.* 2020. Available at: https://developer.nvidia.com/cuda-toolkit.

[29] Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* 1stth ed. USA: John Wiley & Sons, Inc., 1994. ISBN 0471619779.

[30] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B. et al. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* New York, NY, USA: Association for Computing Machinery, 2008, p. 73–82. PPoPP '08. DOI: 10.1145/1345206.1345220. ISBN 9781595937957. Available at: https://doi.org/10.1145/1345206.1345220.

[31] Sapio, A., Bhattacharyya, S. S. and Wolf, M. Efficient Model Solving for Markov Decision Processes. In: *2020 IEEE Symposium on Computers and Communications (ISCC).* 2020, p. 1–5. DOI: 10.1109/ISCC50000.2020.9219668.

[32] Shmatikov, V. Probabilistic Analysis of Anonymity. In: *Proc. 15th IEEE Computer Security Foundations Workshop (CSFW'02).* IEEE Computer Society Press, 2002, p. 119–128.

[33] Solar Lezama, A. *Program Synthesis by Sketching.* USA, 2008. Dissertation. ISBN 9781109097450. AAI3353225.

[34] Stewart, W. *Introduction to the numerical solution of Markov chains.* Princeton, NJ: Princeton Univ. Press, 1994. ISBN 0691036993. Available at: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+152880593&sourceid=fbw_bibsonomy.

[35] Češka, M., Dannenberg, F., Paoletti, N., Kwiatkowska, M. and Brim, L. Precise Parameter Synthesis for Stochastic Biochemical Systems. *Acta Inf.* Berlin,

Heidelberg: Springer-Verlag. september 2017, vol. 54, no. 6, p. 589–623. DOI: 10.1007/s00236-016-0265-2. ISSN 0001-5903. Available at: https://doi.org/10.1007/s00236-016-0265-2.

[36] Ábrahám, E., Jansen, N., Wimmer, R., Katoen, J.-P. and Becker, B. DTMC model checking by SCC reduction. In:. September 2010. DOI: 10.1109/QEST.2010.13.

Appendix A

Storage Medium

`/synthesis/*` — source code of Paynt (containing our parallel methods) from date May 25, 2021

`/README.txt` — useful information about the storage medium content

`/text/*` — source code of this thesis

`/xmarci10.pdf` — final version of this thesis