RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair for Software Modeling and Verification

BACHELOR OF SCIENCE THESIS

PARAMETER SYNTHESIS FOR CONTINUOUS-TIME MARKOV CHAINS

Fynn Mazurkiewicz

Examiners:
Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen
Prof. Dr. Erika Ábrahám

Additional Advisor:
Matthias Volk

Aachen, 02.01.2020

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Fynn Mazurkiewicz
Aachen, den 2. Januar 2020

Abstract

Continuous-Time Markov Chains (CTMCs) are an important model for any real world system involving stochastic behaviour. They are used for various kinds of safety-related systems. Each state of the system is described and transitions between the states are modelled by their transition rate. Many times, these transition rates are not known exactly. Other times, we might want to explore how the model behaves with different transition rates. In both cases, transition rates can be replaced by parameters from a continuous interval. However, it is significantly more computation-intensive to assert properties on these parametric CTMCs (pCTMCs) due to their dense parameter intervals.
We present an approach first described by Brim et al. to perform the transient analysis on parametric CTMCs. By approximating the pCTMC behaviour an upper and lower bound of the probability distribution vector can efficiently be calculated. The parameter space of the model is iteratively used for bounds calculation and decomposed to reduce the error in these approximations. We also implemented the approximation and decomposition algorithm including various optimizations in the model checker Storm.

Contents

1 Introduction

A Continuous-Time Markov Chain (CTMC) is a stochastic model that allows modelling real-world systems and asserting properties on that model. CTMCs are used in many domains such as aerospace engineering [11] or automotive engineering [12]. CTMCs describe states of a system and the rate at which the system might transition from one state to another.

A major issue when dealing with CTMCs is that they require exact transition rates. Often, the rates are not known exactly. A parametric CTMC (pCTMC) solves this problem by allowing parameters in the model definition over a real-valued interval. This makes the exact transient analysis of pCTMCs impossible since there are infinitely many CTMCs induced by one pCTMC.

This thesis builds on results by Brim et al. [4] and Ceska et al. [13] and outlines a technique for using min-max approximation to compute transient probabilities in the parametric case. The core concept of this technique is to use the parameter interval bounds to calculate the maximum or minimum state probabilities of the model during each iteration of the transient analysis. By iteratively refining the parameter regions we can reduce the introduced inaccuracy to any given error bound.

Additionally, an implementation of this algorithm along with optimizations is provided within the Storm model checker [7].

This thesis is divided into five chapters. Chapter 2 gives a brief overview over CTMCs, pCTMCs and their transient analysis. The min-max approximation is described in detail in Chapter 3. We also present the parameter space decomposition for precision refinement. Chapter 4 looks at the implementation of the min-max algorithm within the Storm model checker and outlines various changes to improve the performance. Moreover, we compare our results to the original implementation in the PRISM model checker on various models. Chapter 5 summarizes all results in a stand-alone readable fashion.

2 Preliminaries

2.1 Continuous-Time Markov Chains

Analysing a real-world system is often done in two steps. Firstly, one translates that system into a well-defined mathematical model. Secondly, one queries this model and asserts different properties of interest.

A Markov chain is used to model such real-world systems. We denote the transitions in this model by an exit-rate function which maps each state to the rate of leaving it as well as a transition probability matrix, which determines the probability of going to a successor state. The probability of reaching any following state is only dependent on the current state. This is often coined *memorylessness*.

An interesting subset of Markov chains are *Continuous-Time Markov Chains* (CTMCs), which are used to model and analyse models with finite state space and their evolution over time. In these models, time is dense and transitions between two states may occur at any given moment. Since the state residence time is random, a negative expontential distribution is used for modelling this behaviour. All definitions are based on [1].

Definition 2.1 (Continuous-Time Markov Chain) *A CTMC C is given as follows:*

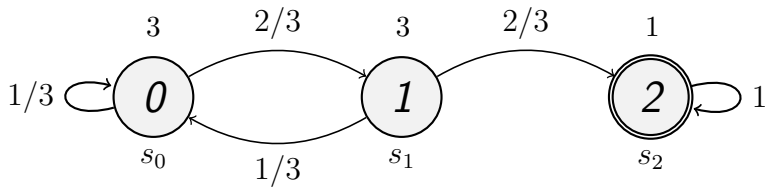
$$C = (S, P, r, init, AP, L),$$

where S is a finite set of states, $P : S \times S \mapsto [0, 1]$ is the transition probability matrix, $r : S \mapsto \mathbb{R}_{\geq 0}$ is the exit-rate function of any state, $init : S \mapsto [0, 1]$ the initial distribution, AP a set of atomic propositions and $L : AP \mapsto 2^S$ is a function that assigns any of the atomic propositions to all states that it holds in. For P to be a probability matrix, the sum of all outgoing mass always has to be one:

$$\sum_{s' \in S} P(s, s') = 1 \forall s \in S$$

To follow along, an example CTMC C^e is given.

Example 2.1 *In this example, $S_{C^e} = \{0, 1, 2\}$, $P_{C^e} = \begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix}$ and $r_{C^e} = \begin{pmatrix} 3 & 3 & 1 \end{pmatrix}$. The initial distribution can be represented in the vector $init = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$. The CTMC can be visualized as follows.*



We mark any state i with the label s_i to indicate, that we are interested in this state. Note that s_2 is absorbing.

We are often interested in the relations between states. We fix all predecessors of a given state s in the predecessor set.

Definition 2.2 (Predecessor set)

$$pred(s) = \{s' \mid \forall s' \in S : P(s', s) > 0\}$$

Likewise, we define the successor set of a given state s.

Definition 2.3 (Successor set)

$$succ(s) = \{s'' \mid \forall s'' \in S : P(s, s'') > 0\}$$

In order to calculate the *transition rate* between two states, we multiply the exit-rate of the current state with the transition probability between the current and any given successor state. This converts a regular CTMC into a Rate CTMC.

Definition 2.4 (Rate CTMC) *The transition rate matrix $R : S \times S \mapsto \mathbb{R}_{\geq 0}$ describes all transition rates from one state s to another one s' and is defined as follows.*

$$R(s, s') = r(s) * P(s, s'),$$

where $r : S \mapsto \mathbb{R}_{\geq 0}$ is the exit-rate function of any state and $P : S \times S \mapsto [0, 1]$ the transition probability matrix.

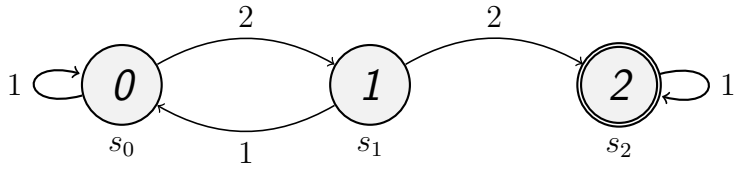
The transition rate matrix replaces P and r, so that we can easily redefine a CTMC by its transition rates:

$$C = (S, R, init, AP, L),$$

where S, $init$, AP, L are as before and $R : S \times S \mapsto \mathbb{R}_{\geq 0}$ is the transition rate matrix.

Example 2.2 *The transition rate matrix of Example 2.1 is given by* $R_{C_e}(s, s') = \begin{pmatrix} 1 & 2 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$.

It can be visualized as follows.



2.2 Uniformization

Dealing with CTMCs can become problematic since the calculation becomes increasingly unstable with particularly high or low transition rates. In order to circumvent this problem, any CTMC is first transformed by scaling all transition rates to the interval $[0, 1]$. This process is called *Uniformization*. First, one determines the maximum exit rate of the CTMC.

Definition 2.5 (Maximum exit rate)

$$r_{max} = max\{r(s) \mid s \in S\}$$

Example 2.3 *For Example 2.1 the maximum exit rate is* $r_{max} = 3$.

The maximum exit rate is used to build the *uniformised transition rate matrix* by dividing all transition rates by a constant $q_{max} \geq r_{max}$ that is larger than the maximum exit rate.

Definition 2.6 (Uniformisation matrix) *The uniformisation matrix* $Q^{unif(C)} : S \times S \mapsto [0, 1]$ *is defined for all* $s, s' \in S$ *as:*

$$Q^{unif(C)}(s, s') = \begin{cases} \dfrac{R(s, s')}{q_{max}} & \text{if } s \neq s' \\ 1 - \displaystyle\sum_{s'' \neq s} \dfrac{R(s, s'')}{q_{max}} & \text{otherwise} \end{cases},$$

where $q_{max} \geq r_{max}$ *is the uniformisation factor.*

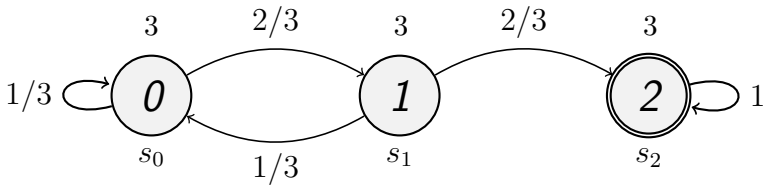
This way, all transition rates remain small and the transient computations stay stable. The uniformization matrix induces the uniformized CTMC, where all exit-rates are set to the maximum exit rate.

Definition 2.7 (Uniformized CTMC)

$$C = (S, Q^{unif(C)}, r^{unif}, init, AP, L),$$

where S, *init*, AP, L *are as before,* $r^{unif}(s) = q_{max} \forall s \in S$ *and* $Q^{unif(C)}$ *is the uniformization matrix.*

Example 2.4 *Given* $q_{max} = 3$, *it holds that* $Q^{unif(C^e)} = \begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix}$. *The uniformized CTMC can be visualised as follows:*



2.3 Time-bounded Continuous Stochastic Logic

To formalize the transient analysis of CTMCs, we introduce the time bounded until operator of *Continuous Stochastic Logic* (CSL) first described by Aziz et al. [1]. CSL is used to query whether a model satisfies certain properties and equips the *CTL until-operator* with a time interval. The foundations of *CTL* are outlined in [3] and [6].

Definition 2.8 (CTL until-operator $\mathcal{U}$) *$\Phi \,\mathcal{U}\, \Psi$ asserts that a Ψ state can be reached by only visiting Φ-states before. Φ and Ψ are logic formulas over state labels, that describe states in which the formula holds.*

Example 2.5 *$(s0 \vee s1)\,\mathcal{U}\,(s2)$ holds for Example 2.1 since we will always be in state 0 or 1 until reaching state 2.*

Definition 2.9 (Time-bounded CSL until-operator $\mathcal{U}^I$) *$\Phi\ \mathcal{U}^I\ \Psi$ asserts that a Ψ state can be reached only via Φ-states within a time interval $I \subseteq [0, \infty)$.*

In the following, we will restrict the interval I to either point intervals $I = [t, t], t \in \mathbb{R}$ or upper bound intervals $I = [0, t], t \in \mathbb{R}$.

Definition 2.10 (CSL until-operator $\mathcal{U}^I$) *$\Phi\ \mathcal{U}^I\ \Psi$ asserts that a Ψ state can be reached only via Φ-states within a time interval $I \subseteq [0, \infty)$.*

In the following, we will restrict the interval I to either point intervals $I = [t, t], t \in \mathbb{R}$ or upper bound intervals $I = [0, t], t \in \mathbb{R}$.

Definition 2.11 (CSL probabilistic operator $\mathbb{P}$) *$\mathbb{P}_{\leq\lambda}\Phi$ determines, whether the CSL path formula Φ satisfies the given probability bound λ.*

2.4 Transient Analysis of CTMCs

When analysing any system, it is often interesting to evaluate how this model behaves over time. CSL is used to describe interesting properties and transient analysis can be used to solve time-bounded CSL queries. Transient analysis was first described by Baier et al. [2].

Definition 2.12 (Transient probability vector) *The transient probability vector*

$$\pi^{C,t} := (\pi_{s_1}(t), ..., \pi_{s_n}(t))$$

fixes the probability of being in any of the states $s_1, ..., s_n \in S$ after time t in a CTMC C. It is calculated as a sum of expressions giving the state distributions after i discrete steps weighted by the ith Poission probability.
We introduce a short-hand for accessing the probability of a single state.

$$\pi^{C,s,t} := \pi^{C,t}(s)$$

The initial distribution vector is denoted with $\pi^{C_p,0} := [init(s_0), ..., init(s_n)]$. *The probability vector for* $t > 0$ *is defined as:*

$$\pi^{C,t} = \underbrace{\sum_{i=0}^{\infty} e^{-q_{max}t} \frac{(q_{max}t)^i}{i!}}_{\text{Taylor-Maclaurin \& Poisson Prob.}} \cdot \underbrace{\pi^{C_p,0}}_{\text{inital distribution}} \cdot \underbrace{(Q^{unif(C)})^i}_{\text{Uniformization Matrix}}$$

In practice we will truncate the infinite sum at k_ϵ. This introduces an error which can be calculated. Given an acceptable error ϵ, we choose k_ϵ minimal, such that

$$1 - \sum_{i=0}^{k_\epsilon} e^{-qt} \frac{(qt)^i}{i!} \leq \epsilon$$

Example 2.6 *Given the example CTMC* C^e, $t = 2$, $q_{max} = 3$ *and the initial distribution* $\pi^{C_p,0} = (1, 0, 0)$ *the transient probabilities are calculated.*

$$\begin{aligned}
\pi^{C_e,3} &= \sum_{i=0}^{\infty} e^{(-3\cdot 2)} \frac{(3\cdot 2)^i}{i!} \cdot (1,0,0) \cdot (Q^{unif(C^e)})^i \\
&= \sum_{i=0}^{\infty} e^{(-3\cdot 2)} \frac{(3\cdot 2)^i}{i!} \cdot (1,0,0) \cdot \begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix}^i \\
&= \frac{1}{e^6} \cdot (1,0,0) \\
&\quad + \sum_{i=1}^{\infty} e^{(-3\cdot 2)} \frac{(3\cdot 2)^i}{i!} \cdot (1,0,0) \cdot \begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix}^i \\
&= \frac{1}{e^6} \cdot (1,0,0) \\
&\quad + \frac{6}{e^6} \cdot (1,0,0) \cdot (\begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix})^1 \\
&\quad + \sum_{i=2}^{\infty} e^{(-3\cdot 2)} \frac{(3\cdot 2)^i}{i!} \cdot (1,0,0) \cdot \begin{pmatrix} 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \\ 0 & 0 & 1 \end{pmatrix}^i \\
&= ... \\
&= \{0.090335, 0.089999, 0.819664\}
\end{aligned}$$

One can very clearly see, that the absorbing state $s = 2$ *holds most of the mass at* $t = 2$.

2.5 Parametric Continuous-Time Markov Chains

When modelling a real-world system, it is often hard to find precise transition rates. Sometimes, only intervals of transition rates are known. Other times, we might like to be able to specify parameters in a model to simulate how the system behaves in different scenarios. For both cases, we can extend the definition of a CTMC and their transient analysis to allow for parametric models.

Definition 2.13 (Parameter) *A numeric parameter v represents any value from a continuous interval, potentially part of the transition probability matrix or the exit rates.*

$$v \in [v^{\perp}, v^{\top}] \subseteq \mathbb{R}$$

Model parameters can occur in the exit or the transition rates or both. Each parameter adds a new dimension to the model. Naturally, all parameters span the models parameter space.

Definition 2.14 (Parameter space) *The parameter space is defined as the Cartesian product over the intervals of all parameters $v_1, ..., v_n$.*

$$\boldsymbol{P} = [v_1^{\perp}, v_1^{\top}] \times ... \times [v_n^{\perp}, v_n^{\top}]$$

Fixing all parameters in the parameter space yields a concrete parameter point.

Definition 2.15 (Parameter point) *A parameter point p is a vector of parameters, each fixed to a value from their interval.*

$$\boldsymbol{p} \in \boldsymbol{P}$$

Example 2.7 *Given parameters $v_1 \in [0, 1]$ and $v_2 \in [1, 2]$ the parameter space spans $[0, 1] \times [1, 2]$ and $p_1 = [1, 1]$ is a parameter point.*

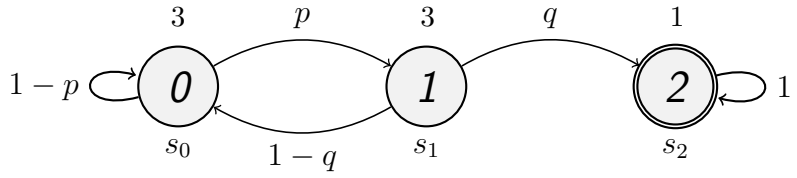
Using these definitions, the definition of a parametric CTMC follows intuitively.

Definition 2.16 (Parametric CTMC) *A pCTMC is a CTMC*

$$C_{\boldsymbol{P}} = (S, P_{\boldsymbol{P}}, r_{\boldsymbol{P}}, init, AP, L),$$

where $S, init, AP, L$ are as before and some entries in the transition probability matrix $P_{\boldsymbol{P}}$ or the exit-rates $r_{\boldsymbol{P}}$ are parameters.

Example 2.8 *We extend the previous Example 2.1 to a parametric CTMC C_p^e. $S_{C_p^e} = [0, 1, 2]$, $P_{C_p^e} = \begin{pmatrix} 1-p & p & 0 \\ 1-q & 0 & q \\ 0 & 0 & 1 \end{pmatrix}$ and $r_{C_p^e} = \begin{pmatrix} 3 & 3 & 1 \end{pmatrix}$, where $p \in (0, 0.5]$ and $q \in (0, 1)$.*



Fixing the parameter space of a parametric CTMC in all possible combinations induces a possibly infinite set of instantiated CTMCs.

Definition 2.17 (Set of CTMCs) *The parameter space of a pCTMC spans a set of CTMCs, induced by all* $\boldsymbol{p} \in \boldsymbol{P}$.

$$\boldsymbol{C} = \{C_{\boldsymbol{p}} \mid \boldsymbol{p} \in \boldsymbol{P}\}$$

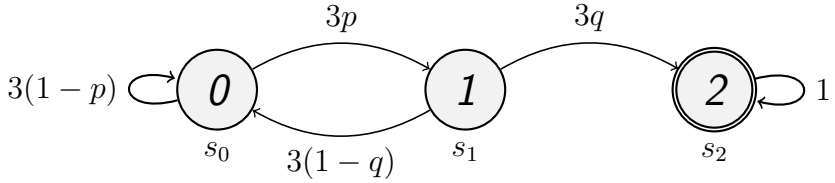
In a parametric CTMC, the transition rate matrix is constructed exactly the same as in the non-parametric case.

Definition 2.18 (Parametric transition rate matrix)

$$R^{\boldsymbol{C}}(s, s') = r^{\boldsymbol{C}}(s) * P^{\boldsymbol{C}}(s, s'),$$

where $r^{\boldsymbol{C}} : S \mapsto \mathbb{R}_{\geq 0}$ *is the exit-rate function of any state and* $P^{\boldsymbol{C}} : S \times S \mapsto [0, 1]$ *the transition probability matrix, both potentially containing parameters* $p_i \in [p_i^{\perp}, p_i^{\top}]$.

Example 2.9 *The transition rate matrix of* C_p^e *is given by* $R_{C_p^e}(s, s') = \begin{pmatrix} 3-3p & 3p & 0 \\ 3-3q & 0 & 3q \\ 0 & 0 & 1 \end{pmatrix}$.



2.6 Transient Analysis of pCTMCs

The maximum transition rate is determined in a very similar way as in the non-parametric case.

Definition 2.19 (Parametric maximum transition rate)

$$r^C_{max} = max\{r^{C_{\boldsymbol{p}}}(s) \mid C_{\boldsymbol{p}} \in \boldsymbol{C}, s \in S\}$$

Definition 2.20 (Parametric Uniformisation matrix)

$$Q^{unif(C)}(s,s') = \begin{cases} \dfrac{R^{\boldsymbol{C}}(s,s')}{q_{max}} & \text{if } i \neq j \\ 1 - \displaystyle\sum_{s'' \neq s} \dfrac{R^{\boldsymbol{C}}(s,s'')}{q_{max}} & \text{otherwise} \end{cases},$$

where $q_{max} \geq r^{\boldsymbol{C}}_{max}$ *is the uniformisation factor.*

As we pick different parameter points, the satisfaction probabilities of time-bounded CSL and transient probabilities of each state changes. These probabilities are described by the landscape function.

Definition 2.21 (Landscape function) *The landscape function* $\lambda_s^{\Phi,\boldsymbol{P}}$ *was first defined by M. Češka et al. [13]. For each parameter point* $\boldsymbol{p} \in \boldsymbol{P}$ *it returns the probability of state* s *satisfying the CSL formula* Φ*.*

$$\lambda_s^{\Phi,\boldsymbol{P}} : \boldsymbol{P} \mapsto \mathbb{R}_{\geq 0}$$

As the intervals of all parameters are real-valued, it is impossible to perform the transient analysis for each parameter point. In other words, the set $\mathbf{C}$ of CTMCs [Def. 2.17] is infinite. We approximate the landscape function by examining the bounds of the landscape function within the parameter intervals.

Definition 2.22 (Landscape function bounds) *The concrete minimum* $\overline{min}_s^{\Phi,\boldsymbol{P}}$ *and maximum bounds* $\overline{max}_s^{\Phi,\boldsymbol{P}}$ *of the landscape function* $\lambda_s^{\Phi,\boldsymbol{P}}$ *are intuitively defined over all parameter points and represent the minimum (maximum) bound of landscape functions for all parametrized CTMCs* $\boldsymbol{C}$*.*

$$\overline{min}_s^{\Phi,\boldsymbol{P}} = min\{\lambda_s^{\Phi,\boldsymbol{P}}(\boldsymbol{p}) \mid \boldsymbol{p} \in \boldsymbol{P}\}$$
$$\overline{max}_s^{\Phi,\boldsymbol{P}} = max\{\lambda_s^{\Phi,\boldsymbol{P}}(\boldsymbol{p}) \mid \boldsymbol{p} \in \boldsymbol{P}\}$$

Computing these exact bounds is also computationally infeasible due to the continuous parameter intervals. We therefore approximate these bounds in the next section.

3 Parameter Synthesis

The main technique for calculating the transient probabilities is called *min-max approximation* where the bounds of the landscape function $\lambda_s^{\Phi,\mathbf{P}}$ are approximated. This technique was presented in [4] and [13].

$$min_s^{\Phi,\mathbf{P}} : min_s^{\Phi,\mathbf{P}} \leq \overline{min}_s^{\Phi,\mathbf{P}}$$
$$max_s^{\Phi,\mathbf{P}} : max_s^{\Phi,\mathbf{P}} \geq \overline{max}_s^{\Phi,\mathbf{P}}$$

This introduces an inaccuracy of $max_s^{\Phi,\mathbf{P}} - min_s^{\Phi,\mathbf{P}}$ for any state s.

By iteratively calculating the bounds of the landscape function for any parameter space $\mathbf{P}$ and decomposing it into smaller subregions P_i, we are able to decrease the introduced inaccuracy $max_s^{\Phi,\mathbf{P}} - min_s^{\Phi,\mathbf{P}}$ to any required accuracy bound.

While this paper will focus on computing the minimized transient probabilities, the maximized vector can be obtained symmetrically [4].

3.1 Min-Max Approximation

The key idea is to locally minimize (maximize) mass flowing from one state to another during each iteration of the transient analysis. The following presents the method for computing $\pi_{\perp}^{\mathbf{C},s,t}$, the minimum transient distribution of a state s. In order to build the approximation we introduce the operator $\odot_{\perp}$ such that:

$$(\pi^{\mathbf{C},s,0} \odot_{\perp} (Q^{unif(\mathbf{C})})^i)(s') \leq min\{(\pi^{\mathbf{C},s,0} \cdot (Q^{unif(C_\mathbf{p})})^i)(s') \mid C_\mathbf{p} \in \mathbf{C}\} \quad \forall s' \in s$$

The operator $\odot_{\perp}$ ensures that the approximate lower bound remains smaller than the concrete lower bound for all parameter points. The distribution after the i-th iteration can be calculated as follows:

$$\underbrace{(\pi^{\mathbf{C},s,0}}_{\text{initial dist.}} \odot_{\perp} \underbrace{(Q^{unif(\mathbf{C})})^i}_{\text{Uniformization matrix power to i steps}} = \underbrace{((\pi^{\mathbf{C},s,0} \odot_{\perp} (Q^{unif(\mathbf{C})})^{i-1})}_{\text{Reorder to reuse previous calculations}} \odot_{\perp} Q^{unif(\mathbf{C})},$$

This avoids matrix-matrix computations and uses matrix-vector multiplication instead.

Let π be the distribution vector of the current iteration. Then the operator $\odot_{\perp}$ can be computed for all states $s \in S$ with the help of the algebraic expression $\sigma(s)$.

$$\begin{aligned}\sigma(s) &= (\pi \cdot Q^{unif(C)})(s) \\ &= \underbrace{\sum_{s' \in pred(s)} \pi(s') \cdot Q^{unif(C)}(s', s)}_{\text{All incoming mass}} + \underbrace{\pi(s)}_{\substack{\text{Mass from} \\ \text{prev. iteration}}} \underbrace{- \pi(s) \cdot \sum_{s'' \in succ(s)} Q^{unif(C)}(s, s'')}_{\text{All outgoing mass}}\end{aligned}$$

$pred(s)$ and $succ(s)$ describe the predecessor and successor set of state s. $\sigma(s)$ can be rewritten as follows.

$$\begin{aligned}
\sigma(s) &= \sum_{s' \in pred(s)} \pi(s') \cdot \frac{R(s', s)}{q_{max}} + \pi(s) - \sum_{s'' \in succ(s)} \pi(s) \cdot \frac{R(s, s'')}{q_{max}} \\
&= \sum_{s' \in pred(s) \setminus succ(s)} \pi(s') \cdot \frac{R(s', s)}{q_{max}} \\
&\quad + \sum_{s' \in pred(s) \cap succ(s)} \pi(s') \cdot \frac{R(s', s)}{q_{max}} \\
&\quad + \pi(s) \\
&\quad - \sum_{s'' \in succ(s) \setminus pred(s)} \pi(s) \cdot \frac{R(s, s'')}{q_{max}} \\
&\quad - \sum_{s'' \in succ(s) \cap pred(s)} \pi(s) \cdot \frac{R(s, s'')}{q_{max}}
\end{aligned}$$

$$\begin{aligned}
= &\left.\sum_{s' \in in(s)} \pi(s') \cdot \frac{R(s', s)}{q_{max}}\right\} \text{Incoming mass} && (1) \\
&+ \pi(s) \qquad \} \text{Mass from previous iteration} && (2) \\
&\left.+ \sum_{s' \in inout(s)} \frac{\pi(s') \cdot R(s', s) - \pi(s) \cdot R(s, s')}{q_{max}}\right\} \text{Exchanged mass} && (3) \\
&\left.- \sum_{s' \in out(s)} \pi(s) \cdot \frac{R(s, s')}{q_{max}}\right\} \text{Outgoing mass} && (4)
\end{aligned}$$

In the last equation we defined $in(s) = pred(s) \setminus succ(s)$, $inout = pred(s) \cap succ(s)$ and $out = succ(s) \setminus pred(s)$ as the set of incoming, outgoing and two-way transitions respectively.

In order to determine the approximated bounds during each iteration step the parameters that occur in the transition rate matrix are minimized or maximized according to the current state. As the minimized probability vector should be determined, the parameters that occur in (1) are also minimized so that as little mass as possible is added to the state. For (3), the parameters for calculating the incoming mass $\pi(s') \cdot R(s', s)$ are minimized while the parameters that occur in the outgoing mass $\pi(s) \cdot R(s, s')$ are maximized. Likewise, the parameters for the outgoing mass (4) are also maximized during each iteration. $\odot_\perp$ can now be calculated as $(\pi \odot_\perp Q^{unif(C)})(s_i) = \sigma(s_i)$. By multiplying the value of $\sigma(s_i)$ with the Poisson probability as described in Definition 2.12, we can compute the final probability distribution.

3.1.1 Algorithm

The pseudo-code for computing the transient probabilities is presented in Fig. 1. It takes the uniformized transition rate matrix of the pCTMC, the initial distribution and the time bound and calculates the probability distribution vector.

$POISSON(i, t)$ calculates the i-th Poisson distribution given time bound t. See Definition 2.12 for more information.

$$POISSON(i,t) = e^{-q_{max}t}\frac{(q_{max}t)^i}{i!}$$

$CALCSUMBOUND()$ calculates the sum truncation bound k_ϵ given any error bound ϵ. In our implementation, we simply iterated the error bound sum until we satisfied the given error bound. Finding the right truncation bound is crucial, since it reduces computation effort significantly by requiring fewer iterations. See Definition 2.12 for more information.

$MIN(r)$ and $MAX(r)$ minimize (maximize) a transition rate r by its parameters. We always locally minimize (maximize) the parameters, which means that a parameter minimized in one call might be maximized in another call. In our implementation, we restrict the transition rates to linear polynomials.

$$MIN(r) = \begin{cases} r, \text{where all parameters are minimized} & \text{if rate coefficient is positive} \\ r, \text{where all parameters are maximized} & \text{if rate coefficient is negative} \end{cases}$$

Line 9 corresponds to equation *(2)* in the Min-Max approximation of Section 3.1 and keeps the mass obtained from the previous iteration. Lines 10 and 11 add all incoming mass of the current iteration, see equation *(1)*. Lines 12 to 14 calculate the in-out mass as described above in equation *(3)*. Finally, lines 15 and 16 substract all outgoing mass from the current state probability, compare equation *(4)*.

After each iteration, we store the calculated probability distribution so we can access it in the next iteration (line 17). We compute the final distribution by multiplying the Poisson probability in line 18.

3.2 Parameter Space Decomposition

We introduce a technique which, given a method for finding the transient probability vector, a pCTMC C, a precision threshold ϵ_p and a probability threshold λ, refines a parameter space $\mathbf{P}$ into a set of disjunct regions $T \cup F \cup U = \mathbf{P}$, which either satisfy the bounds (T), violate the bounds (F) or are undecided within the given precision threshold (U). We expect all formulas to be of the form $\mathbb{P}_{\geq\lambda}(\diamond^{[t,t]}\Phi)$. The precision threshold ϵ_p describes the requested ratio of unknown regions: $\frac{|U|}{|P|} < \epsilon_p$.

```
function CALCTRANSIENTPROBMIN
input
  R − The uniformized pCTMC transition rate matrix
  initialDist − The initial probability vector
  t − The time bound
expects
  in, inout, out − Vectors as defined above
output
  A vector of minimized transient probability of each state
begin
1   truncBound = CALCSUMBOUND()
2   prevDist = initialDist
3   nextDist = initialDist
4   finalDist = [0 * |S|]
5   iteration = 0
6   for (iteration = 0; iteration < truncBound; iteration++)
7     prevDist = nextDist
8     for s ∈ S:
9       currProb = prevDist[s]

10      for s_in ∈ in(s):
11        currProb += (prevDist[s_in] * MIN(R[s_in, s])) / q_max

12      for s_io ∈ inout(s):
13        currProb += (prevDist[s_io] * MIN(R[s_io, s])
14                  − prevDist[s] * MAX(R[s, s_io])) / q_max

15      for s_out ∈ out(s):
16        currProb −= (prevDist[s] * MAX(R[s, s_out])) / q_max
17      nextDist[s] = currProb
18    finalDist[s] = currProb * POISSON(iteration)
end
```

Figure 1: Pseudocode for calculating the minimal transient probabilities

```
function PARAMETERSYNTHESIS
input
  C − The pCTMC
  P − The parameter space vector
  ε_p − The precision threshold
  λ − The probability threshold
output
  Set of regions, which satisfy or violate threshold ε_p
  or are undecided.
begin
1   SAT = ∅
2   VIOLATED = ∅
3   UNDECIDED = P
4   do:
5     R = DECOMPOSE(UNDECIDED)
6     for r ∈ R:
7       (min) = CALCTRANSIENTPROBMIN(C_r)
8       (max) = CALCTRANSIENTPROBMAX(C_r)
9       if min ≥ λ:
10        SAT = SAT ∪ r
11      else if max ≤ λ:
13        VIOLATED = VIOLATED ∪ r
13      else:
14        UNDECIDED = UNDECIDED ∪ r
15  until |UNDECIDED| / |P| ≤ ε_p
16  return SAT, VIOLATED, UNDECIDED
end
```

Figure 2: Pseudocode for decomposing parameter space given transient probability bounds

The algorithm (see Fig. 2) works by iteratively computing the bounds for all currently existing sub regions (line 7) and then checking whether these bounds either entirely satisfy (line 8) or violate (line 10) the requested bound. If neither is the case, the subregion will be readded to inspection and decomposed further. Even though $DECOMPOSE$ (line 5) is not described in detail, one naïve approach would be to simply decompose all regions by splitting them in half.

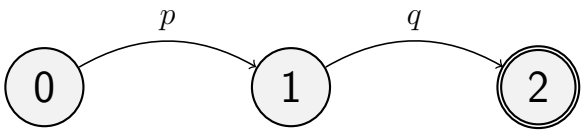


Figure 3: A visualization of the *tiny.sm* model.

4 Implementation

This chapter will focus on how we implemented the proposed Min-Max algorithm in the Storm model checker [7]. It is supposed to give an insight into implementation details and also aid any future changes in the implementation. It it not intended to provide a complete documentation, rather it will highlight several interesting aspects of our work. For a full documentation, please view the documented source files. We will also draw a comparison to the original implementation [5] in the PRISM model checker [11] by analysing their behaviour on different models.

4.1 Results

We demonstrate our implementation on two case studies. All experiments were run on a Linux workstation with an Intel Core i7-8750H CPU @ 2.20GHz with 12 cores and 16GB DDR4 RAM. For details, see Appendix 6.1.

4.1.1 Tiny Model

The tiny model displayed in Figure 3 doesn't describe any particular real-world system. Since it only contains 3 states, it is fairly easy to analyse and serves as a good introduction into refinement results. The corresponding PRISM source code is given in Appendix 6.3.

The tiny model models a fictive system with two parameters and was created to demonstrate how parameters affect the satisfiability of the model. Figure 4.1.1 visualizes the this model.

Figure 4 shows the refinement results for the region $p \in (0, 1], q \in [1, 2]$ and property $\mathbb{P}_{<0.2}(\diamond^{=1} x = 2)$. This property queries which parameter points ensure that we reach the final state $x = 2$ with a probability less than 0.2 at time point exactly 1. Green regions are regions where the parameter valuations satisfy the property, red regions are regions where they do not and the white regions represent undecided results with the given refinement options.

The splits in the boxes of the graph indicate that our implementation split a larger parameter region into smaller regions. On the left one can see, that we always split a region into

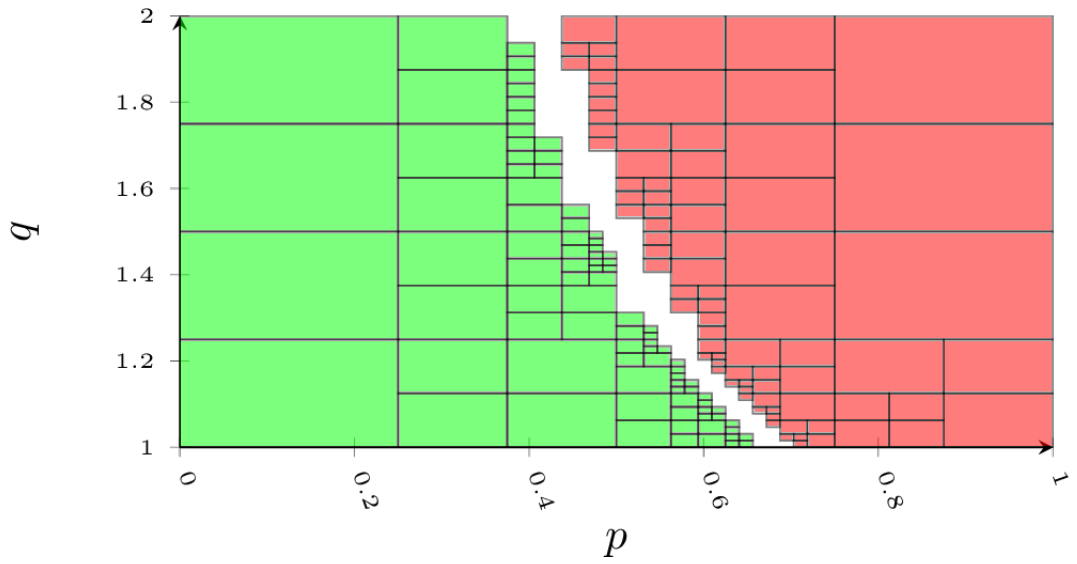


Figure 4: An example visualisation of region refinement using the tiny model

four subregions over both parameters. For example, region $p \in [0.25, 0.5] \times q \in [1, 1.25]$ is split into $[0.25, 0.375] \times [1, 1.125]$, $[0.25, 0.375] \times [1.125, 1.25]$, $[0.375, 0.5] \times [1, 1.125]$ and $[0.375, 0.5] \times [1.125, 1.25]$ respectively.

From the region refinement we can conclude that as p grows, more mass can flow to the final state and the property is violated. Since p is always smaller than q, p is more important in terms of satisfying the property $\mathbb{P} < 0.2$.

4.1.2 Cyclic Server Polling System

This case study is based on a cyclic server polling system described by Ibe et al. [10]. It models a server which cycles through a set of queues and serves each queue. N describes the number of stations that the server handles. The parameter μ describes the polling duration and γ describes the serving speed. The corresponding PRISM source code is given in Appendix 6.4.

This model was analysed with $\mu \in (0, 1]$ and $\gamma \in (0, 10]$. The property $\mathbb{P}_{<0.2}(\diamond^{=1} \quad a = 1)$ queries whether the probability that the server is currently serving a queue, is less than 0.2 at $t = 1$. The results in Figure 5 show that as we increase the polling time μ, it becomes less likely that we are serving a queue. This makes sense since a server that takes more time polling will spend less time serving the queues.

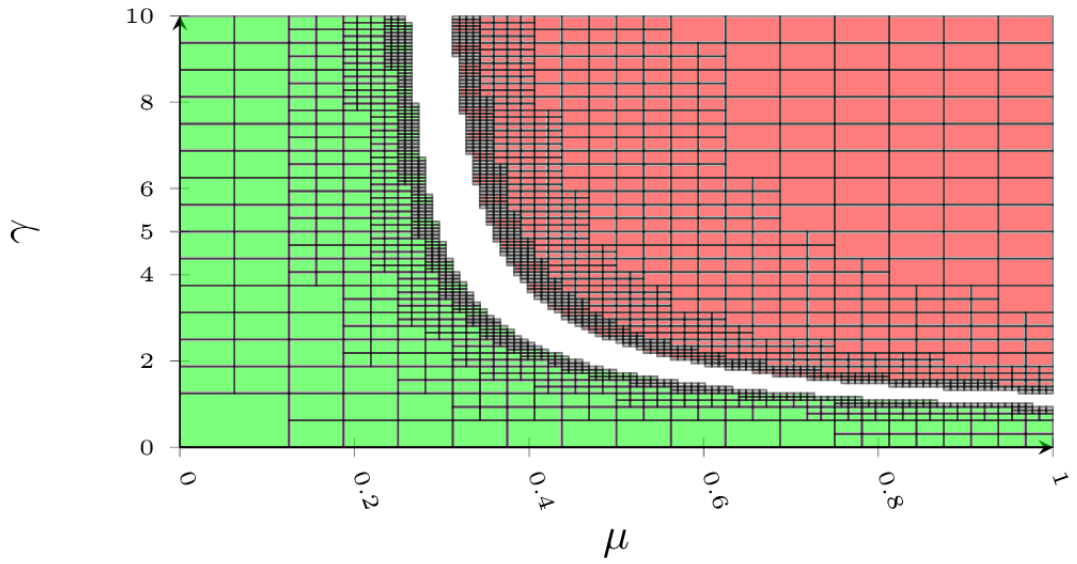


Figure 5: Region refinement of the cyclic polling model (Appendix 6.4)

4.2 Optimizations

In order to improve the usability and performance of the implementation, several optimization techniques were used.

4.2.1 Caching

After profiling the naïve implementation, it became evident that finding the ingoing and outgoing transitions and calculating the maximum and minimum transition rates was computationally expensive. Thus, we introduced a generic caching decorator, that can be used anywhere in Storm. If used, all function call results are cached in a map by the call parameters the first time the function is called. For all subsequent calls the cached value is returned and no re-computation is necessary. We used an unordered map with call parameters as keys for best performance and exploit advanced language features such as templating, operator overloading and argument expansions. For details, see the class *CacheDecorator* in *SparseCtmcParameterLiftingModelChecker.cpp*. Caching function calls made model checking almost eight times faster, see Figure 6.

4.2.2 Numerical Instability

During transient analysis, the Poisson probability has to be computed. This includes calculating e^{-qt}, which can quickly become problematic if either the maximum transition rate q or the time bound t are very high, since e^{-qt} will be very close to 0.

c	Model		Iterations		Time (s)		Time / Iteration (s)	
	States	Transitions	Storm	Prism	Storm	Prism	Storm	Prism
2	15	33	102	38	0.001	0.003	9.80E-06	7.89E-05
4	45	123	152	52	0.007	0.003	4.61E-05	5.77E-05
8	153	471	248	76	0.046	0.005	1.85E-04	6.58E-05
16	561	1,839	432	120	0.296	0.014	6.85E-04	1.17E-04
32	2,145	7,263	787	203	2.098	0.023	2.67E-03	1.13E-04
64	8,385	28,863	1476	358	16.249	0.054	1.10E-02	1.51E-04
128	33,153	115,071	2826	725	133.196	0.310	4.71E-02	4.28E-04
265	141,246	492,369	5671	1369	1268.492	3.030	2.24E-01	2.21E-03

Table 1: Performance comparison for a tandem queueing network

During implementation, we rewrote

$$1 - \sum_{i=0}^{k_\epsilon} e^{-qt} \frac{(qt)^i}{i!} \leq \epsilon \quad \text{to} \quad 1 - \sum_{i=0}^{k_\epsilon} \frac{(qt)^i}{e^{qt} i!} \leq \epsilon$$

In addition, we calculate the sum truncation bound iteratively and reuse the result of the previous iteration since

$$\sum_{i=0}^{k_\epsilon} e^{qt} \frac{(qt)^i}{e^{qt} i!} = \sum_{i=0}^{k_\epsilon} \frac{(qt)^{(i-1)}}{(i-1)!} \cdot \frac{qt}{e^{qt} i}.$$

This means that during each iteration $\frac{qt}{e^{qt} i}$ can be multiplied to the previous iteration result to calculate the next addend.

4.3 Performance

We compare our implementation within Storm to the Prism PSE implementation [5]. For the exact system specs and call arguments, see Appendix 6.1 and Appendix 6.2.

For a case study, we compare performance on a model for a tandem queueing network (Appendix 6.6) which is provided as one of the benchmarks for Prism. Information on the model state and transition count can be seen in the column *Model* of Table 1. In order to eliminate differences in how regions are chosen we only test performance on one region at a time. The queue capacities can be adjusted by a model parameter c, which is used to demonstrate performance on arbitrarily sized models. Since Storm is implemented in C++ and Prism runs Java, Storm inherently has a slight performance advantage over Prism. Nonetheless, our implementation turned out to be slower. This is mostly due to the fact that Prism PSE uses Fox-Glynn approximation [8] to calculate the sum truncation bound, while we use a naïve approach. As a result, Prism PSE requires significantly less iterations than Storm. The *Iterations* column displays this implementation difference. Still, Prism is also faster per iteration for larger models, see column *Time per Iteration*. Using a lower

truncation bound is more effective with higher state counts, which explains why Storm performs much worse on larger models.

Figure 6 displays performance on various models for a single region call and also demonstrates the effectiveness of our caching optimizations. Model 1 to 4 compares caching on the tandem model with different parameters and model 4 to 8 compares the SIR model (Appendix 6.5). The plots show the time it took for the transient analysis to finish in the cached and non-cached case. We also give the timings of Prism for reference comparison. Analysis takes a lot longer as the models get larger or the time bound higher. A detailed specification of all models can be found in Appendix 6.7.

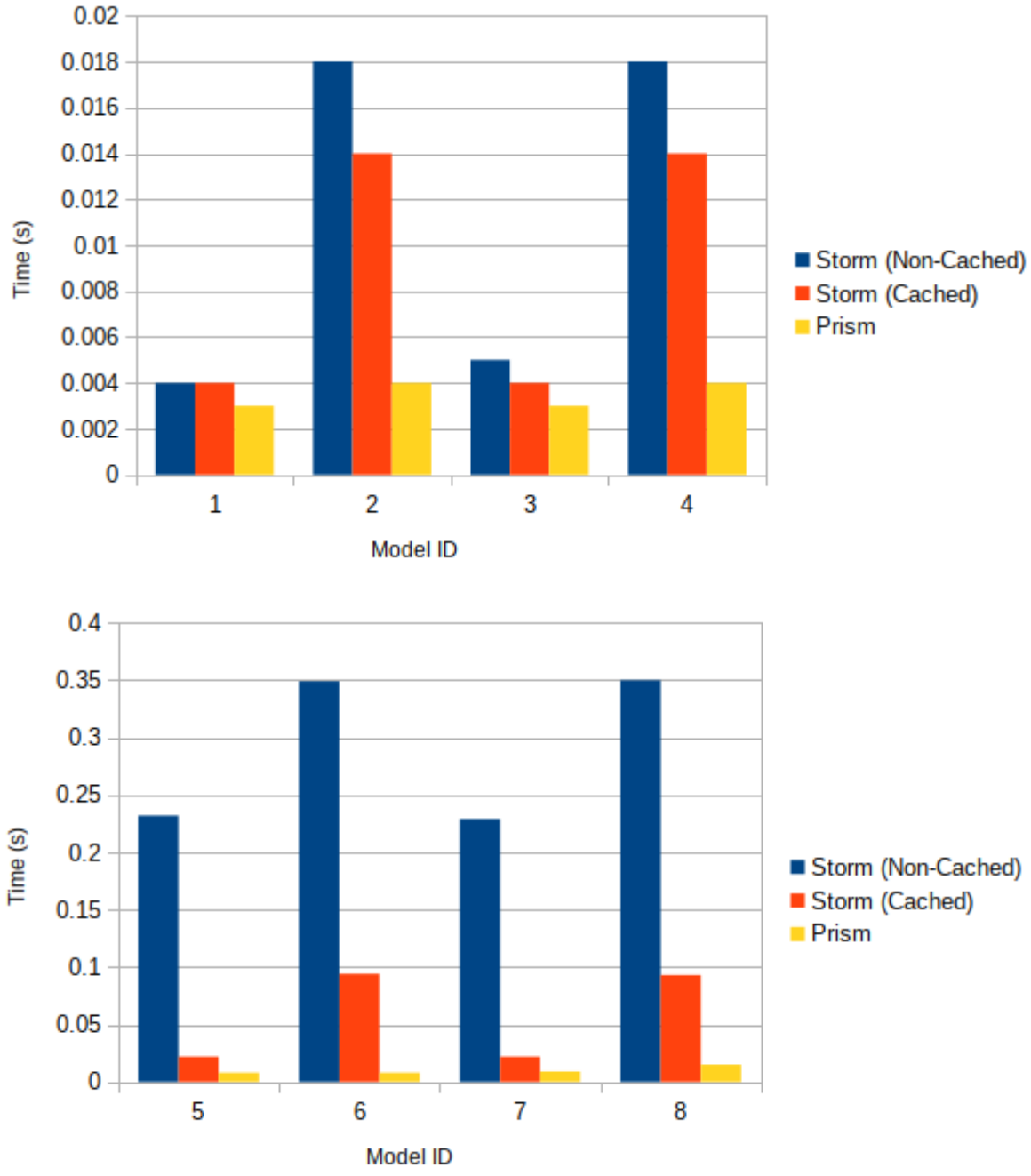


Figure 6: Performance comparison for Non-Cached, Cached and Prism (Various models)

5 Conclusion

5.1 Discussion

Being able to model and analyse parametric systems is a crucial subset of verification. Parametric CTMCs provide the perfect foundation for real-time related models. During default transient analysis we compute the uniformized CTMC to avoid numerical instability. This yields the uniformized transition rate matrix which is used to iteratively compute the probability distribution up to a given truncation bound. This truncation bound can be adjusted to satisfy any given error bound. This method however doesn't suffice for parametric models.

This thesis built on the results of [4] and [13] to provide a gentle introduction into approximation for parametric transient analysis. By minimizing (maximizing) the outgoing and incoming mass per iteration, we are able to determine approximate bounds of the probability distribution vector. We provide multiple pseudo-code algorithms that build an understanding of the Min-Max approximation. The error introduced by this technique can be reduced by region refinement, where we alternate computing the region bounds and splitting the given region into smaller sub regions. This way, we can refine the approximation until a given error bound is reached.

We also implemented the proposed algorithms in the Storm model checker [7]. This proved more challenging than expected since the existing code base is very advanced and does not have a lot of documentation. During implementation, various techniques such as caching were deployed to improve performance. Comparing our implementation to Prism, it turned out that our prototypical implementation is slower. This is mostly due to the fact that Prism uses Fox-Glynn approximation [8] which yields a much lower iteration bound.

5.2 Outlook

In the future, several optimizations to the implementation could be implemented. A possible first step would be to use Fox-Glynn approximation to compute a potentially much lower sum truncation bound. This will improve performance, as can be seen in Table 1.

While we focused on optimizations per region, one could also consider optimizations inbetween region refinements. One such optimization would be to remember the upper and lower approximate bounds of the probability distribution vector and carry these bounds into the next region refinement. This would allow to skip computing some of the lower and upper bounds for a second time.

6 Appendix

6.1 System Specs

All results in this paper were achieved by the following system specs:
- Ubuntu 18.04.3 LTS
- Intel Core i7-8750H CPU at 2.20GHz with 12 cores
- 16GB DDR4 RAM

6.2 Program Arguments

Storm was run with the following parameters:
bin/storm-pars –prism MODELPATH -pc –prop PROP –parametric –region:engine pctmc –region REGION

Prism was run with the following parameters:
bin/prism MODELPATH –pse TIMEBOUND REGION 1e-06

6.3 Model: tiny.sm

```
ctmc

const double p;
const double q;

module TINY

    x : [0..2] init 0;

    [] x=0 -> p:(x'=1);
    [] x=1 -> q:(x'=2);

endmodule
```

Figure 7: tiny.sm: A simple model used for testing

6.4 Model: cyclic.sm

```
ctmc
const int N=2;
const double mu;
const double gamma= 10;
module server
        s : [1..2]; // station
        a : [0..1]; // action: 0=polling, 1=serving

        [loop1a] (s=1)&(a=0) -> gamma    : (s'=1);
        [loop1b] (s=1)&(a=0) -> gamma    : (a'=1);
        [serve1] (s=1)&(a=1) -> mu       : (s'=1)&(a'=0);

        [loop2a] (s=2)&(a=0) -> gamma    : (s'=1);
        [loop2b] (s=2)&(a=0) -> gamma    : (a'=1);
        [serve2] (s=2)&(a=1) -> mu       : (s'=1)&(a'=0);
endmodule
module station1
        s1 : [0..1];

        [loop1a] (s1=0) -> 1 : (s1'=0);
        []       (s1=0) -> mu/N : (s1'=1);
        [loop1b] (s1=1) -> 1 : (s1'=1);
        [serve1] (s1=1) -> 1 : (s1'=0);
endmodule
module station2 = station1 [s1=s2,
 loop1a=loop2a, loop1b=loop2b, serve1=serve2] endmodule
```

Figure 8: cyclic.sm: A cyclic polling network with two stations

6.5 Model: sir.sm

```
ctmc

const double ki;
const double kr;

const int maxPop = 30;
const int initS = maxPop-5;
const int initI = 5;
const int initR = 0;

module SIR

popS: [0..maxPop] init initS;
popI: [0..maxPop] init initI;
popR: [0..maxPop] init initR;

[] popS > 0 & popI > 0 & popI < maxPop  ->     ki*popS*popI
: (popS'= popS-1) & (popI'= popI+1);
[] popI > 0 & popR < maxPop ->     kr*popI
: (popR'= popR+1) & (popI'= popI-1);
[] popI=0 -> 1 : true;

endmodule

rewards
  (popI > 50) : 1;
endreward
```

Figure 9: sir.sm: A SIR epidemic model, taken from Prism-PSY. [5]

6.6 Model: tandem.sm

```
ctmc
const int N=2;
const double mu;
const double gamma;
module server

        s : [1..2]; // station
        a : [0..1]; // action: 0=polling, 1=serving

        [loop1a] (s=1)&(a=0) -> gamma    : (s'=1);
        [loop1b] (s=1)&(a=0) -> gamma    : (a'=1);
        [serve1] (s=1)&(a=1) -> mu       : (s'=1)&(a'=0);
endmodule

module station1
        s1 : [0..1];
        [loop1a] (s1=0) -> 1 : (s1'=0);
        []       (s1=0) -> mu/N : (s1'=1);
        [loop1b] (s1=1) -> 1 : (s1'=1);
        [serve1] (s1=1) -> 1 : (s1'=0);
endmodule

module station2 = station1 [s1=s2,
    loop1a=loop2a, loop1b=loop2b, serve1=serve2]
endmodule
```

Figure 10: tandem.sm: A model which simulates a tandem queueing system, taken from HMKS99. [9]

6.7 Models for Cache comparison

Model ID	Model	Parameters	Timebound
1	tandem.sm	{"kappa": [4,4]}	1
2	tandem.sm	{"kappa": [4,4]}	5
3	tandem.sm	{"kappa": [3,4]}	1
4	tandem.sm	{"kappa": [3,4]}	5
5	sir.sm	{"ki":[0.003,0.003],"kr":[0.2,0.2]}	1
6	sir.sm	{"ki":[0.003,0.003],"kr":[0.2,0.2]}	10
7	sir.sm	{"ki":[0.001,0.003],"kr":[0.1,0.2]}	1
8	sir.sm	{"ki":[0.001,0.003],"kr":[0.1,0.2]}	10

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous time markov chains. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 269–276, London, UK, UK, 1996. Springer-Verlag.

[2] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time markov chains by transient analysis. volume 1855, pages 358–372, 07 2000.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[4] L. Brim, M. Češka, S. Dražan, and D. Šafránek. Exploring parameter space of stochastic biochemical systems using quantitative model checking. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 107–123, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[5] M. Češka, P. Pilař, N. Paoletti, L. Brim, and M. Kwiatkowska. Prism-psy: Precise gpu-accelerated parameter synthesis for stochastic systems. In M. Chechik and J.-F. Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–384, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[6] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[7] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk. A storm is coming: A modern probabilistic model checker. In R. Majumdar and V. Kunčak, editors, *Computer Aided Verification*, pages 592–600, Cham, 2017. Springer International Publishing.

[8] B. L. Fox and P. W. Glynn. Computing poisson probabilities. *Commun. ACM*, 31(4):440–445, Apr. 1988.

[9] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

[10] O. C. Ibe and K. S. Trivedi. Stochastic petri net models of polling systems. *IEEE J.Sel. A. Commun.*, 8(9):1649–1657, Dec. 1990.

[11] M. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36(4):40–45, Mar. 2009.

[12] J. Nellen, T. Rambow, M. T. B. Waez, E. Ábrahám, and J. Katoen. Formal verification of automotive simulink controller models: Empirical technical challenges, evaluation and recommendations. In *Formal Methods - 22nd International Symposium, FM 2018,*

Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings, pages 382–398, 2018.

[13] M. Češka, F. Dannenberg, N. Paoletti, M. Kwiatkowska, and L. Brim. Precise parameter synthesis for stochastic biochemical systems. *Acta Inf.*, 54(6):589–623, Sept. 2017.