University of Twente
Enschede - The Netherlands

Formal
Methods
& Tools

# Probabilistic model checking
# A comparison of tools

Masters Thesis
in Computer Science
of

H.A. Oldenkamp

born on the 23th of September 1979

May, 2007

Chairman:
Prof. Dr. Ir. Joost-Pieter Katoen

Supervisors:
Ivan S. Zapreev MSc
Dr. David N. Jansen
Dr. Mariëlle I.A. Stoelinga

University of Twente,
Faculty EEMCS,
Computer Science Department,
Formal Methods and Tools Group

# Summary

Model checking is a technique to establish the correctness of hardware or software systems in an automated fashion. The goal of this technique is to try to predict system behaviour, or more specifically, to formally prove that all possible executions of the system conform to the requirements. *Probabilistic* model checking focusses on proving correctness of stochastic systems (i. e. systems where probabilities play a role). A probabilistic model checker tool automates the correctness proving process. These tools can verify if a system – which is described by a model, written in a formal language – satisfies a formal specification, which is expressed using logics, such as Probabilistic Computation Tree Logic (PCTL). We have studied the efficiency of five probabilistic model checker tools, namely: PRISM (Sparse and Hybrid mode), MRMC, ETMCC, YMER and VESTA. We made a tool by tool comparison, analysing model check times and peak memory usage. This was achieved by using five representative case studies of fully probabilistic systems, namely; Synchronous Leader Election (SLE), Randomized Dining Philosophers (RDP), Birth-death process (BDP), Tandem Queuing Network (TQN) and Cyclic Server Polling System (CSP). Besides their performance, we also investigated the characteristics of each tool, comparing their implementation details, range of supported probabilistic models, model specification language, property specification language and supported algorithms and data structures. During our research, we have performed nearly 15,000 individual runs. By ensuring that our experiments are automated, repeatable, verifiable, statistically significant and free from external influences, our findings are based on a solid methodology.

We have witnessed a significant difference in model check time as well as memory usage between the tools. From our experiments we learned that YMER (which is a statistical tool) is by far the best tool for verifying medium to large size models. It outperforms the other statistical model checker VESTA and all numerical tools. YMER has a remarkably consistent (low) memory usage across various model sizes. Although its performance is excellent, we found that YMER does have limitations: the range of supported models and probabilistic operators is limited and it can not provide the same level of accuracy as numerical tools. YMER may occasionally report wrong answers, but this can be expected of tools using a statistical approach, where there exists a trade-off between speed and accuracy. The benefit of statistical tools is that they scale much better (performance wise) in relation to the state-space size than the numerical tools.

Comparing the numerical tools we conclude that MRMC has the best performance (time and memory wise) for models up to a few million states. This is especially true for steady-state and nested properties[1], for other properties (i. e. bounded Until,interval Until and unbounded Until) MRMC and PRISM$^{Sparse}$ are rather close. On larger models, PRISM$^{Sparse}$ (and sometimes also PRISM$^{Hybrid}$) performs better. The sparse engine is usually faster than the hybrid engine at the cost of substantially greater memory usage. As for ETMCC, it has the worst time and memory performance, it frequently runs out of memory in situations where the models could easily be checked by the other tools. The performance differences between the two leading numerical tools MRMC and PRISM have several causes. First of all, PRISM always construct an MTBDD (Multiterminal Binary Decision Diagram), in sparse mode the MTBDD is converted to a sparse matrix after performing some pre-computations. This may take a significant time and influence the model check time. There is no such influence for MRMC as it starts model checking on the pre-generated sparse matrix. Secondly, the MTBDD size plays a crucial role in PRISMs performance. Large MTBDDs lead to poor performance of the hybrid engine. The difference between MRMC and PRISM$^{S}$ is caused by the pre-computation step that is performed by PRISM on the MTBDD. The pre-computation time is included in the model check time. Finally, MRMCs performance on large models might be influenced by its high memory usage, in situations where the memory usage exceeds the systems RAM space, swapping will cause a slow down.

On the aspect of user friendliness we find PRISM the most user friendly tool. MRMC is more appropriate as a fast back-end verification engine as it has a simple input format.

[1] We verified two nested properties, namely for the BDP case study: $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[true\,\mathcal{U}^{\leq 100}\ (n=70)]\ \mathcal{U}\ (n=50)]$ and for TQN: $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ snd]]$

# Preface

This master thesis project has been a challenging and educational activity. I have had the opportunity to work on a project that deals with collecting, analysing and interpreting vast amounts of information, which honed my methodical and analytical capabilities. Another benefit of this project is that I gained an insight into the field of probabilistic model checking, which was a relatively new subject to me.

I would like to thank all of my supervisors for there patience and thorough feedback. I am also grateful for the information provided by Dr. Dave Parker from the University of Birmingham on the inner workings of the PRISM tool. On a final note I would like to say to my first supervisor, Ivan Zapreev:
ИВАН, ОЧЕНЬ ПРИЗНАТЕЛЕН ЗА ГВОЮ НЕОЦЕНИМУЮ ПОМОЩЬ.

Marcel Oldenkamp
May, 2007

# Glossary

| | |
|---|---|
| **BDD** | Binary Decision Diagram |
| **BDP** | Birth-Death Process |
| **BSCC** | Bottom Strongly Connected Component |
| **CMRM** | Continuous Markov Reward Model |
| **CSL** | Continuous Stochastic Logic |
| **CSP** | Cyclic Server Polling |
| **CSRL** | Continuous Stochastic Reward Logic |
| **CTMC** | Continuous-Time Markov Chain |
| **CUDD** | CU Decision Diagram |
| **DMRM** | Discrete Markov Reward Model |
| **DTMC** | Discrete-Time Markov Chain |
| **ETMCC** | Erlangen-Twente Markov Chain Checker |
| **GPL** | General Public License |
| **GSMP** | Generalized Semi-Markov Process |
| **GUI** | Graphical User Interface |
| **J2SE** | Java 2 Standard Edition |
| **JOR** | Jacobi Over-Relaxation |
| **JVM** | Java Virtual Machine |
| **MDP** | Markov Decision Process |
| **MOVES** | Software Modelling and Verification |
| **MRMC** | Markov Reward Model Checker |
| **MTBDD** | Multi-Terminal Binary Decision Diagram |
| **OS** | Operating System |
| **OSSD** | On-the-fly Steady-State Detection |
| **PCTL** | Probabilistic Computation Tree Logic |
| **PEPA** | Performance Evaluation Process Algebra |
| **PRCTL** | Probabilistic Reward Computation Tree Logic |
| **PRISM** | Probabilistic Symbolic Model Checker |
| **QuaTEx** | Quantative Temporal Expressions |
| **RAM** | Random Access Memory |
| **RDP** | Randomized Dining Philosophers |
| **RSS** | Resident Set Size |
| **RWTH** | Rheinisch-Westfälische Technische Hochschule |
| **SLE** | Synchronous Leader Election |
| **SOR** | Successive Over-Relaxation |
| **State-space** | The set of all possible states and transitions. |
| **STD** | Student’s t-Distribution |
| **Swap** | The memory space (i. e. RAM) of a computer can be extended by using the hard drive as memory. This is called swap. Note that accessing swap memory typically takes a lot longer than accessing RAM. |
| **SZ** | Size (memory) in physical pages |
| **TQN** | Tandem Queuing Network |
| **VSZ** | Virtual Memory Size (RAM + swap) |

# Contents

# Chapter 1

# Introduction

In the early days of software (and hardware) development, it was (and regularly still is) common practice to write software first and (perhaps) test it later. The concept of verifying software/hardware correctness by means of testing often proved inadequate for complex systems. Experience learned that the effect of software bugs can vary from causing a slight inconvenience to disastrous effects, such as the explosion of the Ariane 5 launch system. Since the early 80's, people have been working on a way to *formally* verify whether a system satisfies a certain behavioural property by means of a technique called model checking [21]. Model checking is a technique to establish the correctness of a system. In contrast to testing, model checking looks at *all* possible behaviours of a system. While testing can only find errors, formal verification by means of model checking can also prove their absence. It enables expressing properties to which the answer is "yes" or "no", such as: it is never the case that traffic lights "$A$" and "$B$" are green simultaneously. While formal verification focuses on the absolute correctness of systems, in practice such inflexible demands are hard, or even impossible, to guarantee. Instead, systems are subject to various phenomena of stochastic nature, such as message loss or garbling, unpredictable environments, faults, and delays. Correctness thus is of a less absolute nature. Accordingly, instead of checking whether system failures are impossible, a more realistic aim is to establish, for instance, whether "the chance of failure is at most 0.01%". Such properties can be checked using probabilistic model checking. There are many software tools available that automate the process of probabilistic model checking. Such tools accept a system model description ("the possible behaviour"), a specification of the property to be considered (the "desirable behaviour"), and then systematically check the validity of the property on the given model.

We are interested in the performance difference between available probabilistic model checker tools. We desire an in-depth analysis of the difference in speed (i. e. how much computation time does the tool require to verify a particular model and property) and memory usage (i. e. the maximum amount of memory consumed by the tool). In addition, we are interested in the overall differences amongst the tools, such as their variety of supported probabilistic models and logical operators for property specification. This thesis sets out to compare several model checkers for probabilistic systems (henceforth called tools), by means of an empirical study, and attempts to observe and explain relevant phenomena.

## 1.1 Approach

We selected five probabilistic model checker tools for use in an experimental comparison. Limiting ourselves to five tools allowed for an in-depth research between their differences in the time allowed. For those familiar with probabilistic model checking we note that three out of the five tools utilise "numerical" techniques, the others are based on "statistical" methods (i. e. simulation and sampling). These terms will be explained in Chapter 2. In order to compare the different tools we established a list of key factors which all tools have in common and that are of interest to compare. In the initial phase we investigated the general characteristics of each tool and compared their implementation details, range of supported probabilistic models, model specification language, property specification language (i. e. temporal logic operators) and supported algorithms & data structures. This was accomplished by investigating available documentation and publications related to the tools and naturally by using the tools themselves. The core of the project involved an in-depth study into the performance differences between each of the tools. We therefore constructed a test environment to gather and analyse data related to:

1. *speed* - the computation time required to verify a particular model and property
2. *memory usage* - the peak memory consumed by the tool during its execution

Using five representative case studies taken from the literature on performance evaluation and probabilistic model checking we performed several experiments. For each case study we generated equivalent[1] models written in the model description language of each individual tool. We constructed a set of properties expressed in the temporal logic PCTL (Probabilistic Computation Tree Logic) or CSL (Continuous Stochastic Logic) and arranged for the size of the model to be adjustable. We then collected performance data by letting each tool verify the properties on all the model sizes.

In order to gather reliable data, we created an isolated test environment, meaning; we made sure the conditions for each experiment were the same except for the *independent*[2] variables (i. e. the ones we want to manipulate, namely the model size and property). We used a standard Personal Computer that was designated to this project only, such that others could not inadvertently change the environment. All the data, such as tool installation files, input models and result files were stored on the local hard drive, as to avoid influence in performance measurements due to network traffic issues. The tools were considered as a “black box”, meaning we made external measurements. This ensured a uniform method of collecting performance data, independent of the tools implementation. We did not build any measurement constructions into the source code of the tools, since this might influence their performance, in addition, this would require the availability of the source code of each tool. The verification parameters of each tool were set to corresponding levels. For example, the error bound $\epsilon$, which is used in solving a system of linear equations, is set to $10^{-6}$ for all non-statistical tools.

**Automation.** Because we created multiple properties and model sizes per case study, and had to verify each combination using the five tools, automation became a necessity (we performed nearly 15,000 individual runs). All experiments and measurements were automated by means of Linux shell scripts. This enabled us to easily repeat experiments many times and collect data in a uniform style. An experiment consisted of verifying one property on one particular model size using one of the model checker tools. The tools were restarted before each experiment; this prevents features such as caching to influence the results. Each experiment was repeated 20 times, after which we calculated the sample mean and standard deviation of data such as elapsed time. The number of runs was limited to 3 in stead of 20 whenever the total time of a single run exceeded 30 minutes, this prevented experiments from consuming excessive amounts of time, but resulted in a less accurate standard deviation. To counter this effect we used the student’s t-distribution [26], which takes the number of samples into account. The raw data produced by the tools was processed automatically by means of shell scripts and a Java application that we designed to perform the necessary calculations and generate results for easy display in LaTeX[3].

[1] With equivalent models we mean that they model the same system and reflect identical (possible) behaviour.
[2] An *independent* variable is selected and manipulated by the experimenter to observe its effect on the dependent variable (i. e. the variable that is observed and measured) [31].
[3] LaTeX is a document preparation system for high-quality typesetting, see `www.latex-project.org`.

**Organisation of the thesis.** Chapter 2 presents the necessary background information on (probabilistic) model checking, including the different probabilistic models such as Discrete-Time Markov Chains and Continuous-Time Markov Chains and the logics PCTL and CSL for property specification. Chapter 3 offers a tool by tool overview ranging from general background information to supported logical operators, algorithms and data structures. In Chapter 4 we perform a series of experiments on five probabilistic model checker tools to measure and compare their efficiency. We utilize five well known case studies and compare the performance results of each tool. We measure and analyse the time and peak memory usage required by each tool (to verify a specific PCTL/CSL property). Finally, Chapter 5 concludes the thesis.

In this thesis we denote footnotes by a superscripted digit$^0$ and references are placed between brackets [0]. A glossary explaining the abbreviations and technical terms is listed on page 7.

Chapter 2

# Background on probabilistic model checking

This chapter gives an introduction to (probabilistic) model checking, it elaborates the basic concepts relevant for this thesis, starting with model checking in general and moving on to probabilistic model checking, probabilistic models and logics for checking probabilistic models.

## 2.1 Model checking

When designing systems (hardware or software) it is important to know whether or not your system will operate as expected/required. For instance you do not want to find out after implementation and delivery of your nuclear power plant control system that it is not as safe as was expected. There are several techniques available that can be used to verify the functional correctness of a system. Examples of such techniques are: theorem proving, simulation/testing and model checking. This thesis focuses on the last technique, namely formal verification by means of model checking [21]. The goal of this technique is to try to predict system behaviour, or more specifically, to formally prove that all possible executions of the system conform to the requirements. Typical problems that are addressed are [38]:

- Safety [6]: e. g. does a given mutual exclusion algorithm guarantee mutual exclusion?
- Liveness [6]: e. g. will a packet transferred via a routing protocol eventually arrive at the correct destination?
- Fairness [29]: e. g. will a repeated attempt to carry out a transaction be eventually granted?

As the name suggests, model checking is performed on a model of a system. The model is usually generated from a high level system description, such as process algebra or Petri net. Typically, these generated models are non-deterministic finite-state automata[1]. These automata (i. e. transition systems) describe the possible system behaviour. They can be seen as directed graphs consisting of a finite set of states (nodes) labelled with atomic propositions and state transitions (edges) that show how the system can change from one state to another. The atomic propositions represent the basic properties that hold in each state. Once the system is represented by a model we can check if the model satisfies a formal specification (i. e. if it has certain properties). The properties that are checked against the system model are expressed using logics, such as LTL (Linear Time Logic) or CTL (Computation Tree Logic). These logics can express properties on states or paths in the automata. Once a model and property have been formulated the model checking process will perform a systematic state-space exploration to verify if the property holds. This form of traditional model checking focuses on delivering a 100% accurate guarantee whether or not a property holds in a certain model. There are many cases where giving an absolute guarantee

[1] The interested reader is referred to [72] for detailed information on finite-state automaton.

is not feasible or even impossible. Examples are communication protocols, such as Bluetooth or IEEE 802.11 Wireless LAN, that have to deal with a certain probability of message loss. This is where probabilistic model checking can be utilized.

## 2.2 Probabilistic model checking

Probabilistic model checking is an automatic formal verification technique for the analysis of systems which exhibit *stochastic* behaviour [41]. The technique is similar to model checking as discussed earlier. The major difference is that a probabilistic model contains additional information on likelihood or timing of transitions between states, or to be more specific, it can model stochastic behaviour. Probabilistic model checking refers to a range of techniques for calculating the probability of the occurrence of certain events during the execution of the system, and can be useful to establish properties such as “shutdown occurs with probability at most 0.01” and “the video frame will be delivered within 5ms with probability at least 0.97” [50]. Applications range from areas such as randomized distributed algorithms to planning and AI, security [60], and even biological process modelling [55] or quantum computing. An overview of the probabilistic model checking procedure is given in Figure 2.1. It shows that a probabilistic model checker takes as input a property and a model and delivers the result “Yes” or “No” (i. e. whether or not the property is satisfied) or some probability. The following sections elaborate on the input of the model checker tools, namely the model types and logics. But first an example is introduced, that is used throughout this chapter.

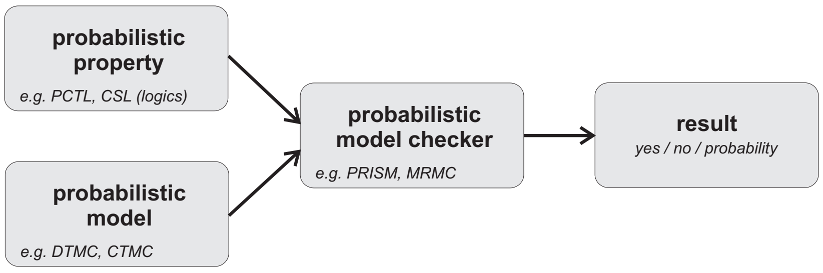


Figure 2.1: Probabilistic model checking overview

**The Hubble space telescope example.** This example (an adaptation from [38]) models the failure behaviour of the Hubble space telescope[2]. We start by describing the real system, which consists of parts that can possibly fail. Once the system is understood, we will model its failure behaviour, so that we may predict its behaviour and ask question such as: “What is the probability that the system will operate without failure for the next 10 years?”

**System.** The system has a steering unit with six gyroscopes, which are used to aim the telescope. Redundancy is an important issue when designing systems such as the space telescope, since it is obvious that performing repairs is not trivial. This is why the telescope is designed in such a way that it will still function with full accuracy when only three of its six gyroscopes are operational. With less than three gyroscopes the telescope turns into sleep mode, meaning repairs are necessary[3]. If none of the gyroscopes are operational the telescope will crash. No repair mission will be undertaken, as long as there are more than two gyroscopes operational.

**Model.** The model of the system that has just been described is depicted in Figure 2.2. The model has a total of nine states. Each state has a number and label (shown respectively inside and

[2] The Hubble space telescope is an astronomical observatory orbiting around earth, since April 24, 1990.
[3] In reality, since its launch there have been four service missions to the telescope; 1993, 1997, 1999 and 2002.

outside side the state symbol). States 1 through 6 are not labelled explicitly; their labels equal their state number, which represents the number of operational gyroscopes. An edge labelled $f$ represent a failure event of a gyroscope, $r$ means a repair mission has been undertaken successfully and $s$ means the telescope is going to sleep. States 7, 8, and 9 are labelled *sleep2*, *sleep1* and *crash*. State 6 is considered the initial state where all gyroscopes are operational; this is also the state of the telescope after a successful repair mission. The state with label *crash* is a terminating state[4]. We now have a model of our system that shows its possible behaviour. Later on we expand this model by adding probabilities and rates.

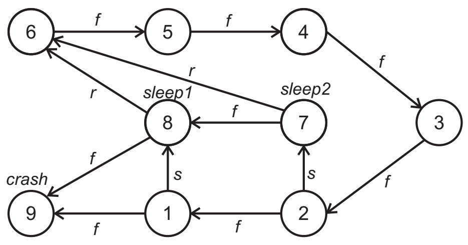


Figure 2.2: State transition system of the Hubble space telescope

## 2.3 Probabilistic models

In order to model check a system that exhibits stochastic behaviour we will first need to build a formal probabilistic model of that system. There are several commonly used probabilistic model representations for stochastic system, for example:

- Discrete-Time Markov Chains (DTMC)
- Continuous-Time Markov Chains (CTMC)
- Markov Decision Process (MDP)
- Stochastic Petri nets
- Bayesian networks

These models usually form a combination of probability theory and graph theory. They consist of states, transitions, and arcs that connect them. From now on we will focus on Markov chains. These particular models are used by all probabilistic model checker tools studied in this thesis. A Markov chain should be considered as a transition system, where we can move from one state to another and the choice of which state to go to depends on some probability distribution. Moving from one state to another is referred to as “making a transition”. A more formal definition will be presented later on. We will only deal with Markov chains that have a finite or countable set of states, if this condition is not met, we speak of a Markov process[5] [61]. A Markov chain has a very specific characteristic, namely that it retains no memory of earlier transitions. This is called the Markov property. It means that only the current state of the process can influence the probability of next transitions. We consider two different types of Markov chains, which are frequently supported by probabilistic model checker tools:

1. DTMC - Discrete-Time Markov Chain
2. CTMC - Continuous-Time Markov Chain

The Markov chains discussed here are considered time-homogeneous (i. e. the transition matrix, containing the probabilities or rates, remains constant). The meaning of transition matrix will become clear in the next sections, where a brief explanation is given of DTMCs and CTMCs. For a more elaborate treatment see [74].

[4] A terminating state, also called absorbing state, is a state that once entered cannot be exited.
[5] When a Markov process has a *discrete* state-space (set of all possible states and transitions) we call it a Markov chain.

### 2.3.1 Discrete-Time Markov Chains (DTMC)

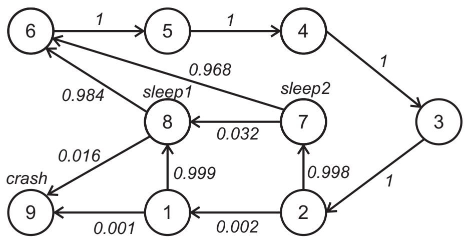
A DTMC is a transition system that defines the probability of moving from one state to another.

**Definition 2.1** (labelled DTMC)**.** A labelled DTMC is a quadruple $< S, \overline{s}, P, L >$ where:

- $S$ is a finite set of states,
- $\overline{s} \in S$ is the initial state,
- $P : S \times S \rightarrow [0, 1]$ is the transition probability matrix, where $P(s, s')$ is the probability of moving from state $s$ to $s'$,
- $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$,
- $L : S \rightarrow 2^{AP}$ is the labelling function, which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ that are valid in $s$.
  (adapted from [33])

According to [49], DTMCs are stochastic models of systems with countable state-space that change their states at times $n = 0, 1, 2, \ldots$ and have the following property: if the system enters state $s$ at time $n$, it stays there for exactly one unit of time and then jumps to state $s'$ at time $n+1$ with probability $P(s, s')$, regardless of its history up to and including time $n-1$. The definition shows that states are labelled with atomic propositions, they indicate for instance the status of the system (e. g. *waiting*, *sending*). The system can change state according to a probability distribution given by the transition probability matrix. A transition from state $s$ to $s'$ can only take place if $P(s, s') > 0$. If $P(s, s') = 0$, no such transition is possible. The system can occupy the same state before and after the transition, since according to definition 2.1 self-loops are allowed. A sequence of states and transitions forms a path, where a path is defined as a finite or infinite sequence $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} \ldots \xrightarrow{p_{i-1}} s_i \longrightarrow \ldots$ with $i \in \mathbb{N}, s_i \in S$, and $p_i \in \mathbb{R}_{\langle 0,1]}$ such that $p_i > 0 \wedge p_i = P(s_i, s_{i+1})$ for all $i \geq 0$.

We represent a DTMC as a transition diagram, where states are depicted by circles, state labels by text outside the circle and transitions with non-zero probabilities by arrows labelled with their probabilities. Figure 2.3 shows the DTMC model and matrix $P$ of the Hubble space telescope (first introduced on page 16). The number of states and transitions have remained the same, only now the transitions are assigned with probability values. It shows that as long as there are more than two gyroscopes operational the next gyroscope will fail with probability 1. This is represented by the states labelled 6, 5, 4 and 3 with outgoing transitions that have probability 1. In the situation where two gyroscopes are operational, the system can either go to the *sleep* mode with probability 0.998 or one of the remaining gyroscopes may fail with probability 0.002, which is depicted by the outgoing transitions of state 2. Each possible state transition with its corresponding probability is shown in the transition system and probability matrix in Figure 2.3.



(a) Transition system

$$P = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.999 & 0.001 \\
0.002 & 0 & 0 & 0 & 0 & 0 & 0.998 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.968 & 0 & 0.032 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.984 & 0 & 0 & 0.016 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

(b) Probability matrix

Figure 2.3: DTMC of the Hubble space telescope

### 2.3.2 Continuous-Time Markov Chains (CTMC)

A CTMC can be seen as an extension of the DTMC. The difference is that a DTMC models discrete time steps and a CTMC allows the modelling of real (continuous) time. This means that state changes in a CTMC can occur at any arbitrary time, as opposed to fixed time $n = 0, 1, 2, \ldots$ in a DTMC. The memoryless property still applies, meaning that the probability of moving to a future state depends only on the current state. CTMCs are often used for analysing performance and dependability of systems. Two examples of practical applications of CTMC models are; determining the mean time between failures in safety-critical systems and identifying bottlenecks in high speed communication networks. A labelled CTMC is defined as follows (adapted from [11]):

**Definition 2.2** (labelled CTMC)**.** A labelled CTMC is a quadruple $< S, \overline{s}, R, L >$ where:

- $S$ is a finite set of states,
- $\overline{s} \in S$ is the initial state,
- $R : S \times S \rightarrow \mathbb{R}_{\geq 0}$ is the rate matrix, where $R(s, s')$ is the rate of moving from state $s$ to $s'$,
- $L : S \rightarrow 2^{AP}$ is the labelling function, which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ that are valid in $s$.

Similar to a DTMC, the definition contains a set of states S, the labelling function L and an initial state. Instead of the probability matrix $P$ a CTMC has a rate matrix $R$, which gives rates $R(s, s')$ at which transitions occur between each pair of states $s, s'$. If $R(s, s') = 0$ then no transition from state $s$ to $s'$ is possible, because it has zero probability. Otherwise, if $R(s, s') > 0$ and state $s$ has only a single possible successor state $s'$, then $1 - e^{-R(s,s')\cdot t}$ denotes the probability of moving from state $s$ to $s'$ within $t$ time units. In the case where state $s$ has more than one successor, i. e. $R(s, s') > 0$ for more than one state $s'$, we have to deal with competition between the outgoing transitions of $s$. This can be explained as follows, suppose we have a state $s$ with multiple outgoing transitions, as soon as we enter state $s$ we start a countdown clock for each outgoing transition (depending on its specified rate). The transition of which the clock finishes first wins. This situation is known as the race condition. To account for the race condition we need to look at the total rate of the outgoing transitions of state $s$, known as the exit rate:

$$E(s) = \sum_{s' \in S} R(s, s') \tag{2.1}$$

The probability of moving from (a non terminating) state $s$ to state $s'$ within $t$ time units is specified as:

$$P(s, s', t) = \frac{R(s, s')}{E(s)} \cdot (1 - e^{-E(s)\cdot t}) \tag{2.2}$$

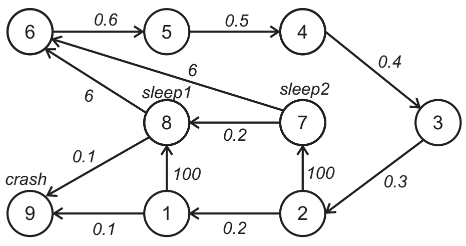
By determining the so called *embedded* DTMC of a CTMC we can look at the pure probabilistic behaviour (i. e. ignore the time spent in any state). The probability $P(s, s')$ of moving from state $s$ to $s'$ is determined by the probability that the delay of going from state $s$ to $s'$ finishes before the delays of other outgoing edges from $s$, formally:

$$P(s, s') = \begin{cases} \frac{R(s,s')}{E(s)} & \text{if } E(s) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

When $s$ is of an absorbing state, we have $E(s) = 0$ and $P(s, s') = 0$ for any state $s'$. $P$ is known as the probability matrix of the embedded DTMC of the CTMC (note that for DTMCs usually $P(s, s) = 1$ for a terminating state $s$). The transformation of a CTMC into a DTMC can be performed by means of uniformization, more details on this subject can be found in [45, 74, 49, 76].

Figure 2.4 shows a CTMC model of the Hubble space telescope, which was introduced on page 16. We have previously seen the DTMC model, introduced in Section 2.3.1, the CMTC model takes into account the life span of a gyroscope and other time dependent events. It models the real-time probabilistic behaviour of the failure and repair of the gyroscopes. The model is based on the following assumptions:

- each gyroscope has an average lifetime of 10 years,
- the average preparation time of a repair mission is two months,
- it takes about 1/100 year (circa 3.5 days) to turn the telescope into sleep-mode,
- the base time scale is one year.



(a) Transition system

$$R = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 100 & 0.1 \\ 0.2 & 0 & 0 & 0 & 0 & 0 & 100 & 0 & 0 \\ 0 & 0.3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0.1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Rate matrix

$$E = \begin{pmatrix} 100.1 \\ 100.2 \\ 0.3 \\ 0.4 \\ 0.5 \\ 0.6 \\ 6.2 \\ 6.1 \\ 0 \end{pmatrix}$$

(c) Vector E

Figure 2.4: CTMC of the Hubble space telescope

State six of the model corresponds to the system state where all six gyroscopes are operational and any one of them may fail. Since each gyroscope fails with a rate of $\frac{1}{10}$ (because the lifespan is 10 years and the base time scale is one year), the outgoing rate of state six is $6 \cdot \frac{1}{10} = 0.6$ (because any one of the six gyroscopes may fail). The relation between the CTMC and the embedded DTMC of the Hubble space telescope is given by equation 2.3. For example the probability of moving from state 2 to 7 is: $P(2,7) = \frac{R(2,7)}{E(2)} = \frac{100}{100.2} \approx 0.998$, which is consistent with the probability shown in matrix $P$ of the DTMC in figure 2.3.

## 2.4 Logics for checking probabilistic models

Once the system is represented by a model we want to check if the model satisfies a formal specification (i. e. if it has certain properties). These properties can be expressed in a formal manner using logics. These logics enable us to reason about qualitative[6] or quantitative[7] properties of probabilistic systems. This section discusses two temporal logics, namely Probabilistic Computation Tree Logic (PCTL) [33] and Continuous Stochastic Logic (CSL) [11], which are used for verification of DTMCs and CTMCs respectively.

### 2.4.1 Probabilistic Computation Tree Logic (PCTL)

Probabilistic Computation Tree Logic (PCTL) was first introduced by Hansson and Jonsson [33] as an extension of the temporal logic CTL [20] with discrete time and probabilities. PCTL allows expressing properties such as: "*the probability of reaching a certain goal $\psi$ within a specified number of steps k, via paths through a set of allowed states $\phi$, is at least/at most some probability value*". PCTL can express properties on states (state formula) or paths (path formula) in the DTMC.

**Definition 2.3** (PCTL syntax)**.** Let $p \in [0,1]$ be a real number, and $k_i \in \mathbb{N}$ and $\bowtie \in \{<,>,\leq,\geq\}$ a comparison operator. The syntax of PCTL formulas over a set of atomic propositions AP is defined inductively as follows:

- *true* is a state-formula,
- Each atomic proposition $a \in AP$ is a state formula,
- If $\phi$ and $\psi$ are state formulas, then so are $\neg\phi$ and $\phi \wedge \psi$,
- If $\phi$ and $\psi$ are state formulas, then $\mathcal{X}\ \phi$ and $\phi\ \mathcal{U}\ \psi$ and $\phi\ \mathcal{U}^{[k_1,k_2]}\ \psi$ are path formulas[8],
- If $\pi$ is a path formula, then $\mathcal{P}_{\bowtie p}(\pi)$ is a state formula.

The boolean operators $\neg$ and $\wedge$ have their usual meaning, they can be used to derive the operators $\vee$ and $\implies$. The path formulas involve the next operator $\mathcal{X}$ and the unbounded $\mathcal{U}$ or bounded $\mathcal{U}^{[k_1,k_2]}$ until. The semantics of the next and unbounded until are equivalent to that of CTL, whereas the bounded until $\phi\ \mathcal{U}^{[k_1,k_2]}\ \psi$ states that "$\psi$ is satisfied in one of the first $k'$ states, where $k' \in [k_1,k_2]$ and at all preceding states $[0,k')$ $\phi$ holds, with $k_1 \geq 0$ and $k_2 < \infty$. The state formula $\mathcal{P}_{\bowtie p}(\pi)$ asserts that the probability measure of paths satisfying $\pi$ meets the bound $\bowtie p$.

We use a satisfaction relation $\models_{\mathcal{M}}$ to define the truth of PCTL formulas, for state $s$ or path $\pi$ in a DTMC $\mathcal{M} = (S,\overline{s},P,L)$. Intuitively $s \models_{\mathcal{M}} \phi$ means that formula $\phi$ is true at state $s$ in DTMC $\mathcal{M}$, the same applies for path formulas.

**Definition 2.4** (PCTL semantics)**.** Let $p \in [0,1]$ be a real number, $k_i \in \mathbb{N}$, and $\bowtie \in \{<,>,\leq,\geq\}$ a comparison operator. Also let $\pi[i] = s_i$ be the $i$th state along the path $\pi$. The satisfaction relation $\models_{\mathcal{M}}$, where $s$ is a state, $\pi$ a path and $\mathcal{M} = (S,\overline{s},P,L)$ a DTMC, is defined as follows:

| | |
|---|---|
| $s \models_{\mathcal{M}} \mathit{true}$ | for all states |
| $s \models_{\mathcal{M}} a$ | iff $a$ is an atomic proposition valid in s, $a \in \mathit{Label}(s)$ |
| $s \models_{\mathcal{M}} \neg\phi$ | iff $s \not\models_{\mathcal{M}} \phi$ |
| $s \models_{\mathcal{M}} \phi \wedge \psi$ | iff $s \models_{\mathcal{M}} \phi \wedge s \models_{\mathcal{M}} \psi$ |
| $s \models_{\mathcal{M}} \mathcal{P}_{\bowtie p}(\Psi)$ | iff $Pr_s\{\pi \in \mathit{Path}^{\mathcal{M}}(s) \mid \pi \models_{\mathcal{M}} \Psi\} \bowtie p$ |
| $\pi \models_{\mathcal{M}} \mathcal{X}\phi$ | iff $\pi[1]$ is defined and $\pi[1] \models_{\mathcal{M}} \phi$ |
| $\pi \models_{\mathcal{M}} \phi\ \mathcal{U}^{[k_1,k_2]}\ \psi$ | iff $\exists k' \in [k_1,k_2].\ (\pi[k'] \models_{\mathcal{M}} \psi\ \wedge\ (\forall k'' \in [0,k').\pi[k''] \models_{\mathcal{M}} \phi))$ |
| $s \models_{\mathcal{M}} \mathcal{L}_{\bowtie p}(\phi)$ | iff $\lim_{k\to\infty} Pr_s\{\pi \in \mathit{Path}^{\mathcal{M}}(s) \mid \pi[k] \models_{\mathcal{M}} \phi\}$ |

Where $\Pr_s\{\pi \in \mathit{Path}^{\mathcal{M}}(s) \mid \pi \models_{\mathcal{M}} \Psi\} \bowtie p$, means that the probability measure of all paths $\pi \in \mathit{Path}$ starting in s and satisfying $\Psi$ must meet the bound $\bowtie p$.

[6] Qualitative properties assert that a certain event $\phi$ holds with probability 0 or 1 [10].
[7] Quantitative properties guarantee that the probability for a certain event $\phi$ meets given lower or upper bounds [10].
[8] If $k_1 = 0$ In $\mathcal{U}^{[k_1,k_2]}$ it is usually expressed as $\mathcal{U}^{\leq k_2}$.

With the PCTL syntax en semantics established, we can now for instance define the following property on the Hubble space telescope DTMC:

> *"The probability that the telescope eventually crashes without ever having only one operational gyroscope left is at most* $10^{-4}$*."*

which is expressed in PCTL as:

$$\mathcal{P}_{\leq 0.001}(\neg \text{"}1\text{"}\ \mathcal{U}\ \text{"}crash\text{"})$$

## 2.4.2 Continuous Stochastic Logic (CSL)

Continuous Stochastic Logic (CSL) was originally developed by Aziz et al. [9] and later extended by Baier et al. [14]. It is based on the temporal logics CTL [20] and PCTL. It provides means to express properties on CTMCs that refer to steady-state[9] and transient[10] behaviour. CSL resembles PCTL, in fact it extends PCTL, however the difference lies in the time domain. PCTL is restricted to step intervals of natural numbers $\mathbb{N}$, whereas CSL allows real numbers greater than or equal to zero $\mathbb{R}_{\geq 0}$.

**Definition 2.5** (CSL syntax, adapted from [11])**.** Let $p \in [0,1]$ be a real number, $I \subseteq \mathbb{R}_{\geq 0}$ a non-empty interval and $\bowtie \in \{<,>,\leq,\geq\}$ a comparison operator. The syntax of CSL formulas over a set of atomic propositions AP is defined as follows:

- *true* is a state-formula,
- Each atomic proposition $a \in AP$ is a state formula,
- If $\phi$ and $\psi$ are state formula, then so are $\neg\phi$ and $\phi \wedge \psi$,
- If $\phi$ is a state formula, then so is $\mathcal{S}_{\bowtie p}(\phi)$,
- If $\Psi$ is a path formula, then $\mathcal{P}_{\bowtie p}(\Psi)$ is a state formula,
- If $\phi$ and $\psi$ are state formulas, then $\mathcal{X}^I\ \phi$ and $\phi\,\mathcal{U}\ \psi$ and $\phi\,\mathcal{U}^I\ \psi$ are path formulas.

The state formulas do not differ from those used in PCTL, except for the steady-state $\mathcal{S}_{\bowtie p}(\phi)$ which corresponds to the long-run operator $\mathcal{L}_{\bowtie p}(\phi)$. It asserts that the probability of being in a $\phi$ state on the long run, meets the bound $\bowtie p$. The path formula $\mathcal{X}^I\ \phi$ asserts that a transition is made to a $\phi$ state at some time point $t \in I$. Formula $\phi\,\mathcal{U}^I\ \psi$ states that $\psi$ is satisfied at some time instant $t$, within the interval $I$ and at all preceding time instants $[0,t)$ $\phi$ holds. The unbounded until operator $\mathcal{U}$ is another notion for asserting $\phi\,\mathcal{U}^{[0,\infty]}\ \psi$. We use a satisfaction relation $\models_{\mathcal{M}}$ to define the truth of CSL formulas, for state $s$ and path $\pi$ in a CTMC $\mathcal{M} = (S, \overline{s}, R, L)$.

**Definition 2.6** (CSL semantics)**.** Let $p \in [0,1]$ be a real number, and $t \in \mathbb{R}$ and $\bowtie \in \{<,>,\leq,\geq\}$ a comparison operator. Also let $\pi[i] = s_i$ be the $i$th state along the path $\pi$. Let $\delta(\pi,i) = t_i$ be the time spent in state $s_i$, and let $\pi@t$ denote the state occupied in path $\pi$ at time $t$. (Similar definitions can be found in [14, 11]) The satisfaction relation $\models_{\mathcal{M}}$, where $s$ is a state, $\pi$ a path and $\mathcal{M}$ a CTMC, is defined by:

| | |
|---|---|
| $s \models_{\mathcal{M}} true$ | for all states, |
| $s \models_{\mathcal{M}} a$ | iff $a$ is an atomic proposition valid in s, $a \in Label(s)$, |
| $s \models_{\mathcal{M}} \neg\phi$ | iff $s \not\models_{\mathcal{M}} \phi$, |
| $s \models_{\mathcal{M}} \phi \wedge \psi$ | iff $s \models_{\mathcal{M}} \phi \wedge s \models_{\mathcal{M}} \psi$, |
| $s \models_{\mathcal{M}} \mathcal{S}_{\bowtie p}(\phi)$ | iff $\lim_{t\to\infty}\ Pr_s\{\pi \in Path^{\mathcal{M}}(s) \mid \pi@t \models_{\mathcal{M}} \phi\}$, |
| $s \models_{\mathcal{M}} \mathcal{P}_{\bowtie p}(\Psi)$ | iff $Pr_s\{\pi \in Path^{\mathcal{M}}(s) \mid \pi \models_{\mathcal{M}} \Psi\} \bowtie p$, |
| $\pi \models_{\mathcal{M}} \mathcal{X}^I\phi$ | iff $\pi[1]$ is defined and $\pi[1] \models_{\mathcal{M}} \phi\ \wedge\ \delta(\pi,0) \in I$, |
| $\pi \models_{\mathcal{M}} \phi\,\mathcal{U}^I\ \psi$ | iff $\exists t \in I.\ (\pi@t \models_{\mathcal{M}} \psi\ \wedge\ (\forall t' \in [0,t).\pi@t' \models_{\mathcal{M}} \phi))$. |

[9] Steady-state probabilities consider the system "on the long run" (i. e. when a balance has been reached).
[10] Transient probabilities consider the system at a certain time instant.

Similar to PCTL, we can use CSL to formulate properties on the Hubble telescope CTMC. For instance, since the telescope is expected to last at least 10 years, it is interesting to formulate properties such as:

*"The probability that the system will crash within the next 10 years is at most 2%."*

which is expressed in CSL as:

$\mathcal{P}_{\leq 0.02}(true\ \mathcal{U}^{\leq 10}\ \text{``}crash\text{''})$

## 2.5 Model checking Markov chains

Once a model and properties have been formulated, a model checker can verify whether of not the properties hold in the model. The verification amounts to showing that the logical expression evaluates to true when interpreted over the model. To be more formal: in order to check if state $s$ satisfies formula $\phi$ we need to recursively compute the set $Sat(\phi) = \{s \in S \mid s \models \phi\}$ of states that satisfy $\phi$ and check if $s$ is a member of that set. There are several different methods and algorithms available for model checking. The purpose of this section is to briefly cover the basics of solution methods of Markov Chains that are used by the tools discussed in this thesis.

### 2.5.1 Numerical and statistical methods

There are two primary approaches in analysing stochastic behaviour of a system: *numerical* and *statistical*. The numerical approach is divided into symbolic [57] and numerical [74] methods. Model checking tools have to deal with the fact of rapidly increasing numbers of states and transition when generating a state-space of concurrent systems.

**Symbolic** algorithms try to cope with this problem, known as the state-space explosion, by avoiding ever building the state-space as a set of nodes and transitions; instead, they represent the graph implicitly using a more compact data structure. The state-space can be represented using binary decision diagrams (see the work of Ken McMillan [57]) or more recently Multi-Terminal Binary Decision Diagram [47, 24].

**Numerical** algorithms offer a range of methods for solving a system of linear equations, which is needed for solving formulas containing the $\mathcal{U}$ operator. The advantage of using numerical methods is their high accuracy, but the drawback is that they require a large amount of memory (caused by the state-space explosion problem).

**Statistical** methods use simulation and sampling. So instead of building a complete state-space and verifying properties in each state the statistical method uses the model description to generate sample execution paths. It will then estimate if the property holds based on the generated samples. This method will obviously not provide the high level of accuracy as in numerical methods, since it is statistical in nature, but memory requirements are negligible compared to that of numerical methods. Statistical tools usually offer a choice between setting a fixed number of samples and using a method called "sequential acceptance sampling" [79, 85]. The latter method is a statistical procedure that takes observations into account as they are made. For instance when using Wald's [79] sequential probability ratio test, no predetermined number of observations is used, but instead this method determines after each observation if another observation is needed or if the information currently available is sufficient to accept a hypothesis so that the test has the prescribed strength. When the number of samples is not fixed, the tools usually allow setting a desired confidence level (also known as the error probability) i. e. the probability that the answer given by the tool is incorrect. Choosing a better confidence level (i. e. lower probability of getting wrong answers) results in more samples being taken, which in turn causes the model check time to increase. In case of a fixed number of samples, the sample collection will stop when the predetermined maximum is reached and an answer is given based on the collected samples so far.

Further details on the differences between the aforementioned methods can be found in [84].

### 2.5.2 PCTL model checking of DTMCs

In this section we present an insight into the theory of PCTL model checking. This subject has been extensively discussed in for instance [33, 10, 19]. It is known from [33] that PCTL model checking on Markov chains can be done in polynomial time in the size of the system and in linear time on the size of the formula. Given a PCTL formula $\phi$, the model checking process generally starts with building the parse tree of $\phi$, whose nodes represent subformulas of $\phi$. As stated earlier, the intent is to compute the satisfaction set $Sat(\phi)$, where $Sat(\phi) = \{s \in S \mid s \models \phi\}$. This is achieved by processing the subformulas in the parse tree in a bottum-up manner. The processing of leaf nodes, where the formula is either true or some atomic proposition, is straightforward. The same holds for boolean connectives, for instance $Sat(\Phi \wedge \psi) = Sat(\Phi) \cap Sat(\psi)$. Below we present an outline of the actions involved in model checking the different PCTL operators. When illustrating model checking algorithm complexities, we use the term $|S|$ to denote the number of states in a DTMC and $|E|$ to denote the number of transitions with non-zero probability.

**PCTL Next.** Calculations for the PCTL Next formula are somewhat trivial, they involve a single matrix-vector multiplication.

**PCTL Bounded Until.** The calculations for $\mathcal{U}^{\leq k}$ amount to performing some graph analysis and $k$ matrix-vector multiplications[11]. According to [33], the number of required arithmetical operations for bounded until formulas with a finite time bound $k$ is at most $\mathcal{O}(k_{max} \times (|S| + |E|) \times |\phi|)$ or $\mathcal{O}(|S|^3 \times |\phi|)$, depending on the algorithm, where $k_{max}$ is the maximum time parameter in a formula, and $|\phi|$ is the size of the formula.

**PCTL Unbounded Until.** The unbounded until cannot be computed by the bounded until method, since this would require infinitely many matrix-vector multiplications (recall that $\mathcal{U}$ can be denoted as $\mathcal{U}^{\leq\infty}$). The technique, however, is quite similar. The algorithm will first perform some precomputations in time $\mathcal{O}(|S| + |E|)$[19], using general graph traversal algorithms or BDD fixed point computation [24]. In case the probability bound $p$ in $\mathcal{P}_{\bowtie p}$ is either 0 or 1, no further computations are necessary. For the remaining situations (i.e. $0 < p < 1$ ), we require solving a system of linear equations by means of numerical computations. A system of linear equations can be solved in polynomial time using direct methods (e.g., Gaussian elimination or LU decomposition) or iterative methods like the Jacobi- or the Gauss-Seidel-method [74]. For large probability matrices, the iterative methods perform better. More information on algorithms for solving a system of linear equations is presented in Section 2.5.4.

**PCTL Long Run.** The $\mathcal{L}$ operator shows the behaviour of the system in the long run. Initially, a graph analysis is carried out to find all Bottom Strongly Connected Components (BSCC)[12], which takes $\mathcal{O}(|S| + |E|)$ [75] time. Then for each BSCC a system of linear equations is solved, which can be done in polynomial time. Finally, the probability of reaching each BSCC is computed, which amount to solving $\mathcal{P}_{=?}[true\ \mathcal{U}\ BSCC_i]$. The worst case time complexity for model checking the long run property is $\mathcal{O}(|S|^3)$.

[11] For large $k$, the number of multiplications might be smaller if on-the-fly steady-state detection [59, 48] is applied.
[12] A BSCC [75] is a maximal subset (i. e. subgraph) of a graph, that once entered, cannot be exited and any two vertices in a BSCC lie on a cycle.

### 2.5.3 CSL model checking of CTMCs

As for PCTL, model checking CSL [11, 9, 47] proceeds by recursively computing the satisfaction sets. For CSL formulas without a time bound, the problem reduces to probabilistic model checking of DTMCs. Baier et al. [12] demonstrated that CSL model checking of time-bounded formulas can be reduced to transient analysis (in particular uniformization [45]) of CTMCs. The CSL model checking algorithms (as presented in [11]) are polynomial in the size of the model and linear in the length of the formula.

**CSL Next.** To determine if state $s$ satisfies $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \phi]$, we require computing the set $Sat(\phi)$ and a single multiplication of the matrix P with a vector $\underline{b}$. This vector is defined as:

$$\underline{b}(s) = \begin{cases} e^{-E(s)\cdot t1} - e^{-E(s)\cdot t2} & \text{if } s \in Sat(\phi) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

**CSL Bounded Until.** Model checking the $\mathcal{U}^I$ operator involves matrix–vector multiplications and transient analysis. It requires computations on the probability matrix obtained by uniformization [45] of the CTMC. Computations for the bounded until can be performed with worst case time complicity of $\mathcal{O}(|E| \cdot q \cdot t_2)$ [11], where $q$ is the uniformization rate[13], and $t_2$ is the upper bound of interval $I$. In practice, there are some efficiency improvements possible. For instance, for large $t_2$ the number of required computations can be reduced by applying on-the-fly steady-state detection [59, 48]. Without going into detail, if desired see [11], we like to point out that there exists a difference between the number of required computations for $t_1 = 0$ and $0 < t_1 \leq t_2$ in $\mathcal{U}^{t1,t2}$, where the latter situation is computationally more intensive.

**CSL Steady-state.** Computing whether $s \models \mathcal{S}_{\bowtie p}(\phi)$ amounts to solving a system of linear equations (in polynomial time) combined with graph analysis methods, namely a search for all bottom strongly connected components (BSCC), which takes $\mathcal{O}(|S| + |E|)$ [75] time. A steady-state analysis is performed for each BSCC, after which the probabilities of reaching the individual BSCCs are computed. According to [11], steady-state analysis can be performed with a worst case time complexity of $\mathcal{O}(N^3)$.

Table 2.1 shows an overview of the worst case time complexities of algorithms for model checking CSL. These results are based on using a sparse storage structure (see Section 2.6), Gaussian elimination for solving linear equations systems and uniformization for transient analysis.

Table 2.1: From [11], Algorithms for Model Checking CSL and their (worst case) time complexity.

| **Operator** | **Algorithm(s)** | **Time complexity** |
|---|---|---|
| $\mathcal{S}_{\bowtie p}[\Phi]$ | BSCC detection + steady-state analysis per BSCC + computing probability of reaching a BSCC | $\mathcal{O}(\|S\|^3)$ |
| $\mathcal{P}_{\bowtie p}[\mathcal{X}\ \Phi]$ | matrix-vector multiplication | $\mathcal{O}(\|E\|)$ |
| $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Phi]$ | matrix-vector multiplication | $\mathcal{O}(\|E\|)$ |
| $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | solving linear equation system | $\mathcal{O}(\|S\|^3)$ |
| $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | matrix-vector multiplication + transient analysis | $\mathcal{O}(\|E\| \cdot q \cdot t_2)$ |

With:

| | | |
|---|---|---|
| $\|S\|$ | = | number of states in CTMC $\mathcal{M}$ |
| $\|E\|$ | = | the number of transitions with non-zero probability |
| $q$ | = | the uniformization rate (usually maximum entry of $\underline{E}$) |

[13] The uniformization rate is greater of equal to the maximum entry of $\underline{E}$ (see equation 2.1).

### 2.5.4 Solving a system of linear equations

The previous sections have shown that model checking certain formulas, such as $\mathcal{S}_{\bowtie p}(\phi)$, involves solving a system of linear equations. These equations take the form of $Ax = b$. In general there are two methods to solve a system of linear equations:

- direct methods
- iterative methods

#### Direct methods

Direct methods, such as Gaussian elimination [74], compute the solution of a system of linear equations in a fixed number of operations. Direct methods are only recommend when dealing with relative small models (not exceeding the order of thousands of states). The computational effort is in the order of $N^3$, where $N$ is the number of states. Although they are highly reliable, the disadvantage of methods such as Gaussian elimination is that during computation the coefficient matrix must be updated at each step of the algorithm. Because the elements in the matrix are constantly updated (in particular the zero elements) it is difficult to organize a compact storage scheme, resulting in high memory consumption. For this reason most model checker tools apply iterative methods.

#### Iterative methods

Iterative methods do not alter the form of the matrix and thus allow the use of compact storage schemes. For iterative methods it is not known in advance how many computations are required to achieve an accurate answer. They will perform a series of matrix-vector multiplications until the difference between iterations is less than some value $\epsilon$. This is explained by means of an example of the Power method. The example is an abridged version taken from [74]. Suppose we have a discrete-time Markov chain $\mathcal{M}$ with probability matrix

$$P = \begin{pmatrix} .0 & .8 & .2 \\ .0 & .1 & .9 \\ .6 & .0 & .4 \end{pmatrix}$$

If the system starts in state 1, the initial probability vector is given by $\underline{\alpha}^{(0)} = (1,\ 0,\ 0)$. Immediately after the first transition, the system will be either in state 2, with probability .8 or in state 3, with probability .2. The vector denoting the probability distribution after one step is thus $\underline{\alpha}^{(1)} = (0,\ .8,\ .2)$. This result may be obtained by the matrix-vector multiplication $\underline{\alpha}^{(0)}P$. Likewise we can obtain the probability distribution after two steps:

$$\underline{\alpha}^{(2)} = \underline{\alpha}^{(1)}P = (0,\ .8,\ .2) \begin{pmatrix} .0 & .8 & .2 \\ .0 & .1 & .9 \\ .6 & .0 & .4 \end{pmatrix} = (.12,\ .08,\ .8)$$

Thus, for any integer $k$, the state of the system after $k$ transitions is obtained by

$$\underline{\alpha}^{(k)} = \underline{\alpha}^{(0)}P^k$$

When the Markov chain is finite, aperiodic[14] and irreducible[15], also known as ergodic[16], (as in the current example), the vectors $\underline{\alpha}^{(k)}$ converge to the stationary probability vector $\underline{\alpha}$ regardless of the choice of the initial vector. We have

$$\lim_{k\to\infty} \underline{\alpha}^{(k)} = \underline{\alpha}$$

[14] A state $s$ is periodic with period $j$, if on leaving state $s$ a return is possible only in a number of transitions that is a multiple of the integer $j > 1$. A state whose period is 1 is said to be aperiodic. A Markov chain is aperiodic if its states are aperiodic.

[15] A Markov chain is irreducible if every state can be reached from every other state.

[16] An ergodic Markov chain is aperiodic, irreducible, and recurrent nonnull, where a Markov chain with a finite number of states has only transient and recurrent nonnull states.

After a certain (beforehand unknown) number of iterations, an equilibrium will have been reached, meaning we will no longer observe a noticeable difference in the results of the multiplications. The difference between the results of the $k^{th}$ and $(k-1)^{th}$ iteration is denoted $\epsilon$. We usually stop when $\epsilon$ is in the order of $10^{-6}$. The answer remains an approximation, where the accuracy depends on the desired $\epsilon$ value. The Power method, as described in the example, can be used to obtain the solution for so called eigenproblems ($\alpha P = \alpha$). This method is guaranteed to converge in theory, but it is often extremely slow. There are several other iterative methods, such as Jacobi, Gauss-Seidel, JOR, and SOR, which are used to obtain the solution of a system of linear equations, such as ($\alpha Q = 0$). Where $Q$ is the matrix of transition rates, known as infinitesimal generator matrix [49, 74, 11]. The Jacobi and Gauss-Seidel[17] methods are very similar to the Power method. The difference is that they use a different iteration matrix. Both methods do not have guaranteed convergence. In practice, the Jacobi method is usually faster than the Power method and Gauss-Seidel typically converges faster and consumes less memory than Jacobi. It is possible to improve the rate of convergence of both Jacobi and Gauss-Seidel by applying a technique called relaxation; this yields in respectively the JOR (Jacobi Over-Relaxation) and SOR (Successive Over-Relaxation) methods. These methods use a relaxation parameter $\omega$, which (when chosen correctly) can considerably improve the convergence rate over that of Jacobi and Gauss-Seidel. The problem, however, is that the optimal value of $\omega$ depends on the problem being solved and may vary as the iteration process converges, for details see [87]. Additional information regarding the theory behind aforementioned methods can be found in [74], for complexity see [17].

## 2.6 State-space representation: Explicit and Symbolic

The definitions of Markov chains in Section 2.3 have shown that a model representation consists of states, transitions and probabilities or rates. These representations tend to grow extremely large due to the state-space explosion problem. This is caused by the fact that a system is usually composed of a number of concurrent sub-systems. Interleaving these sub-systems to form the overall state-space results in a state-space size that is often exponential in the number of sub-systems. Systems containing several million states and transitions are not out of the ordinary. This is why it is important to use data structures that minimise the computational space and time requirements for model checking large systems. There are two well-known methods applied in state-space storage, namely *explicit* and *symbolic*.

**Explicit.** In explicit state-space representation each state and transition is individually stored using data structures such as a sparse matrix [65]. A sparse matrix is a matrix (i. e. array-based data structure), usually very large, where most of the elements are zero. This method of state-space representation focuses on preventing the storage of, and computation on, a large number of zeros. The benefits of this method are that manipulations are relatively easy (e. g. uniformization), and it frequently provides faster solutions, the drawback is the relative high memory consumption, compared to symbolic methods.

**Symbolic.** The concept behind symbolic state-space representation is the exploitation of regularity and structure in models. Instead of single state representations it uses sets of states. This results in a highly compressed representation of the state-space, provided that the Markov chain exhibits a certain degree of structure and regularity. An example of a symbolic data structure is the Multi-Terminal Binary Decision Diagram (MTBDD) [30]. An MTBDD is a data structure that represents a function mapping of Boolean variables to real numbers. It can be seen as a rooted, directed acyclic graph containing decision nodes, and terminal nodes with real numbers. For more information on symbolic model checking see [57, 39, 47, 63, 24].

[17] Gauss-Seidel can be performed *forward* and *backward*. Forward is generally recommended when most of the nonzero mass lies below the diagonal of the iterative matrix and vice versa for backward [74].

# Chapter 3

# Probabilistic model checker tools

This chapter gives details on each of the probabilistic model checker tools we used, namely; PRISM, ETMCC, MRMC, YMER and VESTA. Each tool section starts with some general background information followed by:

- implementation details
- model and specification language
- properties
- algorithms and data structures

The section that covers properties uses a *basic* formula set, this set is defined as:

**Definition 3.1** (basic set)**.** The *basic* set of formulas, where $\phi$ denotes a *state* formula and $\pi$ a *path* formula, is defined by the grammar:

$$\phi ::= true \mid false \mid \phi \wedge \psi \mid \phi \vee \psi \mid \neg\phi$$
$$\pi ::= \mathcal{X}\phi$$

Additionally, we use the letters $k_i \in \mathbb{N}$ and $t_i \in \mathbb{R}$, with $i \in \{\emptyset, 1, 2\}$ to denote a time bound in respectively PCTL and CSL formulas.

## 3.1 PRISM

PRISM [50] stands for Probabilistic Symbolic Model Checker. The tool is being developed at the University of Birmingham (United Kingdom) for the analysis of probabilistic systems. It is a free and open source tool, distributed under the GNU General Public License (GPL). The information on PRISM in this thesis is based on version 2.1, first released September 8, 2004. It is available at: `http://www.cs.bham.ac.uk/~dxp/prism/`

**Implementation details.** The tool is developed using a combination of Java and C++. Its user interface and parsers are written in Java. The core algorithms are mostly implemented in C++. For state-space representation, PRISM uses a modified version of the CUDD (CU Decision Diagram) package [73], a BDD/MTBDD library developed by Fabio Somenzi at Colorado University written in C. This package is included in the standard distribution. The tool requires Java 2, Standard Edition (J2SE) version 1.4 or higher and operates on Linux, Windows and Solaris systems (support for Mac OS X is under development). PRISM offers two user interface types: command line and Graphical User Interface (GUI). The GUI offers a text editor, property editor and plot capability. Both interfaces use the same underlying model checker. The tool is also capable of performing automated experiments, which allow to specify a range of values for constants in the model and/or property.

**Model and specification language.** System models are described using the PRISM programming language, which is a high-level state-based description language. It is based on the Reactive Modules formalism of Alur and Henzinger [7]. In this language a system is described as the parallel composition of a set of *modules*. A module state is determined by a set of finite-range variables and its behaviour is given using a guarded-command based notation. Communication between modules takes place either via global variables or synchronisation over common action labels [51]. Besides its own model description language, PRISM supports a subset of PEPA [40], which is a stochastic process algebra. PRISM also provides indirect support (either via Digital Clocks [53] or KRONOS [23]) for model checking probabilistic timed automata which include probability, non-determinism and real-time using clocks. PRISM is able to export models in many different formats, including the ones accepted by ETMCC and MRMC. The PRISM model description is translated (by the tool) into one of the following three supported probabilistic models:

- DTMC
- MDP
- CTMC

**Properties.** Properties can be specified using PCTL (for DMTCs and MDPs) or CSL (for CTMCs). In PRISM it is possible to either determine if a probability satisfies a given bound or obtain the actual value. There is also support for the specification and analysis of properties based on costs and rewards, but the implementation of this feature is only partially completed (in version 2.1) and is still ongoing. The subset of PCTL and CSL formulas supported by PRISM is displayed in Table 3.1.

Table 3.1: PRISM supported logic subset

| **Logic** | | **Supported** |
|---|---|---|
| PCTL | Basic | ✓ |
| | $\mathcal{L}_{\bowtie p}[\Phi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq k}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[k1,k2]}\ \Psi]$ | |
| CSL | Basic | ✓ |
| | $\mathcal{S}_{\bowtie p}[\Phi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq t}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\geq t}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Psi]$ | |

$\mathcal{P}_{\bowtie p}$ *with* $\bowtie \in \{<,>,\leq,\geq\}, p \in [0,1]$

**Algorithms and data structures.** PRISM offers a choice between several different algorithms for verifying PCTL and CSL properties. For PCTL, PRISM implements the algorithms of [33, 16, 15, 10]. For CSL, methods based on [47] and [12] are used. The iterative methods supported by

PRISM are listed below:

- Gauss-Seidel (also backwards)
- Jacobi
- JOR (also backwards)
- Power
- SOR

The iterative methods are used for solving a system of linear equations, needed for model checking the steady-state $\mathcal{S}$ and unbounded until $\mathcal{U}$ operators. For the time-bounded until $\mathcal{U}^{[t_1,t_2]}$ operator PRISM uses uniformization [45] and the techniques of Fox and Glynn [28].

PRISM offers the user a choice between any of the following three data structures for model checking:

1. MTBDD (Multi-Terminal Binary Decision Diagram) for model construction and BDD for reachability, more information on MTBBD/BDD's can be found in [30, 24].
2. Sparse matrix [65, 64].
3. Hybrid, a combination of MTBDD and sparse matrix.

All engines perform the same calculations; therefore the choice between "MTBDD", "Sparse" or "Hybrid" will not affect the results of the model checking. However, according to [52], the time and space performance may differ considerably. Typically the sparse engine is quicker than its MTBDD counterpart, but requires more memory. The hybrid engine aims to provide a compromise, providing faster computation than pure MTBDDs but using less memory than sparse matrices. By default, PRISM uses the hybrid engine.

## 3.2 ETMCC

$E \vdash MC^2$ (written ETMCC) is developed by the Stochastic Modelling and Verification group at the University of Erlangen-Nürnberg, Germany, and the Formal Methods & Tools group at the University of Twente, the Netherlands. It is a prototype implementation of a model checker for continuous-time Markov chains. The tool is free for non-profit organizations. The user has to fill in a license agreement form before downloading and using the tool. This thesis discusses version 1.4.2, which is available at: `http://www7.informatik.uni-erlangen.de/etmcc/`

**Implementation details.** The tool is developed in Java, it requires Java version 1.2 or above and operates on Linux, Windows and Solaris systems. ETMCC has no command line interface, all interaction must be performed using a Graphical User Interface (GUI). The GUI offers an editor for constructing properties and is used for loading all necessary input files and displaying the verification results. For our measurements, we added ad-hoc command line support to ETMCC.

**Model and specification language.** ETMCC supports CTMC models only[1]. It accepts model descriptions in the *tra-format* as generated by the stochastic process algebra tool TIPPtool [35] and Petri net tool DaNAMiCS [18]. An example of the *tra-format* is shown in Figure 3.1.

```
send 1 2 1.7 M
receive 1 3 1 M
acknowledge 3 11 1 M
...
```

Figure 3.1: *tra-format* example

[1] There is a trick to let ETMCC handle DTMCs, namely: let all rates be between 0 and 1 and the rates sum up to 1, as in a transition probability matrix. Then, the embedded DTMC is identical to this CTMC, and ETMCC can verify unbounded until properties. We will not utilize this trick in our comparative research.

Each line specifies one transition consisting of an action name, the source state, the target state, the transition rate and the type, which is set to M (Markovian). Note that the *tra-format* also supports transition types P (probabilistic) and I (immediate), but these are not supported by ETMCC. ETMCC also accepts model description in a format called *CSLstandard*, which is a derivative of the *tra-format*, where actions and the type of transition are omitted. The state labelling with atomic propositions must be provided in a *.lab* file.

**Properties.** ETMCC supports two types of logics, namely CSL and action-based CSL (aCSL). The subset of CSL formulas supported by ETMCC is displayed in Table 3.2. Similar to CSL, aCSL provides a means to reason about CTMCs, but opposed to CSL, it is not state-oriented. Its basic constructors are sets of actions, instead of atomic state propositions, for details see [37].

Table 3.2: ETMCC supported logic subset

| **Logic** | | **Supported** |
|---|---|---|
| CSL | Basic | ✓ |
| | $\mathcal{S}_{\bowtie p}[\Phi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq t}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\geq t}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Psi]$ | |

$\mathcal{P}_{\bowtie p}$ *with* $\bowtie\in \{<, >, \leq, \geq\}$, $p \in [0, 1]$

**Algorithms and data structures.** The model checking algorithms used by ETMMC are described in [36, 38]. The tool offers a choice between Gauss-Seidel and Jacobi as the iterative methods used in solving steady-state and unbounded until properties. The methods for bounded until are either transient analysis (recommended) or Volterra integrals for [0;t] (less accurate, slow). ETMCC uses an explicit (i. e. not symbolic) data structure, namely a sparse matrix.

## 3.3 MRMC

MRMC is a model checker for discrete-time and continuous-time Markov reward models. It is distributed under the GNU General Public License (GPL). The MRMC tool is developed by the Software Modelling and Verification (MOVES) group at the RWTH Aachen University (Germany) and Formal Methods & Tools group at the University of Twente (the Netherlands). The version used is 1.1.1b. It is available at: `http://www.cs.utwente.nl/~zapreevis/mrmc/`

**Implementation details.** The developers of MRMC have used ETMCC as inspiration to build a new probabilistic model checker. Therefore, MRMC can be seen as ETMCC's successor. MRMC is a command-line-based tool, which is implemented in the C language and currently supports the Linux platform only.

**Model and specification language.** MRMC supports four types of input models:

- DTMC
- DMRM[2]
- CTMC
- CMRM[3]

The input models are described in the same format as for ETMMC (see Section 3.2), except for the syntax related to aCSL which is not supported by MRMC. The probability (for DTMC) or rate matrix (for CTMC) is defined by a *.tra* file and the state labelling by a *.lab* file. These files contain straightforward text-based descriptions of the models. When dealing with specification and analysis of properties based on rewards, the state reward structure can be specified in a *.rew* file and the impulse reward structure in a *.rewi* file.

**Properties.** Properties can be specified using one of the following logics:

- PCTL
- PRCTL[4]
- CSL
- CSRL[5]

The subset of PCTL and CSL formulas supported by MRMC is displayed in Table 3.3.

Table 3.3: MRMC supported logic subset

| Logic | | Supported |
|---|---|---|
| PCTL | Basic | ✓ |
| | $\mathcal{L}_{\bowtie p}[\Phi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq k}\ \Psi]^{a}$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[k1,k2]}\ \Psi]$ | ✓ |
| CSL | Basic | ✓ |
| | $\mathcal{S}_{\bowtie p}[\Phi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq t}\ \Psi]^{a}$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\geq t}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t}\ \Psi]^{a}$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Psi]$ | ✓ |

$\mathcal{P}_{\bowtie p}$ *with* $\bowtie \in \{<,>,\leq,\geq\}$, $p \in [0,1]$

[a] mrmc syntax requires $\leq x$ bound to be specified as [0,$x$]

[2] DMRM stands for Discrete Markov Reward Model, which is a DTMC with reward extension [8] .
[3] CMRM stands for Continuous Markov Reward Model, which is a CTMC with reward extension [13, 34].
[4] PRCTL, Probabilistic Reward Computation Tree Logic [8] extends PCTL with reward-bounded properties.
[5] CSRL, Continuous Stochastic Reward Logic [13, 34] extends CSL with reward-bounded properties.

**Algorithms and data structures.** For checking PCTL, MRMC uses the algorithms described in [33], for PRCTL see [8] and for CSL see [11]. MRMC supports two algorithms for time- and reward-bounded until-formulas (CSRL). One is based on discretization [77], the other on uniformization and path truncation [67]. This includes state and impulse rewards. In combination with uniformization it used the techniques of Fox and Glynn [28]. The supported iterative methods are listed below:

- Jacobi
- Gauss-Seidel

The state-space is represented using sparse matrix with the compressed row, compressed column technique [68] (also called *Harwell-Boeing* sparse matrix format).

## 3.4 YMER

YMER(3.0) is a tool for verifying transient properties of stochastic systems. It supports Continuous-Time Markov Chains (CTMCs) and Generalized Semi-Markov Processes (GSMPs)[6]. YMER implements statistical model checking techniques, based on discrete event simulation and acceptance sampling, for CSL model checking [83]. YMER also supports numerical techniques for CTMCs model checking. The tool is developed by Håkan Younes at Carnegie Mellon University (Pittsburgh). The current version is 3.0 (released February 1, 2005), which is distributed under the GNU General Public Licence (GPL). The tool is available at: http://www.cs.cmu.edu/~lorens

**Implementation details.** YMER is a command-line-based tool. It uses a random number generator implemented in C, whereas other parts of the tool are written in C++. YMER uses the CUDD[7] package for symbolic data structures, this package is not included in the distribution and should be installed separately. The numerical engine for model checking CTMCs is adopted from the hybrid model checker engine of the PRISM tool. The tool operates on the Linux platform, support for other platforms is not specified.

**Model and specification language.** YMER supports two types of input models:

- CTMC
- GSMP[80]

The language used for model specification is a subset of the PRISM language[8]. Because the language is a subset, it means not all operations on for instance rate values are supported[9].

**Properties.** Properties can be specified using CSL. The subset of CSL formulas supported by YMER is displayed in Table 3.4.

**Algorithms and data structures.** For CSL model checking YMER implements the statistical model checking techniques proposed by Younes and Simmons in [85]. YMER uses discrete event simulation [71] to generate sample execution paths and verifies a given CSL path formula over each sample path. The tool offers a choice between either taking a fixed number of samples or sequential acceptance sampling. There is also support for distributed acceptance sampling, meaning multiple machines can be used to generate samples independently. It uses a master/slave architecture to collect and process samples. The interested reader is referred to [83] for more details on this architecture. Understandably, the data structures for state-space representation

[6] Details on GSMPs can be found in [80].
[7] The CUDD package can be obtained from http://vlsi.colorado.edu/~fabio/CUDD.
[8] The BNF grammar for YMER is listed in [82].
[9] In YMER it is not possible to add or subtract rate values in the model description, only multiplication and division is supported.

Table 3.4: YMER supported logic subset

| Logic | | Supported |
|---|---|---|
| CSL | Basic | ✓ |
| | $\mathcal{S}_{\bowtie p}[\Phi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq t}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\geq t}\ \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t}\ \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Psi]$ | ✓ |

$\mathcal{P}_{\bowtie p}$ *with* $\bowtie \in \{<, >, \leq, \geq\}$, $p \in [0, 1]$

are not applicable for the statistical part of this tool. It has no need for data structures that can store the complete state-space, as is the case in numerical tools. However, the collected samples and intermediate results do need to be stored, but the required size of the samples is negligible compared to a fully generated state-space. The data structures used by YMER are not documented, except for the numerical part of the tool, which uses the sparse/MTBDD hybrid engine from the PRISM tool to represent the state-space.

## 3.5 VESTA

VESTA (2.0) is a tool for statistical analysis of probabilistic systems. It supports statistical model-checking and statistical evaluation of expected values of temporal expressions. Like YMER, the tool can only provide probabilistic guaranties of correctness. The current release is version 2.0 (September 2005). It is developed at the University of Illinois at Urbana-Champaign and can be obtained from `http://osl.cs.uiuc.edu/~ksen/vesta2/`.

**Implementation details.** VESTA is written in Java and requires runtime environment version 1.5 or above. Although the documentation gives no specifics regarding platform support, it has been proven to operate on Linux and Windows. The tool has command-line support and a GUI that offers basic text editing and model check/save/load capability.

**Model and specification language.** VESTA supports two types of input models:

- DTMC
- CTMC

The tool uses a Java-based language to specify models. The model description consists of sequential statements in combination with Java code, such as *for* or *if-then-else* constructions. Each sequential statement consists of a guard, rate and action, all of which can use variables (e. g. *int*, *float* or *arrays*). The language offers no parallel composition. In addition to the Java based language there is support for PMaude [5], an executable algebraic specification language which allows to describe models in probabilistic rewrite theories. It uses a query language called Quantative Temporal Expressions (QuaTEx)[5] to express various quantitative properties of the probabilistic system.

Besides the two modelling languages described above, VESTA allows to plug in any probabilistic modelling language. In order to use the language, it is necessary to provide a discrete event simulator by implementing a predefined Java interface, for details see [4].

**Properties.** Properties can be specified using one of the following logics:

- PCTL
- CSL
- QuaTEx

The subset of PCTL and CSL formulas supported by VESTA is displayed in Table 3.5, for QuaTEx see [5].

Table 3.5: VESTA supported logic subset

| **Logic** | | **Supported** |
|---|---|---|
| PCTL | Basic | ✓ |
| | $\mathcal{L}_{\bowtie p}[\Phi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U} \; \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U}^{\leq k} \; \Psi]^a$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U}^{[k1,k2]} \; \Psi]$ | |
| CSL | Basic | ✓ |
| | $\mathcal{S}_{\bowtie p}[\Phi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U} \; \Psi]$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U}^{\leq t} \; \Psi]^a$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U}^{\geq t} \; \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\Phi \; \mathcal{U}^{[t1,t2]} \; \Psi]$ | |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t} \; \Psi]^a$ | ✓ |
| | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]} \; \Psi]$ | |

$\mathcal{P}_{\bowtie p}$ *with* $\bowtie \in \{\leq, \geq\}$, $p \in [0,1]$

[a] The $\leq$ is not supported as time bound, only $<$.

**Algorithms and data structures.** VESTA uses the statistical methods proposed in [85] and extended in [69], which are based on Monte-Carlo simulation of the model and performing statistical hypothesis testing [42] on the samples generated. The VESTA development team has made some modifications to the methods proposed in [85, 69], these modifications and algorithms are discussed in [70]. One example modification is the addition for support of the unbounded until operator. VESTA (being a statistical tool) has no need for storing a complete state-space, therefore information on data structures is not available.

## 3.6 Summary

In the previous sections we have seen the details on each of the probabilistic model checker tools we used, namely; PRISM, ETMCC, MRMC, YMER and VESTA. The upcoming tables show an overview of the tools background 3.6, platform support 3.7, supported models 3.8, data structures 3.9, logical operators 3.10 and the implemented algorithms 3.11.

Table 3.6: Tool background

| **Tool** | **University** | **Type** | *programming language* | | | *user interface* | |
|---|---|---|---|---|---|---|---|
| | | | **Java** | **C++** | **C** | **GUI** | **cmdl.**[a] |
| PRISM | Birmingham | numerical | ✓ | ✓ | ✓ | ✓ | ✓ |
| ETMCC | Erlangen-Nürnberg and Twente | numerical | ✓ | | | ✓ | |
| MRMC | Aachen and Twente | numerical | | | ✓ | | ✓ |
| YMER | Carnegie Mellon (Pittsburgh) | statistical | | ✓ | ✓ | | ✓ |
| VESTA | Urbana-Champaign, Illinois | statistical | ✓ | | | ✓ | ✓ |

[a] command-line

Table 3.7: Platform support

| **Tool** | *platform* | | | |
|---|---|---|---|---|
| | **Linux** | **Windows** | **Solaris** | **Mac OS X** |
| PRISM | ✓ | ✓ | ✓ | ✓[a] |
| ETMCC | ✓ | ✓ | ✓ | |
| MRMC | ✓ | | | |
| YMER | ✓ | | | |
| VESTA | ✓ | ✓ | ? | |

[a] under development

Table 3.8: Model support

| **Tool** | *supported models* | | | | | |
|---|---|---|---|---|---|---|
| | **DTMC** | **CTMC** | **MDP** | **DMRM** | **CMRM** | **GSMP** |
| PRISM | ✓ | ✓ | ✓ | | | |
| ETMCC | | ✓ | | | | |
| MRMC | ✓ | ✓ | | ✓ | ✓ | |
| YMER | | ✓ | | | | ✓ |
| VESTA | ✓ | ✓ | | | | |

Table 3.9: Data structures

| **Tool** | *data structures* | | |
|---|---|---|---|
| | **Sparse** | **MTBDD** | **Sparse/MTBDD hybrid** |
| PRISM | ✓ | ✓ | ✓ |
| ETMCC | ✓ | | |
| MRMC | ✓ | | |
| YMER | - | - | ✓ |
| VESTA | - | - | - |

Table 3.10: Logical formulas supported by the tools

(a) Probabilistic Computation Tree Logic

| **Tool** | *PCTL* | | | | |
|---|---|---|---|---|---|
| | **basic** | $\mathcal{L}_{\bowtie p}[\Phi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq k}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[k1,k2]}\ \Psi]$ |
| PRISM | ✓ | | ✓ | ✓ | |
| ETMCC | | | | | |
| MRMC | ✓ | ✓ | ✓ | ✓ | ✓ |
| YMER | | | | | |
| VESTA | ✓ | | ✓ | ✓ | |

$\bowtie \in \{<, >, \leq, \geq\}$,
$p \in [0, 1]$,
$k, k_1, k_2 \in \mathbb{N}$
For the **basic** set, see Definition 3.1

(b) Continuous Stochastic Logic

| **Tool** | *CSL* | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **basic** | $\mathcal{S}_{\bowtie p}[\Phi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\leq t}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{\geq t}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\Phi\ \mathcal{U}^{[t1,t2]}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{\leq t}\ \Psi]$ | $\mathcal{P}_{\bowtie p}[\mathcal{X}^{[t1,t2]}\ \Psi]$ |
| PRISM | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| ETMCC | ✓ | ✓ | ✓ | ✓ | | | | |
| MRMC | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| YMER | ✓ | | | ✓ | | ✓ | ✓ | ✓ |
| VESTA | ✓ | | ✓ | ✓ | | | ✓ | |

$\bowtie \in \{<, >, \leq, \geq\}$,
$p \in [0, 1]$,
$t, t_1, t_2 \in \mathbb{R}$

Table 3.11: Algorithms

| **Tool** | **Logic** | **Algorithm reference** | | **Solution method** |
|---|---|---|---|---|
| PRISM | CSL | [12, 47] | *iterative* | Gauss-Seidel |
| | | | | Jacobi |
| | | | | Jor |
| | | | | Power |
| | | | | Sor |
| | | | $\mathcal{U}^{[t_1,t_2]}$ | Uniformization & Fox-Glynn |
| | PCTL | [33, 16, 15, 10] | | see iterative methods CSL |
| ETMCC | CSL[a] | [36, 38] | *iterative* | Gauss-Seidel |
| | | | | Jacobi |
| | | | $\mathcal{U}^{[t_1,t_2]}$ | Transient analysis |
| | | | | Volterra integrals |
| MRMC | CSL | [11] | *iterative* | Gauss-Seidel |
| | | | | Jacobi |
| | | | $\mathcal{U}^{[t_1,t_2]}$ | Uniformization & Fox-Glynn |
| | PCTL | [33] | | see iterative methods CSL |
| | CSRL | [77, 67] | | Discretization 'Tijms Veldman' |
| | | | | Uniformization 'Bruno Sericola' |
| | | | | Uniformization 'Qureshi and Sanders' |
| | PRCTL | [8] | | - |
| YMER | CSL | [85] | | Discrete event simulation |
| VESTA | CSL<br>PCTL | [85, 69, 70] | | Discrete event simulation |

[a] ETMCC also supports the logic aCSL [37].

# Chapter 4

# Comparing tool efficiency

This chapter is about collecting, analysing and interpreting data related to the efficiency of the tools. We perform a series of experiments based on five existing case studies, namely:

- Synchronous Leader Election (SLE)
- Randomized Dining Philosophers (RDP)
- Birth-death process (BDP)
- Tandem Queuing Network (TQN)
- Cyclic Server Polling System (CSP)

We measure efficiency by recording the *model check time* (i. e. the time it takes a tool to verify a specific PCTL/CSL property) and the *peak memory usage* (i. e. the maximum memory consumed by the tool during its execution) of the tools during verification of the case studies listed above. Besides efficiency, we also record and compare the verification results (i. e. the answers produced by the tools).

Section 4.1 starts by explaining the conditions under which the experiments are conducted. The construction of the models and the case study descriptions are addressed in Section 4.2 and 4.3. The latter section also presents the performance results of the tools for each particular PCTL/CSL property that is verified. An analysis is given in Section 4.4, followed by an assessment of the user friendliness of each tool in Section 4.5.

## 4.1 Experiment setup

Our experiments were set up to meet the following three criteria:

1. *Repeatable/Verifiable.*
   Everyone must be able to repeat any experiment and verify the results. This is achieved by providing detailed documentation and by using e. g. open source models, open tools and standard equipment. Furthermore, the experiment process is automated (by means of scripts). The models, properties, script and tool settings are publicly available at [62].
2. *Statistically significant.*
   Make sure that observations are not due to mere chance. This is achieved by gathering a sufficient amount of data; we repeat the same experiment multiple times and compute the standard deviation. In addition, we use methods available from the field of statistics, namely hypothesis testing [78].
3. *Encapsulated.*
   Encapsulation mean that the experiments should be performed in an environment that is free from (unwanted) external influences. It is essential that the environment remains unchanged, except for the independent variables[1]. We must also ensure that our comparison is fair

[1] The independent variable is the variable selected and manipulated by the experimenter (e. g. model size) to observe its effect on the dependent variable (i. e. the observed/measured variable e. g. model check time) [31].

and that we do not attempt to compare the incomparable (for example we made sure the verification parameters, such as the error bound $\epsilon$, of each tool were set to corresponding levels.) To ensure encapsulation, we use a dedicated machine for our experiments.

**Hardware and software settings.** We will now discuss the conditions under which the experiments are conducted, such as hardware and software settings. All experiments are performed on a standard Personal Computer. The operating system is Linux, because this is supported by all tools (see Table 3.7). The specification of the test-environment is listed in Table 4.1. The tools

Table 4.1: Test environment

| | | |
|---|---|---|
| *Hardware* | Memory | 2GB DDR 400MHz |
| | CPU | Intel® Pentium® 4 CPU 3.00GHz DUAL CORE |
| *Software* | Operating system | SuSE Linux 9.1 (kernel 2.6.5-7.202.7-smp) |
| | Java | version 1.5.0_06 |
| | gcc | GNU project C and C++ compiler (version 3.3.3) |
| | time | UNIX command to measure elapsed time (version GNU 1.7) |
| | ps | UNIX command to report process status (version 3.2.1) |

are installed on the local hard disk to prevent any anomalies in performance measurements due to network traffic issues. Furthermore we ensure that the verification parameters of each of the tools match. For example, the solution method for solving a system of linear equations is set to Jacobi, this is usually the default setting, and the error bound $\epsilon$ for all non-statistical tools is set to $10^{-6}$. For statistical tools we bind the probability of error (i. e. the chance of false positive or negative) by $\alpha = \beta = 0.01$. Section 3.1 showed that PRISM offers a choice between several engines that use different data structures. We will use both the “sparse” and “hybrid” engine to perform our experiments, henceforth denoted as PRISM$^S$ and PRISM$^H$. It is expected that the PRISM$^S$ engine will show a faster performance, whereas the PRISM$^H$ engine will consume less memory. As shown in Table 4.1, our system contains 2GB of RAM memory. If any of the tools consume more than 2GB during model checking, we know that memory swapping will cause a slow down in performance. For further details regarding tool settings we refer to Appendix A.

**Gathering data.** Each experiment generates a certain amount of data. The main goal is to compare the tools on their model checking time and memory usage. However other data such as total elapsed time is also relevant. For instance, when a tool is able to check a certain property on a model very fast, but requires a long model construction time, then total time performance is poor. Also a tool that checks a model very fast but gives a wrong answer is worthless, for this we need to compare the results given by the tools. Results consist of a true/false value and usually a probability vector. We record two situations, namely:

1. The property holds in the initial state of the model.
2. The property holds in all states of the model.

Besides the obvious information, such as the tool name, model type, property, and the result, we record the metrics presented in Table 4.2.

Information regarding the number of states and transitions of a model are displayed by each tool at the start of the model checking procedure. Naturally each tool should arrive at the same amount of states and transitions for identical models, despite the fact that the model description language may differ. An important fact to keep in mind is that, unlike numerical tools, statistical tools (like YMER and VESTA) compute the validity of a property only in the initial state of the model. Hence, they will outperform the numerical tools.

Table 4.2: Recorded information (metrics) for each experiment

| | | |
|---|---|---|
| *model size* | *states* | number of states of the model |
| | *transitions* | number of transitions of the model |
| *performance* | *states sampled* | number of sampled states (in case of statistical model checking |
| | *construction time* | time it took to construct the model (sec.) |
| | *model check time* | time it took to model check the given property (sec.) |
| | *total time* | time between tool invocation and termination (sec.) |
| | *memory* | memory consumption (kilobytes) |

**Timing.** The time measurements related to "construction" and "checking" are reported in the output generated by the tools. The total time that the tool operates is measured by means of the GNU version of the UNIX shell command *time*[2] Using the *time* command we store the elapsed time between tool invocation and termination. Of all measured time values, we are mainly interested in the "model check time", which is the time it took to model check a given property, or to be more precise; the time the tool spend on *computing* the answer. The "model construction time" is of less interest, because one would often construct the model only once and then use it to verify multiple properties. Moreover, since not all of the tools deliver information on their model construction time, it is hard to compare this aspect among the different tools. Nonetheless, the information is recorded to make future analysis possible, if so desired. The same applies to the "total time" value, which includes the time it takes a tool to, for instance, read and write the input and output from and to a file[3]. This value is recorded, but analysis is left for future work.

**Memory usage.** Information related to memory usage is gathered by using a separate program that runs in parallel with the model checker during the experiment. This program is a shell script that, every 0.1 seconds, takes a snapshot of the memory consumption of a specified process. We designed the script to use the UNIX command *ps*, which provides the information listed in Table 4.3.

Table 4.3: Memory information provided by the UNIX command *ps*

| | |
|---|---|
| *VSZ* | Virtual Memory Size (RAM + swap) in kilobytes of the entire process. |
| *RSS* | Resident Set Size, the amount of physical RAM actually consumed by the application i. e. the non-swapped physical memory, in kilobytes |
| *SZ* | Size in physical pages of the entire process |

The memory measure script takes measurements from the moment the tool is invoked and continues until the tool terminates, so this includes the memory usage during model construction. Of the available data, we analyse the peak VSZ memory usage. The VSZ value represents the total amount of memory used by the process. The peak value tells us the amount of memory that is minimally required (by the tool) to verify the particular property. This makes it an interesting value for our performance comparison among the tools. As for the RSS and SZ values, they are recorded for possible future analysis but are not used in this thesis. In our opinion the RSS value is of less interest, since it does not reflect the real memory usage of a process (on a machine where

[2] The *time* command runs any specified program command (in our case the model checker tools) and produces timing statistics and resource usage about this program run.

[3] File I/O may consume considerable amounts of time, for example MRMC lists the complete probability vector in its output, which for large models might take up to several minutes to write to file.

swapping is enabled), it only represents the part of the total amount of memory that is stored in physical RAM. As for the SZ value, it is merely an different representation of the VSZ value. The SZ value is expressed in "number of pages[4]". The page size may vary between platforms, in our cause it is 4,096 bytes (4KB). Multiplying the "number of pages" with the page size results in the VSZ value in KB. We find KB more intuitive than number of pages and therefore choose to use the VSZ value. As stated above we use the Linux command *ps* to perform our measurement. Although this method is not perfect[5], the command gives us the means to conduct uniform measurements on our system on all the tools with their different programming languages. It provides a reasonable indication of the memory consumption of a process, moreover by using *ps* we can do a "black box" memory consumption measurement of a process i. e. we can perform measurements without having to change anything to the model checker tools itself. Other methods, such as profiling tools were investigated, but proved to be less suitable. Most of these tools were either, commercial, intended for the Windows platform or they required some form of modification to the application that needed to be profiled. Optionally, we could have used the memory consumption data reported by the model checker tools itself, but this would require all of the tool to produce such data, which is not the case.

**Automation.** All experiments and measurements are automated by means of Linux shell scripts. This enables us to easily repeat experiments many times and collect data in a uniform style. An experiment consists of verifying one property on one particular model using one of the model checker tools. The tools are restarted before each experiment; this prevents features, such as caching, to influence the results. Each experiment is repeated 20 times, after which we calculate the sample mean and standard deviation of data such as the model checking time. The standard deviation indicates the dispersion of a sample set i. e. it shows the range of variation from the sample mean of a group of measurements. Calculation of the mean requires no explanation, for the standard deviation we use the following formula:

$$G(\underline{x}) = \sqrt{\frac{n\sum {x_i}^2 - (\sum x_i)^2}{n(n-1)}} \cdot \frac{\mathrm{STD}[n-1]}{\sqrt{n}} \tag{4.1}$$

Where $\underline{x} = (x_1, \ldots, x_n)$, is the collected sample set, $n$ the number of samples, and $\mathrm{STD}[n-1]$ is the Student's t-Distribution [26] with $n-1$ degrees of freedom. The number of runs is limited to 3 instead of 20 whenever the total time of a single run exceeds 30 minutes, this prevents experiments from consuming excessive amounts of time, but results in a less accurate standard deviation. This is why we use the student's t-distribution in Formula 4.1, it takes the number of samples into account. If a single verification run requires more than 24 hours, we abort the process and denote the model check time as $\infty$ in the charts and tables. As a rule, the result of a time measurement are denoted as: $avg \pm stdev$. For example $16.54 \pm 3$, means the average time is 16.54 seconds with a deviation of 0.03 seconds faster or slower. The raw data produced by the tools is processed automatically by means of shell scripts and a Java application that we designed to perform the necessary calculations and generate results for easy display in LaTeX[6].

The measurement procedures described above are focussed on tools that have command-line support. For a tool without command-line support, such as ETMCC (see Table 3.6) it becomes difficult to automate the measurement process by means of scripts. Instead of reverting to solutions such as macro recorders (these do not work well in the graphical environment of Linux), we solved the issue by slightly modifying ETMCC. We build in support for accepting command-line arguments and printing of results to the standard output.

[4] A page is the smallest unit of memory handled by the operating system (i. e. a building block of memory).

[5] There are some ongoing discussions regarding the accuracy of the data produced by the *ps* command. However, we were not able to find any substantial evidence (or detailed research) on this matter. The main issue involves the fact whether or not to include the memory consumed by shared libraries, as done by *ps*.

[6] LaTeXis a document preparation system for high-quality typesetting, see `www.latex-project.org`.

## 4.2 Model construction

Before discussing the selection of case studies, which is the topic of Section 4.3, we want to address the issue of creating the actual models (DTMCs/CTMCs) for each of the tools. The case studies that we selected had to be modelled using the model description language of each of the tools. For some tools the models where readily available, for instance from [2] or from the example models included in the tool distribution. As became apparent in Chapter 3, the tools use different modelling languages. Despite this fact, we require the model to be equivalent (across all tools). The procedure for model creation and generation is as follows:

1. Obtain or create the desired model in the PRISM programming language.
2. Use the export function[7] of the PRISM tool to export the model into the *.tra* and *.lab* format accepted by MRMC and ETMCC.
3. Create the model for YMER by adjusting[8] the PRISM model using the syntax supported by YMER.
4. Obtain or create the desired model in the VESTA programming language. (The TQN and CSP case studies are provided in the standard distribution of the tool. The BDP case study was re-modelled by hand. We were not able to generate models for the SLE and RDP case studies due to parsing problems.)

By following above procedure, we minimize the chance of creating inequivalent case study models among the different tools. We attempted to generate models as large as possible by varying the model parameters. We discovered that there are two factors restricting the model size. One of which is the file size of the .tra files used by MRMC and ETMCC, which is limited to a maximum of 2GB[9]. PRISM can not export to a file exceeding this size, and even if we managed to generated the file, MRMC would not be able to read it due to limitations of the implemented file reader. In some cases we could not generate (and verify) our model due to the fact that PRISM crashed. We will not go into detail on the cause of this error, but suffice to say the error message reported a (known) problem with the CUDD package used for BDD's.

As MRMC and ETMCC do not support a built-in modelling language, their overhead to generate a sparse matrix representation is low compared to the sparse matrix generation by PRISM. In sparse mode, PRISM converts the MTBDD (which is always constructed first) to a sparse matrix after performing some pre-computations. This, along with the generation of the sparse matrix, may take a significant time and influence the model check time. This aspect should be considered when interpreting the experimental results.

**Properties.** The properties verified in each of the case studies were selected based on their comprising operator(s). For discrete-time models we used at least one property containing the unbounded $\mathcal{U}$ operator and one containing the bounded $\mathcal{U}$ operator. The long run operator $\mathcal{L}$ and the interval until $\mathcal{U}^{[k1,k1]}$ were not included, since there is only one tool (MRMC) that supports them. The next operator $\mathcal{X}$ is also excluded, since its verification is trivial. As for continuous-time models, we included the bounded and unbounded $\mathcal{U}$ operator, the steady-state $\mathcal{S}$ operator and the interval until. In addition we included a nested property (i. e. a property containing multiple operators) for at least one discrete-time and one continuous-time model. In some cases the properties may seem to make no sense in relation to the model, but remember that we selected them based on their operators (and parameters) and not their semantics.

The properties verified in each of the case studies were chosen with such parameters[10] that, in most cases, they will evaluate differently when the model parameters are adjusted. This increases

[7] For model export we used a beta version of PRISM (3.0 beta1). Before exporting (in command-line mode), one needs to define the correct labels in the PRISM property file in order to generate a working *.lab* file. The time consumption of exporting the model has not been measured.

[8] The are some minor differences between the PRISM and YMER modelling language syntax. For instance PRIMS declares a rate as ``const double lambda = 0.5", whereas YMER requires ``rate lambda = 0.5".

[9] The 2GB file size limit is not caused by the file system (*ext3*) of our local hard disk. It handles files up to ≈ 2TB.

[10] With the parameters of a PCTL/CSL property we mean the probability bound $p$ in $\mathcal{P}_{\bowtie p}$ and possible step/time $t$ in $\mathcal{U}^{\bowtie t}$, with comparison operator $\bowtie \in \{<, >, \leq, \geq\}$

the chance of detecting discrepancies in the results produced by the tools. The value of the probability bound $\bowtie p$ in the properties only effects the outcome (as in true or false), it does not influence the model check time and computed probabilities as they will be the same for all $p$ (except for the extreme cases 0 and 1). We avoid using PCTL/CSL formulas with probability bounds very close to (or exactly) the actual value, because statistical model checkers, such as YMER and VESTA, do not handle such properties very well. An example of a situation where VESTA provides an inaccurate answer is illustrated using the simple DTMC in Figure 4.1. The

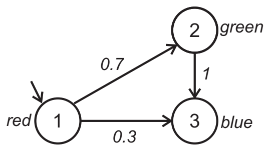


Figure 4.1: Simple DTMC.

probability to move from the initial state *red* to *blue* is exactly 0.3. When verifying the property $\mathcal{P}_{\geq 0.3}[\mathcal{X}\ blue]$ the probability bound equals the actual value. Verifying this property results in unreliable answers (it should evaluate to true). For this particular example, experiments showed that VESTA produces the answer *true* about 50% of the time (and false otherwise), with default error bound $\alpha = \beta = 0.01$. Even when narrowing the error bound to 0.0001 it still produces a wrong answer half of the time. We will not go into detail on the cause of this particular problem. It is a known phenomenon for statistical model checkers, that is related to the indifference region [79].

## 4.3 Case studies: data collection and interpretation

To compare the tools we need to compare their performance on model checking equivalent models. This requires a selection of case studies that serve as a benchmark. We present five representative case studies, taken from the literature on performance evaluation and probabilistic model checking, see for instance [82, 69, 84, 36]. There are three discrete-time and two continuous-time case studies, as shown in Table 4.4.

Table 4.4: Case studies

| | |
|---|---|
| *discrete-time* | Synchronous Leader Election |
| | Randomized Dining Philosophers |
| | Birth-Death process |
| *continuous-time* | Tandem Queuing Network |
| | Cyclic Server Polling System |

For each case study we present; a brief description, the size of the model, the model check times and peak memory usage for each verified property. The timing and memory data is presented by histograms where the linear x-axis indicates the model parameters that determine the state-space size, and the log-scale y-axis indicates the verification time (in seconds) or the memory consumption (in Kbytes). Note that the y-axis is log-scale, so small differences are substantial. Many detailed performance information can be found in the Appendices. For instance, the exact timing values for all case studies are presented in Appendix C and for detailed memory values, see Appendix D. Information related to model size and the amount of samples collected by the statistical tools is presented in Appendix B.

### 4.3.1 Synchronous Leader Election

**Description.** The Synchronous Leader Election (SLE) protocol [44] describes the following problem: given a synchronous ring of $N$ processors, design a protocol such that they will be able to elect a leader (a uniquely designated processor) by sending messages around the ring. The protocol proceeds in rounds and is parameterized by a constant integer $k > 0$. Each round begins by all processors (independently) choosing a random number (uniformly) from the set $[1 \ldots k]$ as an id. The processors then pass their ids around the ring. For instance, if we take $N = 4$ and $k = 6$, then each processor will pick a random id from the integer set [1,2,3,4,5,6]. It will take one complete round (i. e. a minimum of $N + 1$ transitions/steps are required for initialisation and for each processor to send an id to its neighbour and at the same time receive an id of its other neighbour) in order for every processor to see the ids of every other processor and establish if its own id is unique. If there is a unique id, then the processor with the maximum unique id is elected the leader, and otherwise the processors begin a new round. We assume that the ring is synchronous, meaning that there is a global clock and at every time slot; a processor reads the message that was sent at the previous time slot (if it exists), makes at most one state transition, and then may send at most one message [2]. The protocol is used in several studies, for instance [54, 32, 27]. The adjustable parameters of the model are presented below, as well as the list of PCTL properties to be verified.

| | | |
|---|---|---|
| parameters | $n$ | number of processes |
| | $k$ | size of the random number choice set $[1 \ldots k]$ |
| properties | $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ \mathit{leaderelected}]$ | With probability 1, eventually a leader is elected. |
| | $\mathcal{P}_{\geq 0.85}[\mathit{true}\ \mathcal{U}^{\leq 5}\ \mathit{leaderelected}]$ | The probability of electing a leader within 5 steps. In our model, it takes $n + 1$ steps (i. e. transitions) to complete a full round (including initialization). Suppose n=4, then we have defined the probability of electing a leader within 1 round, since $\mathit{rounds} = \frac{\mathit{steps}}{n+1}$. |
| | $\mathcal{P}_{\geq 0.99}[\mathit{true}\ \mathcal{U}^{\leq 40}\ \mathit{leaderelected}]$ | Similar to above, with a different bound on the number of rounds. |

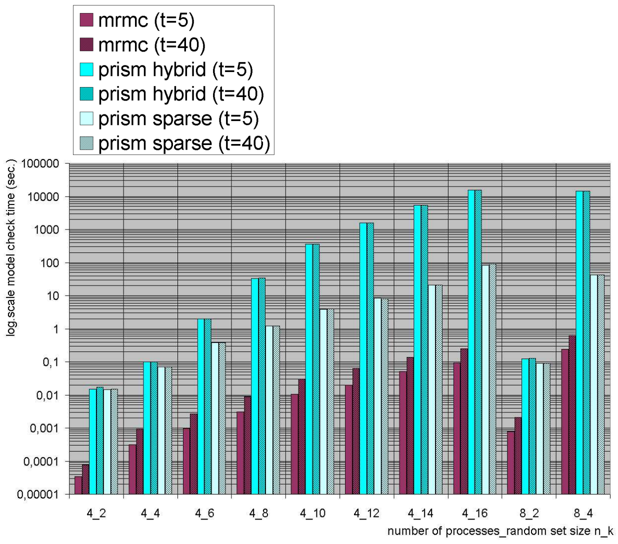
**Model size.** The number of processes and the size of the random number set, in the SLE protocol, is varied according to the values $n$ and $k$ in Table 4.5, which shows the corresponding number of states and transitions.

Table 4.5: SLE - states and transitions

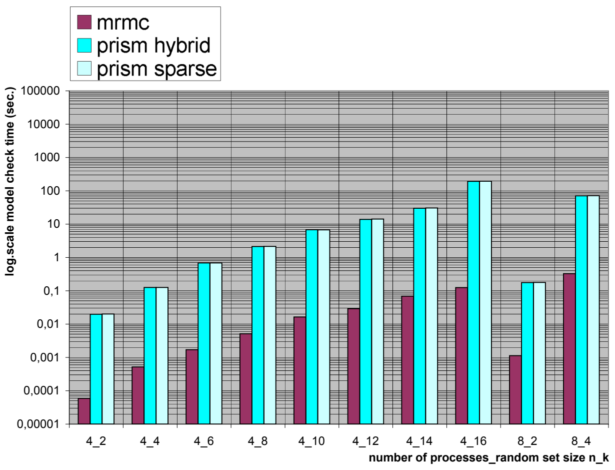
| n | k | #states | #transitions |
|---|---|---|---|
| 4 | 2 | 55 | 70 |
| | 4 | 782 | 1,037 |
| | 6 | 3,902 | 5,197 |
| | 8 | 12,302 | 16,397 |
| | 10 | 30,014 | 40,013 |
| | 12 | 62,222 | 82,957 |
| | 14 | 115,262 | 153,677 |
| | 16 | 196,622 | 262,157 |
| 8 | 2 | 1,803 | 2,058 |
| | 4 | 458,847 | 524,382 |

The SLE protocol has been verified using only MRMC and PRISM (sparse and hybrid), since ETMCC and YMER do not support discrete-time models. As mentioned earlier, we were unsuccessful in creating an equivalent VESTA model for this case study.

**Model check time.** The time it takes to verify a property (i. e. model check time) is displayed using bar charts such as Figure 4.2. As mentioned before, the log-scale y-axis shows the model check time in seconds and the x-axis shows the model parameter(s), in this case $n$ and $k$. Figure 4.2(a) presents the performance on verifying the property $\mathcal{P}_{\geq p}[\mathit{true}\ \mathcal{U}^{\leq t}\ \mathit{leaderelected}]$ for $t = 5$ and $t = 40$. Figure 4.2(b) does the same for $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ \mathit{leaderelected}]$.



(a) $\mathcal{P}_{\geq p}[\mathit{true}\ \mathcal{U}^{\leq t}\ \mathit{leaderelected}]$



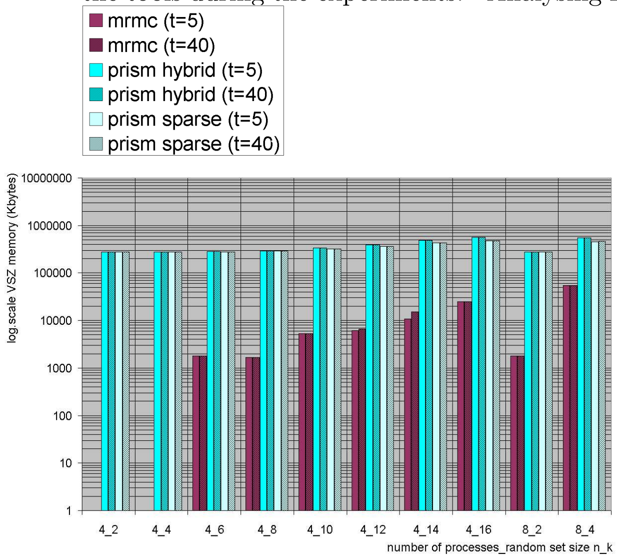
(b) $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ \mathit{leaderelected}]$

Figure 4.2: Synchronous Leader Election - model check times

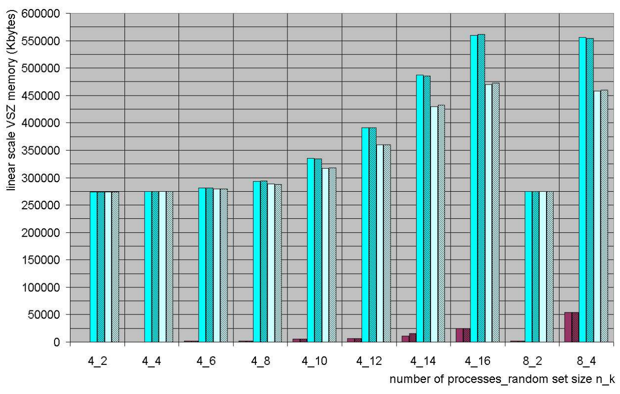
These charts clearly show that MRMC outperforms the PRISM hybrid and sparse engine by a wide margin, for both the bounded and unbounded $\mathcal{U}$ operator. For instance, MRMC is able to verify $\mathcal{P}_{\geq p}[\mathit{true}\ \mathcal{U}^{\leq 40}\ \mathit{leaderelected}]$ (on the model with parameters $n = 4$, $k = 16$) in 0.25 sec., whereas it takes PRISM$^S$ 93 sec. and PRISM$^H$ 4h15m. An obvious reason for the large performance difference between MRMC and PRISM would be that one of the tools sacrifices speed for accuracy (or vice versa). However as it turns out the probability vectors generated by MRMC and PRISM match up to approx. 6 digits, which is acceptable considering $\epsilon$ is set to $10^{-6}$. Studying the difference between both PRISM engines shows that, as expected, the PRISM sparse engine outperforms the hybrid engine, in this case by at most a factor of 340. For the unbounded until property in Figure 4.2(b) there appears to be no performance difference between both engines, we will address this phenomenon later on (in Section 4.4.2, which discusses the causes of performance differences.) When examining Figure 4.2(a) more closely there emerges another difference between the tools. When the time bound $t$ increases from 5 to 40, we can see that it takes MRMC longer to verify the formula. Which is to be expected, since there is a direct relation between $t$ and the number of calculations performed by MRMC (see Section 2.5.3). The difference between $t = 5$ and $t = 40$ is hardly noticeable in the performance graph of both PRISM engines. This can be explained by the way PRISM operates. Prior to performing the computations that depend on $t$, PRISM will carry out some pre-computations on the matrix of the model. These pre-computations are independent of $t$ and take much longer than the second ($t$ dependent) part. Because the second part of the process is so quick compared to the first part we hardly notice the overall difference between model checking various time bounds. One might argue that the difference between the chosen time bounds ($t = 5$ and $t = 40$) is too marginal to show a significant difference in model checking time. We therefore performed a series of measurements with an increased time bound of $t = 400$ and found, as before, no significant difference in model checking time[11]. Detailed timing values for the SLE case study can be found in Appendix C.1.

[11] The lack of performance difference between the different time-bounds for the PRISM tool was *not* caused by "On-The-Fly Steady-State Detection" (OSSD) [59, 48]. When OSSD is triggered, PRIMS will report this event in its log files, which was not the case during verification of any of the properties in the SLE case study.

**Memory usage.** Figures 4.3 and 4.4 show the maximum VSZ (Virtual Memory Size) used by the tools during the experiments. Analysing Figure 4.3(a) we see that PRISM$^S$ uses less memory



(a) log. scale - $\mathcal{P}_{\geq p}[true\ \mathcal{U}^{\leq t}\ leaderelected]$



(b) linear scale - $\mathcal{P}_{\geq p}[true\ \mathcal{U}^{\leq t}\ leaderelected]$

Figure 4.3: Synchronous Leader Election - peak VSZ memory

than PRISM$^H$. This effect becomes more apparent as the model size increases. When the bar chart is plotted using a linear scale, as done in Figure 4.3(b), the difference between the tools is even more emphasized. We see that, for all model sizes, MRMC uses the least memory. For example, observing the model parameters n_k = 4_16 we see that MRMC uses 24,872 KB ($\approx$ 24.3 MB), and PRISM$^S$ 472,536 KB ($\approx$ 461.5 MB) and that PRISM$^H$ requires 561,740 KB ($\approx$ 548.6 MB) of memory.

The memory usage for verifying the unbounded $\mathcal{U}$ operator, see Figure 4.4, shows that again MRMC uses less memory than PRISM$^H$ and PRISM$^S$. For this particular property there appears to be no difference between both PRISM engines.

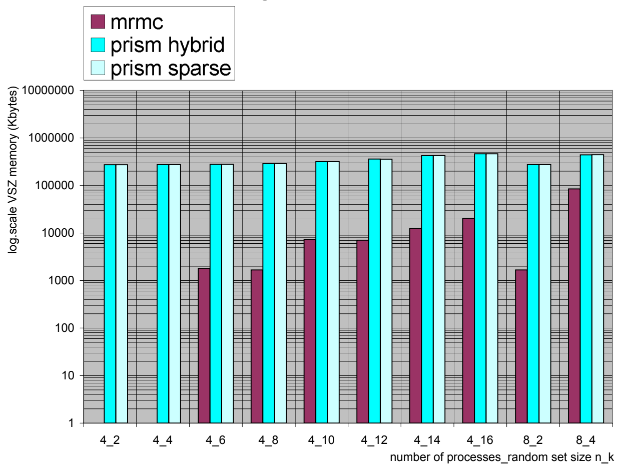
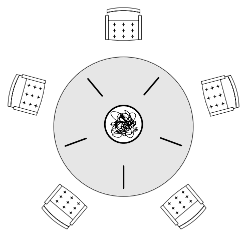


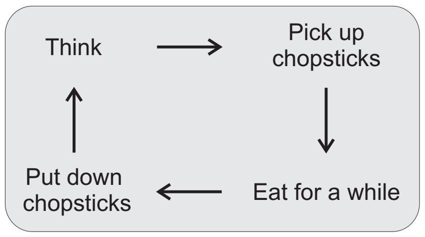
Figure 4.4: Synchronous Leader Election - peak VSZ memory - $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ leaderelected]$

**Verification results.** Studying the results produces by the tools (i. e. the true/false values stating whether or not the property is satisfied) we learned that each tool produced the exact same answer for every property and model parameter. This is what we expected to find, when comparing numerical tools set to use the same algorithm (Jacobi) and error bound ($\epsilon$).

### 4.3.2 Randomized Dining Philosophers

**Description.** As in the leader election problem, the Randomized Dining Philosophers[12] [66] describes a distributed randomized algorithm. Suppose we have $n$ philosophers who spend their lives just thinking and eating. We put the philosophers in a room and seat them at a circular table. There is a large plate of spaghetti in the centre of the table, which is constantly replenished. To the left of each philosopher lays a chopstick. If a philosopher feels hungry, he can pick up his own chopstick on his left. Since eating spaghetti with one chopstick is rather futile, the philosopher requires also the use of the chopstick on his right. When finished eating, he will put down both chopsticks and continue thinking. Of course a chopstick can only be used by one philosopher at a time. Therefore, not all philosophers can eat at the same time, they will need to share chopsticks (i. e. wait until it becomes available again). The dining philosophers situation is depicted in Figure 4.5. Followed by the model parameters and PCTL properties.


(a) Example table setting for 5 philosophers



(b) Philosopher process cycle

Figure 4.5: Dining Philosophers problem

| parameter | $n$ | number of philosophers |
|---|---|---|
| properties | $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ eat]$ | This is a liveness property: eventually some philosopher gets to eat. |
| | $\mathcal{P}_{\geq 0.9}[true\ \mathcal{U}^{\leq 20}\ eat]$ | Similar to above, only no we state that the probability that "eventually some philosopher gets to eat within 20 steps" must be at least 0.9. |

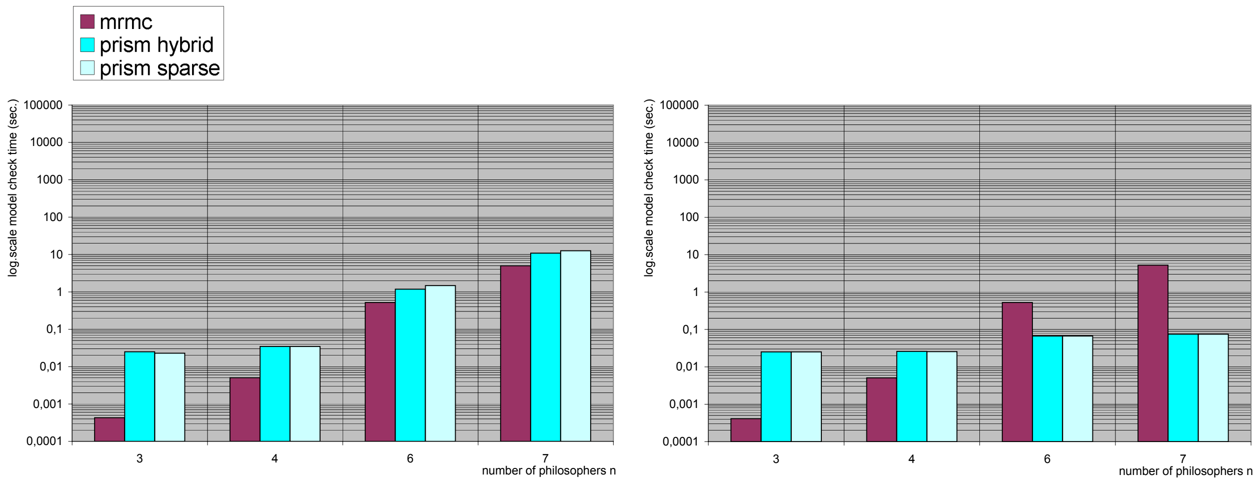
**Model size.** For the Dining Philosophers problem (RDP) we vary the number of philosophers $n$ according to Table 4.6, which shows the resulting number of states and transitions. This (discrete-time) case study has been applied to the PRISM and MRMC tool.

Table 4.6: RDP - states and transitions

| n | #states | #transitions |
|---|---|---|
| 3 | 770 | 2,845 |
| 4 | 10,022 | 47,432 |
| 6 | 594,790 | 4,170,946 |
| 7 | 5,454,562 | 44,070,594 |

[12] The Dining Philosopher problem has also been studied in [56] and was posed originally by Dijkstra [25]. It is used in for instance [54, 32, 63, 24].

**Model check time.** Figures 4.6(a) and 4.6(b) present the model check times on verifying the properties $\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 20}\ \mathit{eat}]$ and $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ \mathit{eat}]$. Our data shows that generally MRMC is faster[13] than PRISM except for verifying the unbound $\mathcal{U}$ operator on the larger models $n \in \{6, 7\}$.



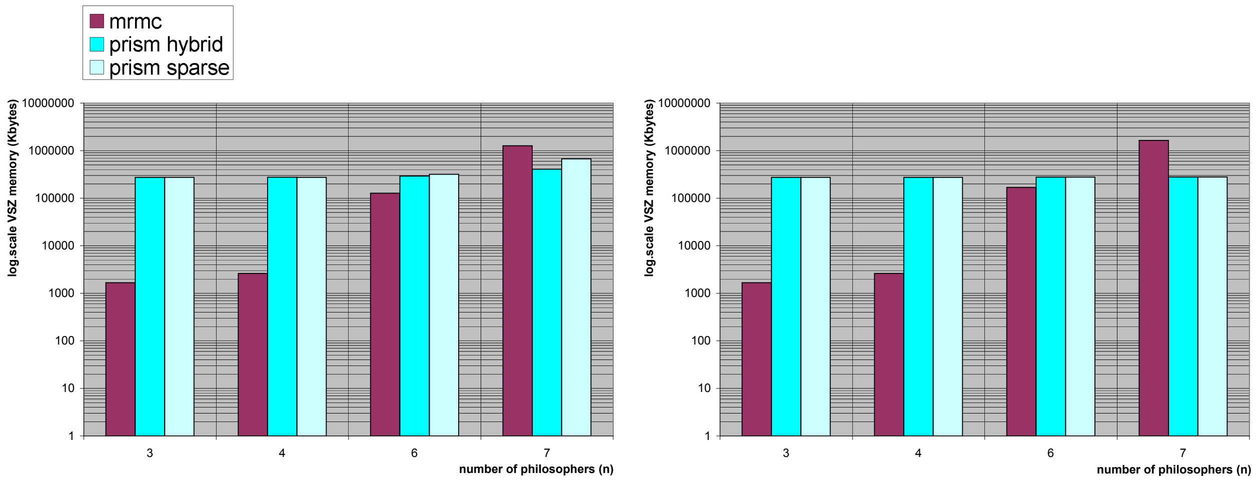
(a) $\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 20}\ \mathit{eat}]$ (b) $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ \mathit{eat}]$

Figure 4.6: Dining Philosophers - model check times

The difference between the PRISM hybrid and sparse engine for Figure 4.6(b) is negligible, but for Figure 4.6(a) we see that as the model size increases the difference is slightly in favour of the hybrid engine (e. g. for n=7 the hybrid engine is 1.74 sec. faster). Detailed model check times can be found in Appendix C.2.

**Memory usage.** Studying the memory usage, as depicted in Figure 4.7, we see the following. As the model size increases, the memory usage of MRMC grows rapidly, whereas that of PRISM increases rather slowly. When verifying the largest model size (k=7), both PRISM$^H$ and PRISM$^S$ use less memory than MRMC. The most extreme difference was measured during the verification of $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ \mathit{eat}]$ with k=7, here MRMC used 1,644,692 KB ($\approx$ 1.60 GB), PRISM$^H$ required 279,376 KB ($\approx$ 272.8 MB ) and PRISM$^S$ 279,104 KB ($\approx$ 272.6 MB ). The opposite happens for the smaller model sizes, where MRMC uses the least memory. Similar to the leader election protocol there appears to be no difference in the memory usage between the two PRISM engines when verifying the unbounded $\mathcal{U}$ operator (see Figure 4.7(b)). For the bounded $\mathcal{U}$ operator (Figure 4.7(a)) we see that, as the model size increases, PRISM$^S$ requires more memory than PRISM$^H$.

[13] Note that the speed mentioned here is related to the model check time only. It is worth mentioning that although it takes MRMC a near 5.3 seconds to model check the property $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ \mathit{eat}]$ for the largest model size ($n = 7$), the total time required is $\approx$ 225 seconds. This is caused by the fact that MRMC spends a considerable amount of time on file I/O, for instance it needs to read the large model description files i. e. the *.lab* and *.tra* file, which have a size of 88.2 MB and 1.3 GB respectively. For the same model size and property, PRISM$^S$ and PRISM$^H$ shows a model check time of $\approx$ 0.074 seconds and a total elapsed time of $\approx$ 1.66 seconds.



(a) $\mathcal{P}_{\geq 0.9}[true\ \mathcal{U}^{\leq 20}\ eat]$ (b) $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ eat]$

Figure 4.7: Dining Philosophers - peak VSZ memory

**Verification results.** The results of MRMC, PRISM$^H$ and PRISM$^S$ are consistent, meaning that for each property and model parameter they produced equal answers.

### 4.3.3 Birth-death process

**Description.** Birth-death processes [58, 46] are used in numerous fields, for instance to model the growth of a population. States in a birth-death process are numbered by integers that denote the current population size. In a birth-death process, the change in population size can occur by at most one, an increase in size is denoted as "birth" whereas a decrease is denoted as "death". The birth-death processes are related to queuing theory, for example we might state that the population represents "customers in the queue at the post office". Birth would then represent the arrival of a new customer and death the departure of a customer. State changes in a birth-death process can only happen between neighbouring states, this situation is depicted in Figure 4.8.

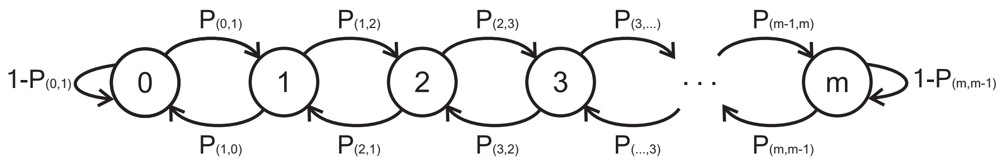


Figure 4.8: Birth-death process

To model check the birth-death process with the selected model checker tools we create a finite Markov chain by limiting the maximum population size ($m$). The probability of growth ($P_{(n,n+1)}$) and death ($P_{(n,n-1)}$) can be made dependent on the current population size ($n$). We defined the transition matrix for our case study as:

$$P_{ij} = \begin{cases} \lambda & i=0 \ \wedge \ j=1 \ \wedge \ n=0 \ \text{;birth from the initial state} \\ \frac{\lambda}{\lambda+(n\cdot\mu)} & j=i+1 \ \wedge \ (0<n<m) \ \text{;birth} \\ \frac{n\cdot\mu}{\lambda+(n\cdot\mu)} & j=i-1 \ \wedge \ (0<n<m) \ \text{;death} \\ \mu & i=m \ \text{;death from the } n=m \text{ state} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

The constants $\lambda$ and $\mu$ in Formula 4.2 are set to 0.8 and 0.001 respectively, in addition we define the chance of remaining in the initial state as $1-\lambda$ and for state $m$ we have $1-\mu$. The chosen $\lambda$ and $\mu$ values result in a decreasing probability of birth as the population grows, and at the same time an increasing probability of death. The size of the model depends on the maximum allowed population size. The model parameters and PCTL properties for this case study are listed below.

| parameter | $m$ | maximum population size |
|---|---|---|
| properties | $\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq \frac{m}{2}}\ (n=\frac{m}{4})]$ | The probability that we reach a quarter of the maximum population, within $\frac{m}{2}$ steps, is at least 0.9. |
| | $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 100}\ (n=70)]\ \mathcal{U}\ (n=50)]$ | With a probability of 1 we will eventually reach a population size of 50, while maintaining a minimum of probability of 0.9 of eventually reaching a population of size 70 within 100 steps. |
| | $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ (n=m)]$ | We will eventually reach the maximum population size. |

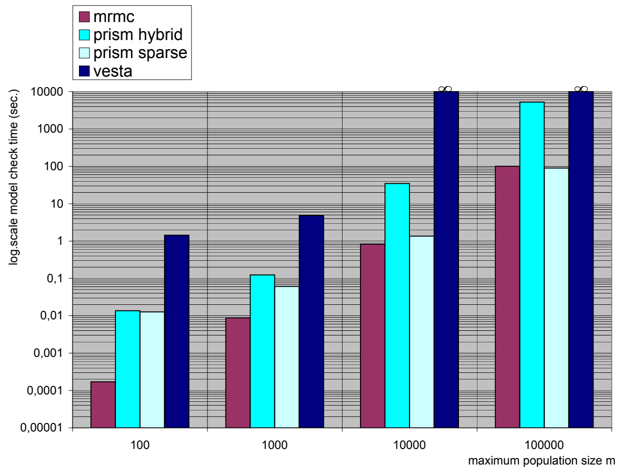
**Model size.** For the Birth-death process we varied the maximum population size $m$ as shown in Table 4.7, where the number of states is always $m+1$ and the transitions $2\cdot(m+1)$. The

Table 4.7: Birth-death process - states and transitions

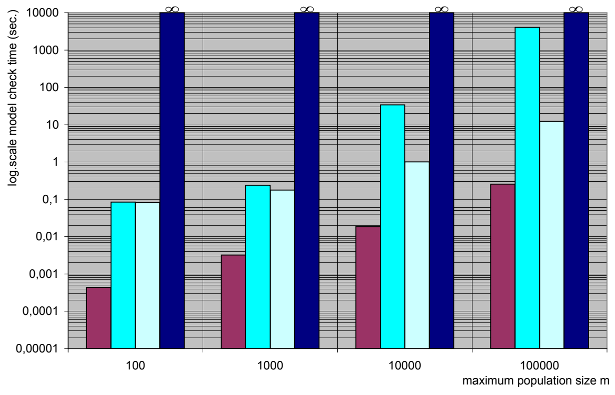
| m | #states | #transitions |
|---:|---:|---:|
| 100 | 101 | 202 |
| 1,000 | 1,001 | 2,002 |
| 10,000 | 10,001 | 20,002 |
| 100,000 | 100,001 | 200,002 |

tools MRMC, PRISM and VESTA where used to verify the three properties shown in Figure 4.9.

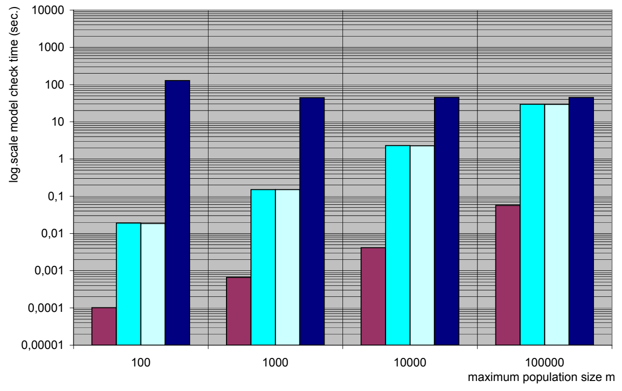
**Model check time.** Starting with Figure 4.9(a) we see that MRMC outperforms PRISM$^S$ for the smaller model sizes ($m \leq 1,000$), for the largest model size PRISM$^S$ wins with $\approx$ 10.4 sec. difference. The PRISM$^H$ engine is in all cases slower than PRISM$^S$, for instance in case of the largest model size ($m = 100,000$) we have PRISM$^H$ $\approx$ 1h27m versus PRISM$^S$ $\approx$ 90 sec. (and MRMC $\approx$ 100.6 seconds). The exact performance values can be found in Appendix C.3. When we look at VESTA we see that it performs poorly compared to MRMC and PRISM. It is slower for the model sizes ($m \leq 1,000$) and for larger sizes it could not provide an answer after continuously operating for 24 hours and had to be aborted. Investigating the amount of samples taken by VESTA (see Table B.4 at $m = 1,000$) shows this amount to be very high ($\approx$ 2 million), compared to the number of states in the model (1,000). We suspect that too many samples are required because the event $n = \frac{m}{4}$ is rather rare. For the nested formula in Figure 4.9(b) we see that, for all model sizes, MRMC is the fastest followed by PRISM$^S$ and PRISM$^H$. It takes MRMC $\approx$ 0.2561 sec. to verify the formula $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n=50)]$, whereas PRISM$^S$ and PRISM$^H$ need 12.24 and 4061 (1h7m41s) seconds respectively. VESTA offers no solution for any of the model sizes after running 24 hours and seems to be stuck in an infinite loop of collecting samples. For the PCTL formula $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ (n=m)]$, see Figure 4.9(c), it is clear that MRMC once again is the fastest tool, followed by PRISM$^S$ and PRISM$^H$, where the latter two show no significant performance difference. VESTA has the worst performance and in addition provides incorrect answers, where PRISM and MRMC state the property to be *true* in the initial state (for all model parameters $m$), VESTA reports it to be *false* (for all model parameters and for all 20 runs).



(a) $\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq \frac{m}{2}}\ (n = \frac{m}{4})]$



(b) $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n = 50)]$



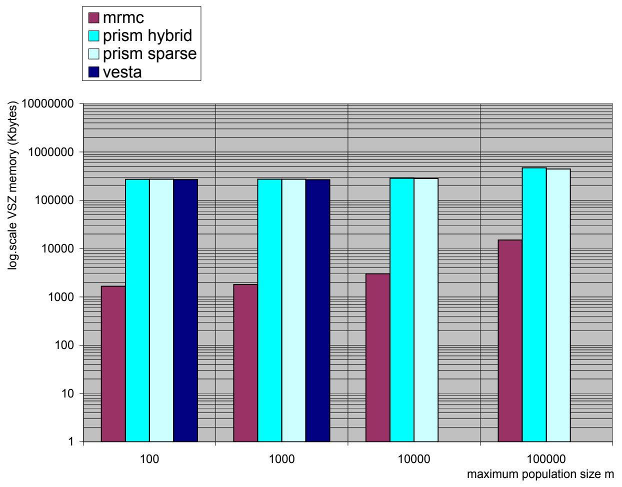
(c) $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ (n = m)]$

Figure 4.9: Birth-death process - model check times

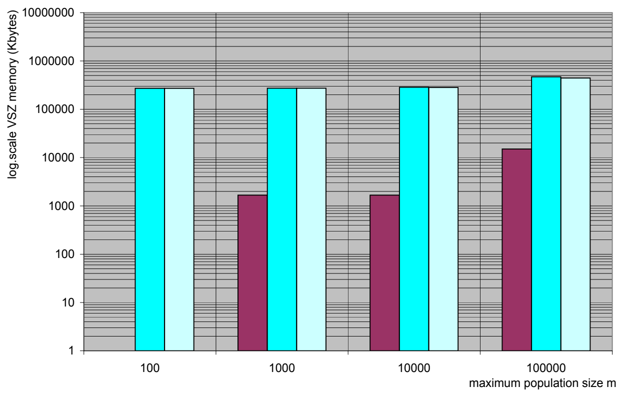
**Memory usage.** The memory usage for all three of the verified PCTL formulas is displayed in Figure 4.10.

In each case MRMC is the tool requiring the least memory. In Figure 4.10(a) and/or Table D.3 we can see that (for the largest model size) MRMC uses 15,016 KB ($\approx$ 14.7 MB), PRISM$^S$ 443.460 KB ($\approx$ 433.1 MB) and PRISM$^H$ 468.704 KB ($\approx$ 457.7MB). In this particular case the sparse engine is not only faster than the hybrid engine, it also uses less memory. The same effect can be seen in Figure 4.10(b). The memory related output reported by PRISM itself confirms our measurements, it indicates that the "hybrid MTBDD matrix" uses more memory (for the same model size) than the "sparse matrix". We believe this is caused by the fact that the birth-death model has little regularity[14], which is what the MTBDD approach tries to exploit (see Section 2.6). The memory usage of VESTA is fairly constant, as expected for a statistical tool. For smaller models it requires a similar amount of memory as used by PRISM, only when the model size increases we see that VESTA uses less than PRISM, but still more than MRMC. The missing memory values, in Figure 4.10(a) and 4.10(b), are due the fact that for these properties (and model parameters) VESTA seemed to require an infinite running time (i. e. at least more than 24 hours) and as a result the experiment was aborted.

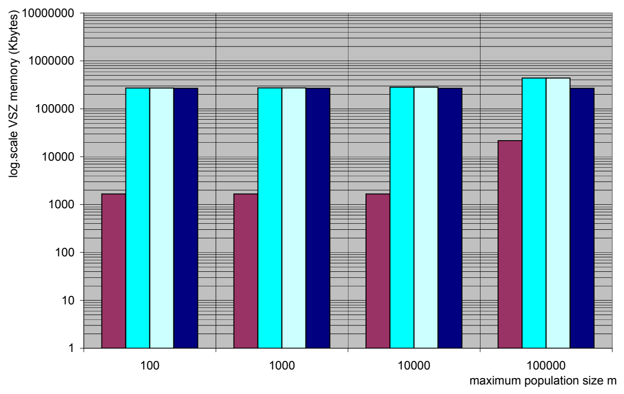
[14] The probability of death and birth depend on the current population size $n$ and therefore every transition in our birth-death model has a unique probability $p$, which makes exploitation of regularity difficult.



(a) $\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq \frac{m}{2}}\ (n = \frac{m}{4})]$



(b) $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n = 50)]$



(c) $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ (n = m)]$

Figure 4.10: Birth-death process - peak VSZ memory

**Verification results.** The results of MRMC, PRISM$^H$ and PRISM$^S$ are equal in all situations, as for VESTA; it produced an incorrect answer for all verifications of the property $\mathcal{P}_{\geq 1}[\mathit{true}\,\mathcal{U}\ (n = m)]$, which should be satisfied (i. e. true) according to MRMC and PRISM. Verifying this unbounded until property, stating that we will "eventually reach the maximum population", would theoretically require sample paths of infinite length. VESTA mitigates this problem by using a stopping probability $p_s$[15]. This means that, while sampling a path from a state, VESTA will stop and return the path so far simulated with probability $p_s$ [70]. Because of the nature of the model there is one very long path from the initial state to the only state satisfying "maximum population" ($n = m$), where the probability of reaching the state $n = m$ becomes ever smaller. Thus when sampling, at some point the chance of stopping becomes greater than the chance of reaching the $n = m$ state, and VESTA will evaluate the property to *false*. A small experiment where we lowered the stopping probability for $n = 100$ from 0.05 to 0.01 showed that, for these settings, VESTA produced correct answers. This supports our theory of the fault being caused by the stopping probability. The timing measurements for this experiment are not displayed here, because we settled on comparing the tools in their default mode.

[15] The stopping probability $p_s$ for VESTA is (default) 0.05, see Appendix A.

### 4.3.4 Tandem Queuing Network.

This case study, taken from [39, 2] (see also [36, 84, 69, 82, 86]), consists of two sequentially composed queues, each of capacity $n$. Messages arrive at the first queue and stay in the queue for some time, before getting routed to the second queue, from where they eventually leave the system. The time between arrival of messages at the first queue is exponentially distributed with rate $\lambda = 4n$. If the first queue is not empty and the second queue not full, then messages are routed from the first queue to the second queue. The routing time distribution is a two-phase Coxian [22] distribution with parameters[16] $\mu_1, \mu_2$ and $a$. The processing time at the second queue is exponentially distributed with rate $\kappa = 4$. Figure 4.11 shows the tandem queueing network and its parameters, which is followed by the CSL property listing.

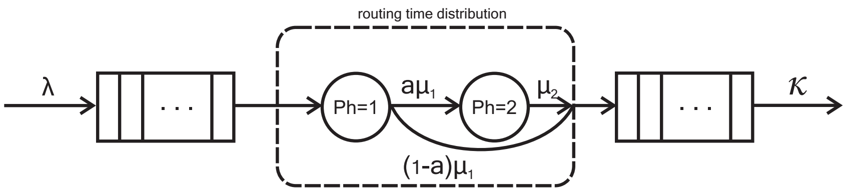


Figure 4.11: Tandem Queueing Network of two sequentially composed queues.

| parameter | $n$ | capacity of both queues |
|---|---|---|
| properties | $\mathcal{S}_{<0.01}[\mathit{full}]$ | The steady-state probability of the queueing network being full (i. e. each queue contains $n$ messages and Ph=2)is less than 0.01. |
| | $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ \mathit{snd}]]$ | The steady-state probability to be in a state that can reach a state in which the second queue is full in a single step with probability $> 0.1$ satisfies $> 0.2$. The steady-state probability of "the second queue becoming full in the next state with probability greater than 0.1" is greater than 0.2. |
| | $\mathcal{P}_{<0.1}[\mathit{true}\,\mathcal{U}^{[0.5,2]}\ \mathit{full}]$ | The probability of the queueing network becoming full between 0.5 and 2 time units is less than 0.1. |
| | $\mathcal{P}_{\leq 0.01}[\mathit{true}\,\mathcal{U}^{\leq 2}\ \mathit{full}]$ | The probability of the queueing network becoming full within 2 time units is less than (or equal to) 0.01. |
| | $\mathcal{P}_{\leq 0.5}[\mathit{true}\,\mathcal{U}^{\leq 10}\ \mathit{fst}]$ | The probability that the first station of the tandem network becomes fully occupied within 10 time units is less than (or equal to) 0.5. |
| | $\mathcal{P}_{\geq 1}[\mathit{snd}\,\mathcal{U}\ \mathit{sndn}]$ | The probability of leaving a situation where the second queue is entirely populated is equal to 1. |

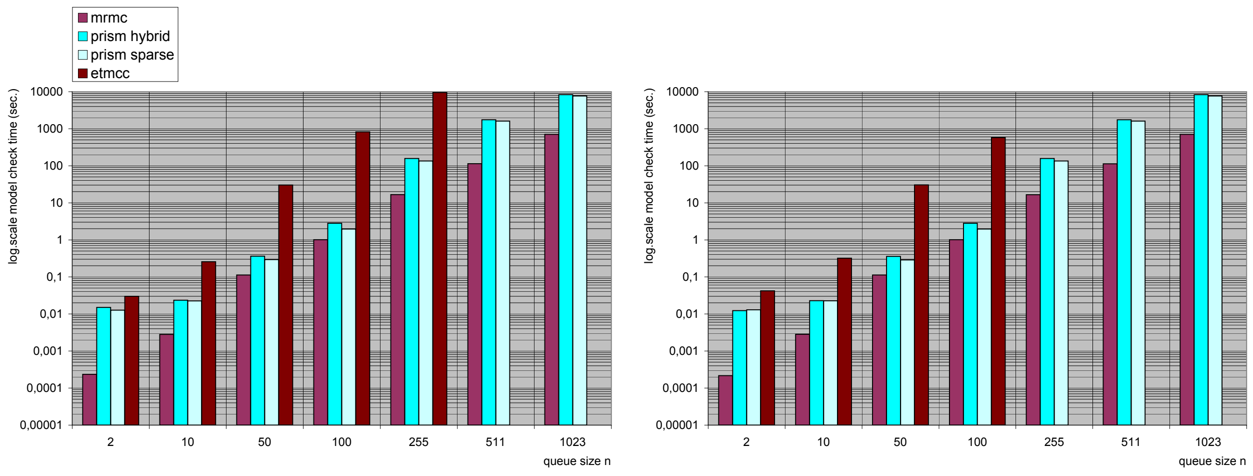
[16] The parameters for arrival time and the routing time distribution of the tandem queueing network remain constant during all experiments; we use $\mu_{i\in\{1,2\}} = 2$ which is the exit rate for the $i$th phase of the distribution, $1 - a = 0.9$ which is the probability of skipping phase two, and $ph \in \{1,2\}$ which denotes the current phase of the Coxian distribution.

**Model size.** The queue size $n$ of both queues in the Tandem Queuing Network (TQN) has been varied according to Table 4.8, resulting in a maximum of approx. 2 million states and 7 million transitions.

Table 4.8: TQN - states and transitions

| n | #states | #transitions |
|---:|---:|---:|
| 2 | 15 | 33 |
| 10 | 231 | 729 |
| 50 | 5,151 | 17,649 |
| 100 | 20,301 | 70,299 |
| 255 | 130,816 | 455,939 |
| 511 | 523,776 | 1,829,379 |
| 1023 | 2,096,128 | 7,328,771 |

**Model check time.** We start by examining the performance on verifying the steady-state operator, see Figure 4.12. This is done for the numerical tools only, since the others do not support steady-state detection[17] We see that, for the larger queue sizes, the timing values of ETMCC are missing. This is caused by the fact that ETMCC has run out of memory and terminated, thus no results are available. ETMCC is the slowest of the tools, whereas MRMC is the fastest, followed by PRISM$^S$ and PRISM$^H$. For example if we look at Figure 4.12(a) (and or Table C.4) queue size 1023 we see that MRMC requires 708.7 sec. ($\approx$ 12min.), PRISM$^S$ 7681 sec. ($\approx$ 2h8m), and PRISM$^H$ 8311 sec. ($\approx$ 2h18m31s). The CTMC of the TQN is strongly connected, meaning that a search for BSCC's (Bottom Strongly Connected Components), which is required for steady-state detection, will yield a single BSCC. Since all states are in a single BSCC, no reachability probabilities need be computed. The calculations for the BSCC steady-state probabilities for both formula require the same amount of iterations, this is why there is little difference between the model check times of the formulas in Figure 4.12(a) and 4.12(b). We would also like to note that for both steady-state properties, queue size 1023, PRISM reaches it maximum iteration point[18] and the iterative method is stopped early.



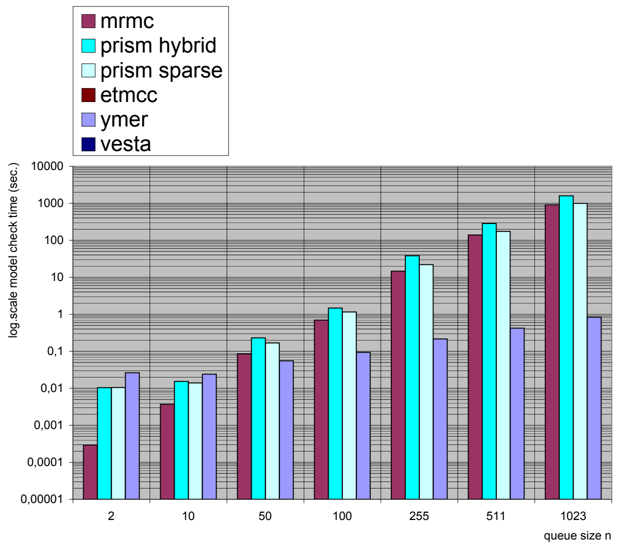
(a) $\mathcal{S}_{<0.01}[\mathit{full}]$ (b) $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ \mathit{snd}]]$

Figure 4.12: TQN - Steady-state, model check times

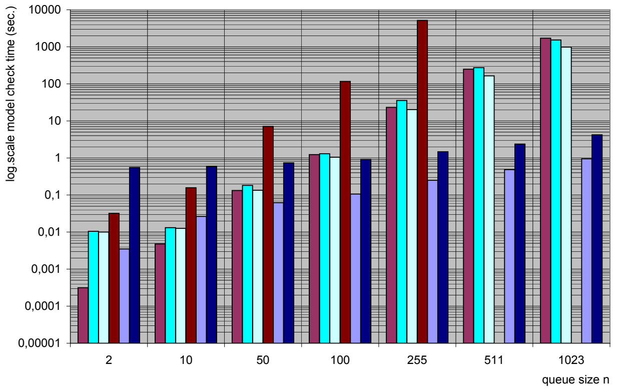
[17] In a nutshell, the steady-state operator in not supported by the statistical tools, because it is unclear when to stop the sample path generation.

[18] The maximum number of iterations was set to 10,000 for each (numerical) tool. The TQN case study is the only case where this maximum number of iterations was reached.

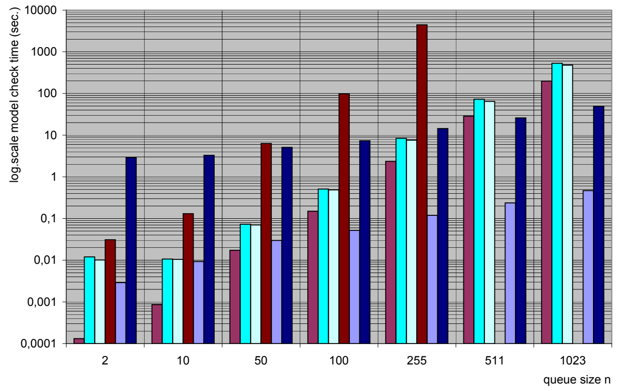
The graphs in Figure 4.13 show the performance on verifying the $\mathcal{U}$ operator. Starting with Figure 4.13(a), we see the performance results of MRMC, PRISM and YMER on $\mathcal{P}_{<0.1}[\mathit{true}\,\mathcal{U}^{[0.5,2]}\ \mathit{full}]$ (the other tools do not support the interval until operator). When the model size increases ($n >= 50$), the statistical tool YMER easily outperforms any of the other tools. Also note that



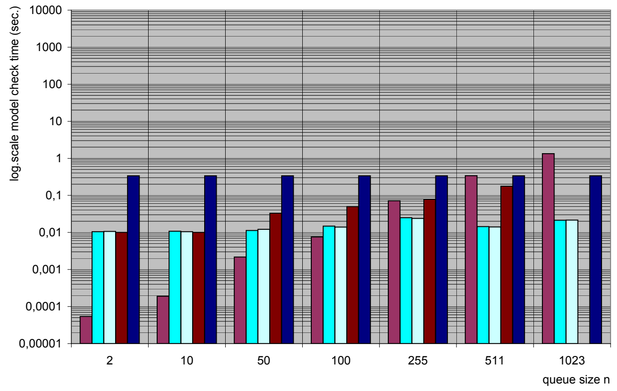
(a) $\mathcal{P}_{<0.1}[\mathit{true}\,\mathcal{U}^{[0.5,2]}\ \mathit{full}]$



(b) $\mathcal{P}_{\leq 0.01}(\mathit{true}\,\mathcal{U}^{\leq 2}\ \mathit{full})$



(c) $\mathcal{P}_{\leq 0.5}[\mathit{true}\,\mathcal{U}^{\leq 10}\ \mathit{fst}]$



(d) $\mathcal{P}_{\geq 1}[\mathit{snd}\,\mathcal{U}\ \mathit{sndn}]$

Figure 4.13: TQN - Until, model check times

an increase in model size has a less drastic effect on the performance of YMER as it does on the other tools. For each model size we see that MRMC is faster than PRISM$^S$. The largest difference is 83.4 sec., namely for queue size 1023, where MRMC and PRISM$^S$ requires 911.1 and 994.5 sec. respectively. The hybrid engine is slower than the sparse engine it uses (on average) 1583.8 sec. for the same queue size. Figures 4.13(b) and 4.13(c) display the performance graphs on $\mathcal{P}_{\leq 0.01}(\mathit{true}\,\mathcal{U}^{\leq 2}\ \mathit{full})$ and $\mathcal{P}_{\leq 0.5}[\mathit{true}\,\mathcal{U}^{\leq 10}\ \mathit{fst}]$ respectively. YMER is in both cases the fastest tool when it comes to the larger models ($n \geq 50$). VESTA on the other hand seem to have more difficulties competing against the numerical tools, it is faster only for the larger queue sizes (4.13(b): $n \geq 255$ and 4.13(c): $n \geq 511$). In figure 4.13(b) we see that MRMC has the fastest verification time for the smaller model sizes (up to $n = 100$). For larger values of $n$, PRISM$^S$ prevails. In every situation we see that the PRISM hybrid engine is slower than the sparse engine but still faster than ETMCC, which performs the poorest of all numerical tools. For queue size $n = 1023$, the tools PRISM$^S$, PRISM$^H$, and MRMC require approximately 983, 1531 and 1705 seconds respectively. As for ETMCC, it can only verify models of queue size up to and including $n = 255$, after which it runs out of memory. At this point we would like to note that we discovered a substantial difference in performance for the MRMC tool when repeating the same experiment with the "On-The-Fly Steady-State Detection" (OSSD) [59, 48] turned off, namely for $\mathcal{P}_{\leq 0.01}(true\,\mathcal{U}^{\leq 2}\ full)$, queue size 1023 we recorded 882 sec. in stead of 1705 sec. It is possible that PRISM would also perform faster without OSSD. This could not be verified seeing that PRISM enables OSSD by default and it can not be turned off. The effect witnessed here is already well known. By utilizing OSSD, verification time is longer (i. e. it doubles compared to not using OSSD) prior to the point from which a steady-state is detected during computations. Once the equilibrium is reached (and detected), the run time for the variant with OSSD turned on remains constant, whereas the runtime for verification without OSSD continues to grow linearly in $t$ [48].

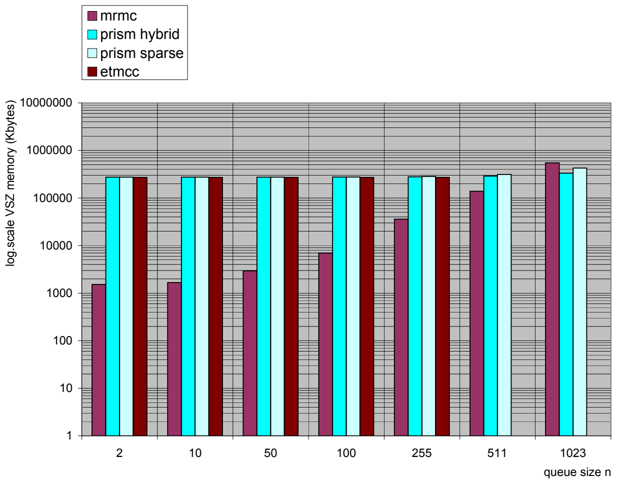
In Figure 4.13(c) we can see that verification of the property $\mathcal{P}_{\leq 0.5}[true\,\mathcal{U}^{\leq 10}\ fst]$ using the numerical tools is performed the fastest by MRMC for all model sizes. Performance wise, PRISM$^S$ and PRISM$^H$ are in second and third place, and ETMCC finishes last. Further analysis showed that on-the-fly steady-state detection was triggered in MRMC, PRIMS and ETMCC (for this particular property and all model sizes), which in this case reduced the number of iterations required by the model checking algorithm, and thus resulted in a faster model check time. Here we witnessed the benefit of OSSD, for instance MRMC took 28.42 sec. for queue size 511 when OSSD was turned on, and 520.7 sec. (not depicted) otherwise.

In Figure 4.13(d) we see the results of verifying the unbounded until property $\mathcal{P}_{\geq 1}[snd\,\mathcal{U}\ sndn]$. Although the model size does not seem to effect VESTA, its overall performance is not as good as the numerical tools (comparison with YMER was not possible, since it does not support the unbounded until operator). For smaller model sizes ($n \leq 100$) MRMC is the fastest tool, but as the model size increases its model check time exceeds that of PRISM and even ETMCC. For queue size $n = 1023$, both PRISM engines require $\approx 0.021$ sec., MRMC needs 1.338 seconds and ETMCC has run out of memory. A overview of exact timing values can be found in Appendix C.4.

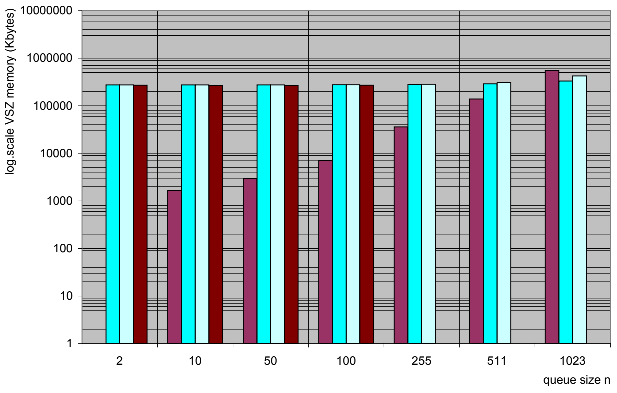
**Memory usage.** Figure 4.14 depicts the peak VSZ memory usage measured during verification of the formulas $\mathcal{S}_{<0.01}[full]$ and $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ snd]]$. As the queue size increases we see that the PRISM memory usage increases as well (although less dramatic as MRMC). We also notice that increasing the queue size increases the difference between PRISM$^H$ and PRISM$^S$, where the latter uses more memory. This indicates that PRISM stores the queuing model more efficiently, using the hybrid approach, but at the cost of performance (as demonstrated in Figure 4.12). As for MRMC; for the smallest model sizes we were unable to accurately measure its peak memory usage, because the measure period was to short (i. e. the verification is completed before enough measurements could be taken). MRMC uses less memory than both PRISM engines, but its memory consumption rapidly increases as the model grows larger and it can not maintain its lead after reaching the largest model size ($n = 1023$). ETMCC uses an average of 4.1 MB less than PRISM$^H$, until it reaches queue sizes $n = 511$ (Figure 4.14(a)) and $n = 255$ (Figure 4.14(b)), where it runs out of memory.

In these particular cases, there is no peak memory value available for ETMCC. This is caused by the fact that ETMCC produces an "`java.lang.OutOfMemoryError:Java heap space"`[19] error. This error appears soon after the tools starts. This could either mean that the application has a memory leak or that ETMCC really needs a lot of memory. A possible cause of the heap space error is that the maximum recursion depth (of a recursive function) is exceeded. The error is actually produced by the Java Virtual Machine (JVM). The JVM manages an internal heap of memory and imposes a fixed limit on the size of this heap. During runtime, the JVM will grow the heap by allocating more memory from the operating system when needed. Before satisfying a memory allocation, the JVM checks if the allocation would result in overstepping the heap size limit. If the JVM can not free memory by means of garbage collection and the limit would be exceeded, it will throw an "`OutOfMemoryError`". We could have tried to remedy this problem by

[19] This is the memory that the JVM uses to allocate java objects.



(a) $\mathcal{S}_{<0.01}[full]$



(b) $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ snd]]$

Figure 4.14: TQN - Steady-state, peak VSZ memory

increasing the Java heap size using certain runtime parameters[20], but this would have influenced our test environment. We need to remain consistent and use the same Java heap size for all experiments (this applies to all tools written in Java, see Table 3.6).

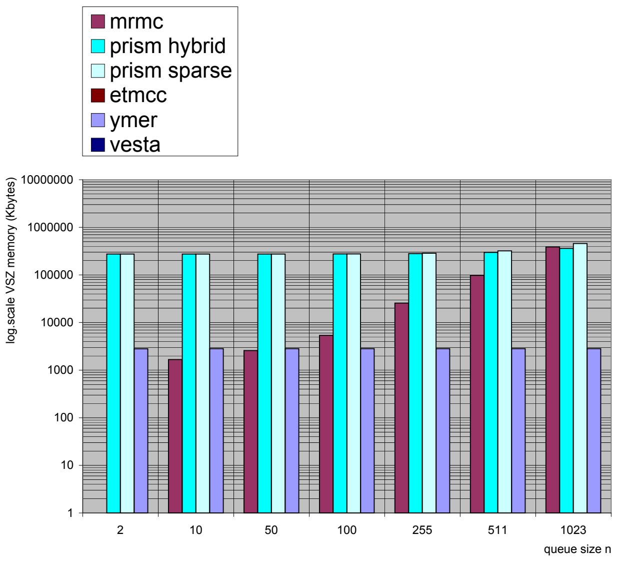
Moving on to the bounded $\mathcal{U}$ operator, of which the peak VSZ memory is presented in Figures 4.15(a) through 4.15(c), we see a similar memory consumption behaviour between the first two figures. We have excluded the graph for the property $\mathcal{P}_{\leq 0.5}[true\,\mathcal{U}^{\leq 10}\ fst]$, because it is almost identical to that of $\mathcal{P}_{\leq 0.01}(true\,\mathcal{U}^{\leq 2}\ full)$ in Figure 4.15(b) (the largest measured deviation is 676 Kbytes). Instead, we provide Figure 4.15(c), which shows the memory consumption for the property $\mathcal{P}_{\leq 0.01}(true\,\mathcal{U}^{\leq 2}\ full)$ on a linear scale, which emphasises the differences amongst the individual tools. We see that for queue sizes $n \geq 100$, YMER uses by far the least memory. MRMC starts out with a significantly lower memory consumption compared to PRISM, but MRMC shows an rapid increase as the model grows. After reaching the largest model size ($n = 1023$) the memory usage of MRMC has surpassed that of PRISM, at this point MRMC uses $\approx$ 500 MB, PRISM$^H$ $\approx$ 338 MB, PRISM$^S$ $\approx$ 429 MB and ETMCC has ran out of memory. As we noticed earlier when investigating the steady-state performance, we again see that as the model size increases the PRISM sparse engine consumes more memory than the hybrid engine, resulting in a trade-off between speed and memory consumption. The memory usage of VESTA is similar to that of PRISM, whereas we would expect VESTA to use less memory, since it is a statistical tool. We believe this might be caused by the fact that both tools use the Java Virtual Machine, which tends to claim a certain amount of memory at start-up and even though garbage collection[21] will decrease the JVM heap size, the JVM will not release its allocated memory back to the operating system, until the process is terminated.

Figure 4.15(d) presents the memory chart for the remaining property $\mathcal{P}_{\geq 1}[snd\,\mathcal{U}\ sndn]$. All the Java-based tools, PRISM,ETMCC and VESTA seem to require the same amount of memory, as for MRMC it starts out low and then increases up to a point ($n = 1023$) where it requires more than PRISM and VESTA (ETMCC has run out of memory).

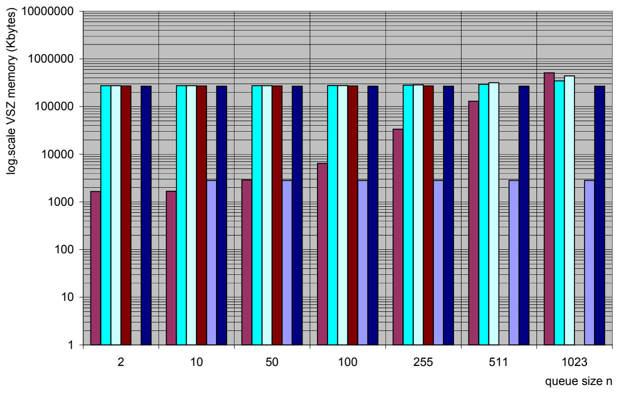
**Verification results.** The results of all tools matched 100%. This includes the statistical tools, meaning that YMER and VESTA produced the correct answer for all experiments (i. e. for all properties, model parameters and all 20 runs).

[20] The Java heap size can be adjusted using the runtime parameters:
`java -Xms<initial heap size> -Xmx<maximum heap size>`

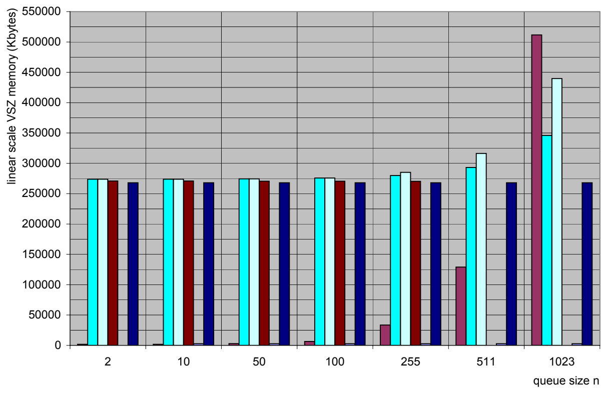
[21] For addition information on Java memory management, we refer to
http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf



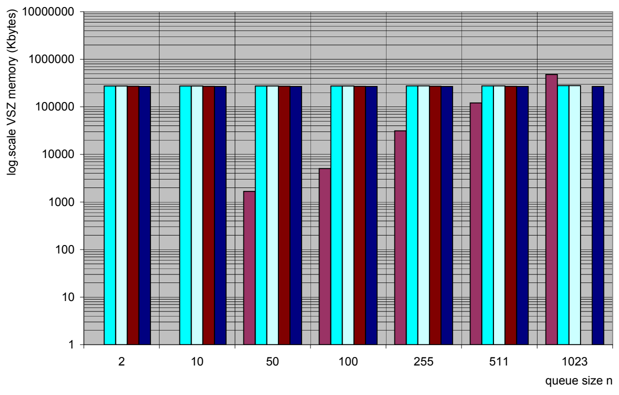
(a) $\mathcal{P}_{<0.1}[true\,\mathcal{U}^{[0.5,2]}\ full]$



(b) $\mathcal{P}_{\leq 0.01}(true\,\mathcal{U}^{\leq 2}\ full)$



(c) linear scale - $\mathcal{P}_{\leq 0.01}(true\,\mathcal{U}^{\leq 2}\ full)$



(d) $\mathcal{P}_{\geq 1}[snd\,\mathcal{U}\ sndn]$

Figure 4.15: TQN - Until, peak VSZ memory

### 4.3.5 Cyclic Server Polling System.

This case study describes a symmetric[22] polling system [43] consisting of $n$ stations and a server. Each station has a single-message buffer and the stations are attended by a single server in cyclic order. The server starts by polling the first station. If this station has a message in its buffer ($busy$), the server starts serving the station. Once the station has been served, or if there was no message in the buffer ($idle$), the server start polling the next station. After polling all stations, the server returns to polling the first station and thus beginning a new cycle. The polling and service times are exponentially distributed with rates $\gamma = 200$ and $\mu = 1$. The arrival rate of messages at a station is equal for all stations and is exponentially distributed with rate $\lambda = \frac{\mu}{n}$. Applications of this case study can be found in for instance [83, 82, 36, 69, 84, 86]. We will apply this case study for different values of $n$ using the following CSL properties.

| parameter | $n$ | number of stations |
|---|---|---|
| properties | $\mathcal{S}_{<0.2}[busy_1 \wedge \neg serve_1]$ | The long run probability that station 1 is waiting for the server is less than 0.2. |
| | $\mathcal{P}_{\leq 0.99}[true\,\mathcal{U}^{[40,80]}\ serve_1]$ | The probability that station 1 will be served within the time bound [40,80] is at most 0.99. |
| | $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\ poll_1]$ | Once station 1 has become busy (i. e. full), with probability of at least 0.5 it will be polled within at most $t \in \{5, 10, 20, 40, 80\}$ time units. |
| | $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\,\mathcal{U}\ poll_1]$ | Once station 1 has become busy, it will eventually be polled. |
| | $\neg(poll_1 \wedge poll_2)$ | The server never polls station 1 and 2 at the same time. |

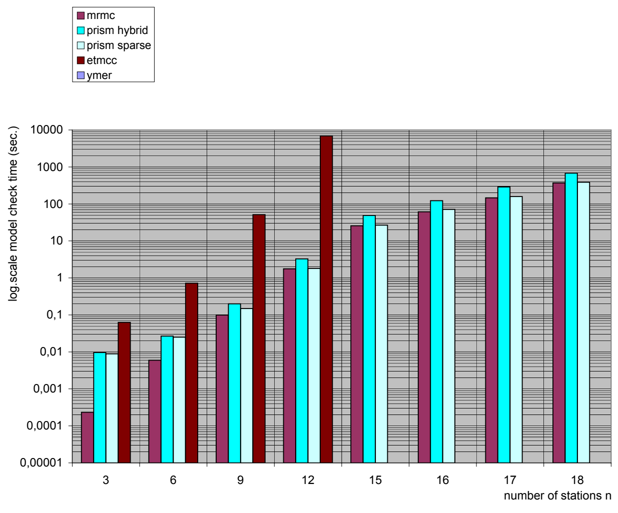
**Model size.** The size of the state-space for this model depends on the number of stations ($n$), as shown in Table 4.9. The largest possible model parameter is $n = 18$, for $n = 19$ the model description file (.tra file) could not be generated because it exceeds the 2Gb size limit (as explained in Section 4.2).

Table 4.9: CSP - states and transitions

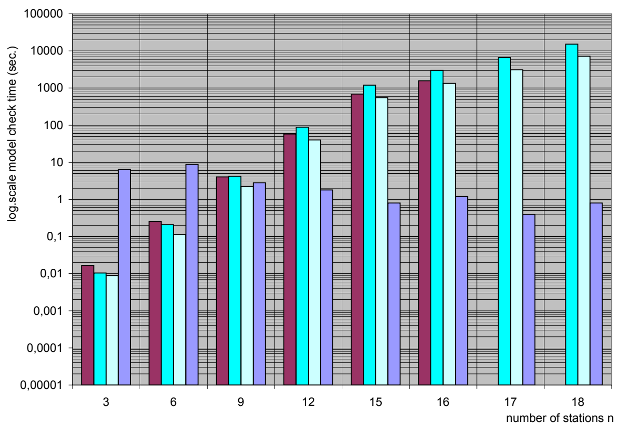
| **n** | **#states** | **#transitions** |
|---|---|---|
| 3 | 36 | 84 |
| 6 | 572 | 2,208 |
| 9 | 6,912 | 36,864 |
| 12 | 73,728 | 503,808 |
| 15 | 737,280 | 6,144,000 |
| 16 | 1,572,864 | 13,893,632 |
| 17 | 3,342,336 | 31,195,136 |
| 18 | 7,077,888 | 69,599,232 |

**Model check time.** Starting with the steady-state operator (Figure 4.16(a)) we see that MRMC is the fastest followed by PRISM$^H$, PRISM$^S$ and EMTCC, where the latter runs out of memory as the number of stations ($n$) exceeds 12. As for the property $\mathcal{P}_{\leq 0.99}[true\,\mathcal{U}^{[40,80]}\ serve_1]$ of which the model check times are shown in Figure 4.16(b), we can see that PRISM$^S$ outperforms MRMC and PRISM$^H$ for all model sizes. For instance, for $n = 16$ it takes PRISM$^S$ (on average) 1339.5 sec. (22.3 min.), MRMC 1574 sec. (26.2 min.), and PRISM$^H$ 2979 sec. (49.7 min.). For larger model sizes ($n \geq 12$) the statistical tool YMER is by far the fastest, for $n = 16$ it requires only 1.2 seconds.

[22] The cyclic server polling system is symmetric, because message arrival rates for all stations are equal.



(a) $\mathcal{S}_{<0.2}[busy_1 \;\wedge\; \neg serve_1]$



(b) $\mathcal{P}_{\leq 0.99}[true\,\mathcal{U}^{[40,80]}\; serve_1]$

Figure 4.16: CSP - Steady-state and interval Until, model check times

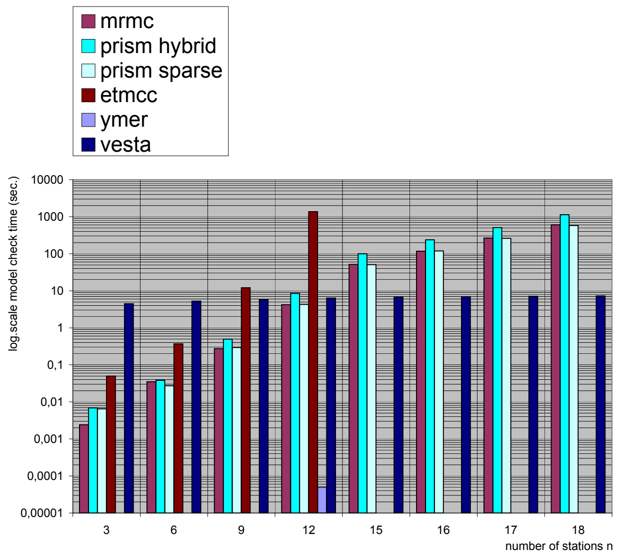
Figure 4.17 shows the performance on verifying the property $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$, where the time bound $t$ is set to 5, 10, 20, 40 and 80 successively. We encountered a problem when trying to verify this property using MRMC. Verification worked fine for model sizes $n \leq 6$, but for larger $n$ the tool produced the following error message: `ERROR: The number of BSCCs exceeds 255, this is not supported`[23]. The On-The-Fly Steady-State Detection (OSSD) proved to be the cause of this error. Since we would still like to be able to see how MRMC relates to the other tools, we turned OSSD off for the duration of this experiment i. e. for the property $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$ with $t \in 5, 10, 20, 40, 80$ and $n \geq 9$.

The data shows that, of the numerical tools, PRISM$^S$ is overall faster than MRMC and PRISM$^H$. The difference becomes more apparent as the time bound $t$ increases. For example, for $t = 80$ and $n = 16$ PRISM$^S$, PRISM$^H$, and MRMC[24] require respectively 533.5, 1166 and 1462 seconds, which is depicted in Figure 4.17(e) and Table C.5. Analysis of the PRISM log files showed that the on-the-fly steady-state detection was triggered for all model sizes in combination with the time bounds 40 and 80. This explains why there is no significant difference between the model check times of PRISM for these two time bounds, see Figures 4.17(d) and 4.17(e). On the other hand, we see that the model check times of MRMC increase when the time bound $t$ is raised, as is evident in Figure 4.17(f). This is caused by the fact that MRMC is unable to apply on-the-fly steady-state detection in these conditions, as explained above.

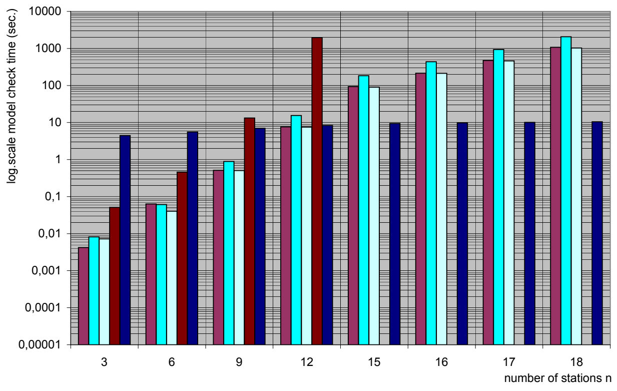
For the statistical tools we see that VESTA performs reasonably for larger model sizes, whereas YMER has excellent performance. Although the graphs in Figure 4.17 do not show values for YMER, checking Table C.5 learns that YMER reports to model check this property in 0 seconds. This is caused by the fact that YMER will only verify if the property holds in the initial state. The property $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$ can be rewritten as $\neg busy_1 \vee \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$ and since $\neg busy_1$ is satisfied in the initial state YMER needs not verify $\mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$ and thus computes the answer almost instantly. This conclusion is supported by the fact that the number of collected samples (see Table B.8) for this property is not specified by YMER, meaning no samples have been collected. We do note that if the order of the property would have been reversed (i. e. $\mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1] \vee \neg busy_1$ ) YMER would require more time, since it would first verify $\mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\; poll_1]$. This shows that the order in which the property is formulated can influence performance.

[23] In the course of this research, a new version (1.2.1) of MRMC was released, which resolves the limitation of 255 BSCCs

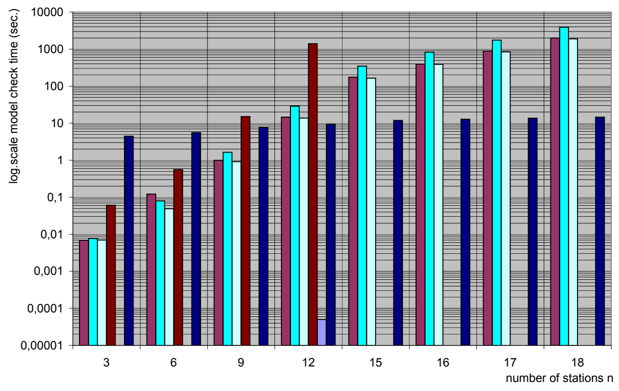
[24] In Figure 4.17 the MRMC values for some of the larger model sizes are missing, this is due to the fact that MRMC suffered a timer overflow, i. e. the variable used by MRMC for keeping track of model check time is not large enough.



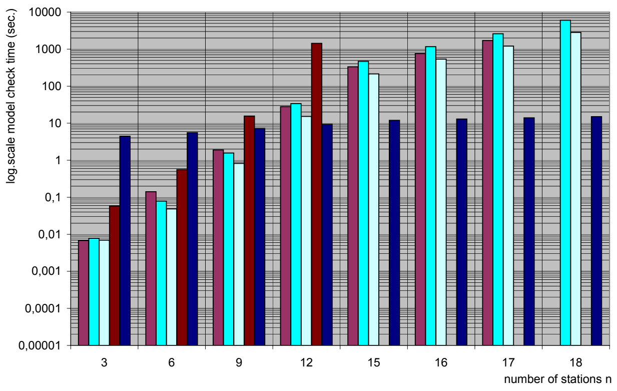
(a) $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq 5}\ poll_1]$



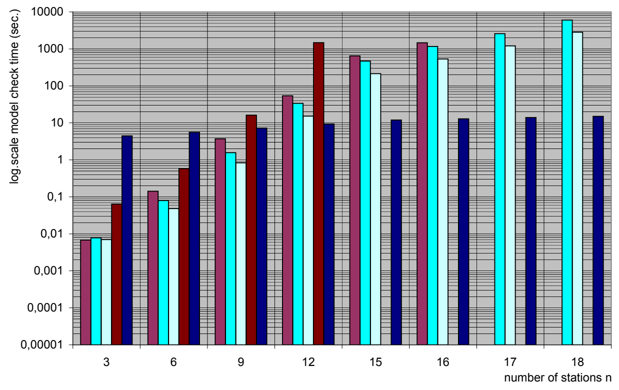
(b) $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq 10}\ poll_1]$



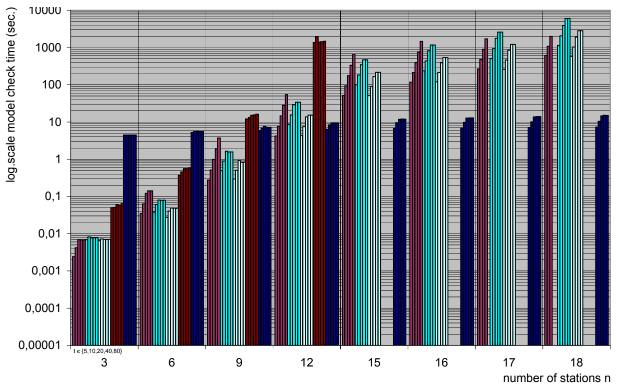
(c) $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq 20}\ poll_1]$



(d) $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq 40}\ poll_1]$



(e) $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq 80}\ poll_1]$



(f) summary of Figure 4.17 a through e

Figure 4.17: CSP - bounded $\mathcal{U}$, model check times

When we look at the model check times of the unbounded $\mathcal{U}$ operator, shown in Figure 4.18, we see that PRISM$^H$ and PRISM$^S$ are equally matched. For larger models ($n \geq 12$) the PRISM engines easily outperform MRMC and VESTA, e.g. where PRISM needs 0.83 seconds to verify the largest model (n=18), MRMC requires $\approx$ 3.8 seconds and VESTA $\approx$ 51.2 seconds.

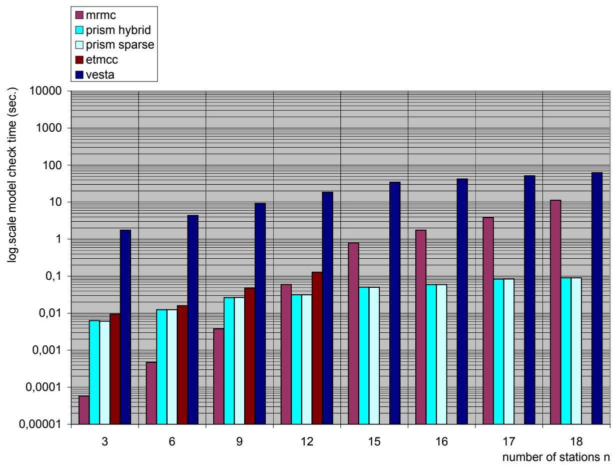
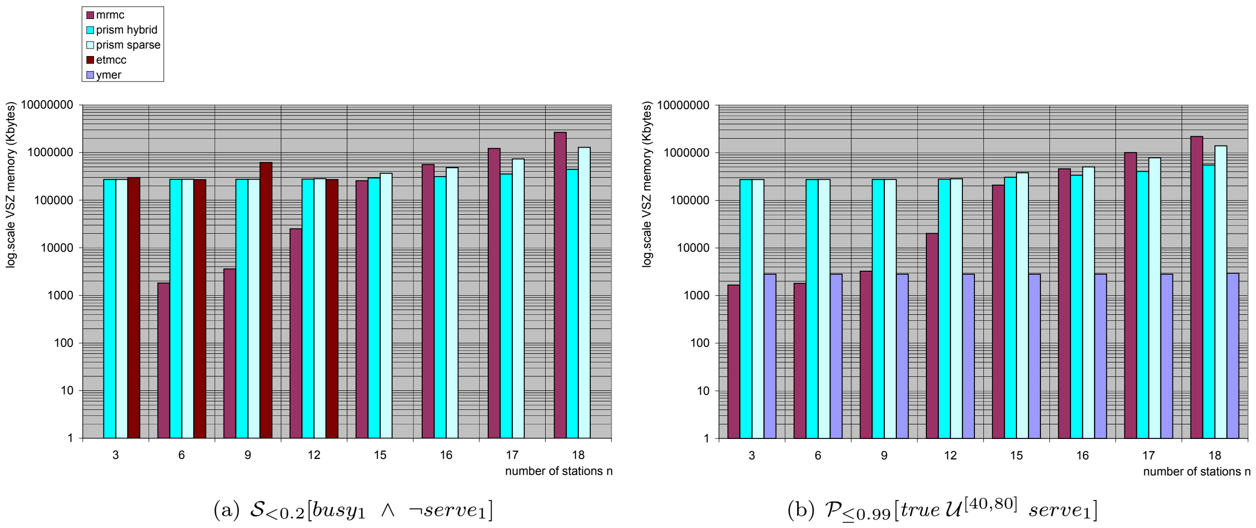


Figure 4.18: CSP - model check times - $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\, \mathcal{U}\; poll_1]$

We have one property remaining, namely $\neg(poll_1 \wedge poll_2)$. Model checking this property is trivial, because it requires no probability computations. All of the tools manage to verify this property within 0.007 seconds, regardless of the model parameters.

**Memory usage.** Starting with the memory usage for the steady-state operator in Figure 4.19(a), we see a similar behaviour as observed in the Tandem Queueing Network Figure 4.14(a). The difference between the memory consumption of two PRISM engines increases as the model size grows and, as expected, the sparse engine requires the most memory (but outperforms the hybrid engine in model check time). MRMC uses the least memory for models with $n \leq 15$ stations, after which PRISM$^H$ is more economic (at this point ETMCC has run out of memory).



(a) $\mathcal{S}_{<0.2}[busy_1 \;\wedge\; \neg serve_1]$

(b) $\mathcal{P}_{\leq 0.99}[true\, \mathcal{U}^{[40,80]}\; serve_1]$

Figure 4.19: CSP - Steady-state and interval Until, peak VSZ memory

Verification of the steady-state property on the largest model size ($n = 18$) resulted in a peak memory consumption of approximately 433 MB, 1255 MB and 2596 MB for PRISM$^H$, PRISM$^S$ and MRMC respectively. The peak memory for verification of the interval until property, see Figure 4.19(b), has similar behaviour as described above. The turning point for MRMC is $n = 15$, after which PRISM$^H$ takes over as the numerical tool that uses the least memory. YMER is the overall winner if we consider model sizes $n \geq 9$.

The peak memory for the property $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\,poll_1]$ is presented in Figure 4.20. We combined the results for the various time bounds $t \in \{5, 10, 20, 40, 80\}$ into a single bar chart in order to illustrate the fact that an increase in time bound does not seem to affect the peak memory consumption of the tools (whereas it does affect the model check time, as we have seen in Figure 4.17(f). As the model size increases, we see a growth in memory consumption for all numerical tools. Compared to PRISM, MRMC uses the least memory for $n \leq 15$, after which PRISM$^H$ takes over. ETMCC can verify models up to $n \leq 12$, after which it runs out of memory. The memory usage of VESTA remains constant during model growth. As for YMER; we were unable to measure its peak memory usage for this property. This is due to the fact that YMER verified the property within such a short time, that an accurate measurement was not possible. But we believe it is safe to assume that the peak memory consumption does not exceed that of any of the other tools.

Figures 4.21(a) and 4.21(b) show the peak memory usage during verification of the properties $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\,\mathcal{U}\,poll_1]$ and $\neg(poll_1 \wedge poll_2)$. The situation does not differ much from what we have previously seen, namely MRMC uses less memory than PRISM for smaller model sizes ($n \leq 15$). For these particular properties, there seems to be no difference between the peak memory usage of both prism engines.

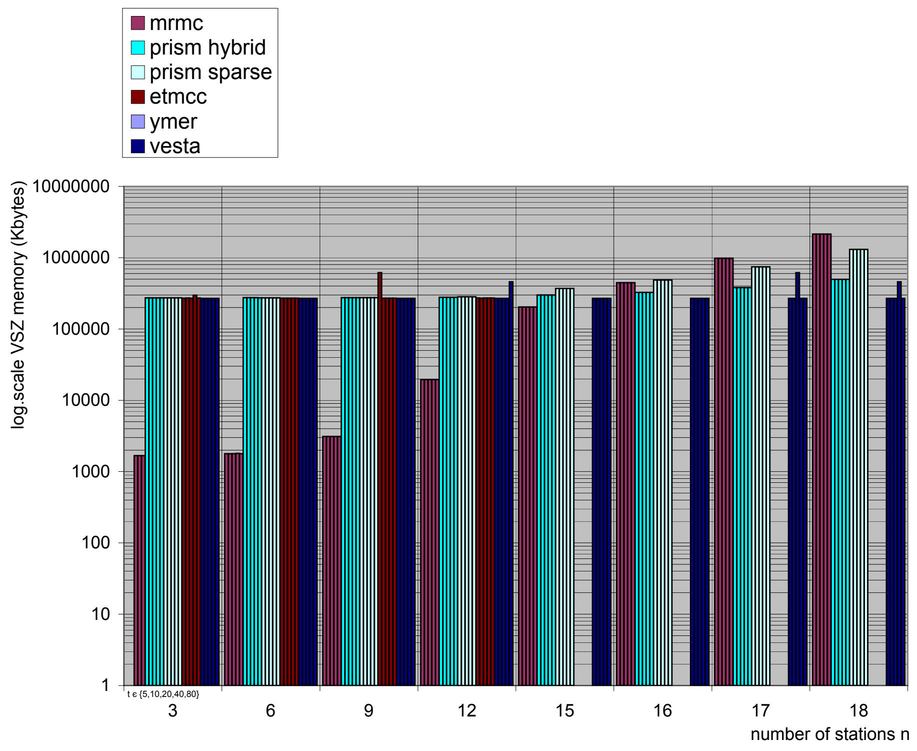
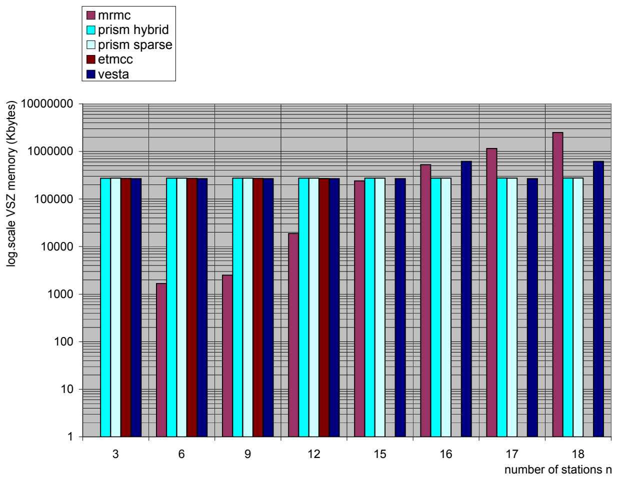
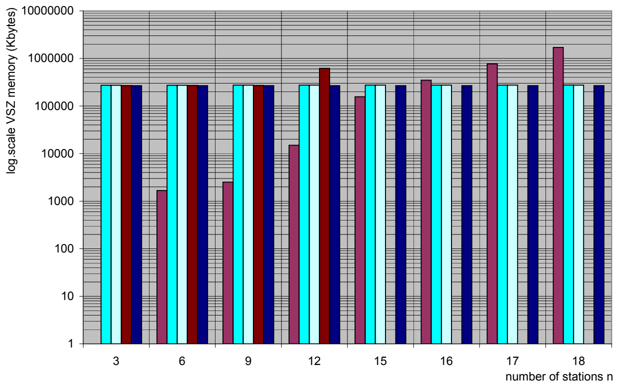


Figure 4.20: CSP - peak VSZ memory - $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\,\mathcal{U}^{\leq t}\,poll_1]$, with $t \in \{5, 10, 20, 40, 80\}$



(a) $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\, \mathcal{U}\ poll_1]$



(b) $\neg(poll_1 \wedge poll_2)$

Figure 4.21: CSP - peak VSZ memory

**Verification results.** The results produced by all numerical tools (MRMC, PRISM$^H$, PRISM$^S$, and ETMCC) are identical. Examining the statistical tools showed VESTA to be less reliable in verifying the properties $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 5}\ poll_1]$ and $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 10}\ poll_1]$ on the larger model sizes. Verification of the first property resulted in 87.5% of the answers being wrong, for the latter property the percentage of wrong answers was 62.5% (out of 160 measurements). YMER provided correct answers for all of its verified properties, except for $\mathcal{P}_{\leq 0.99}[true\, \mathcal{U}^{[40,80]}\ serve_1]$. For some model sizes, the actual probability proved to be close to the $\leq 0.99$ bound. Verification of this particular property, for which we took 160 measurements (i. e. 20 per model parameter), resulted in 3.13 % wrong answers. Out of the total number of measurements taken in this case study by each of the statistical tools, which is 960 measurements, we can state that VESTA had 25% and YMER 0.52% wrong.

## 4.4 Analysis

In Section 4.3 we have done a case by case analysis of the difference in speed, memory usage and verification results between the tools. We now consider topics related to the overall timing and memory measurement.

So far we have only shown the averages and standard deviation of the data related to timing measurements and have not discussed the individual measurements. The variation between the individual runs (i. e. the standard deviation of the 20 repeated experiments) is usually very low ($< 1\%$), as can be witnessed in Appendix C. We did notice however that the first measurement (i. e. 1 of 20) often exhibits a slower model check time than the 19 measurements that follow. This is believed to be caused by disk caching. After the first experiment the Operating System (OS) will have stored the requested data in the cache, when the second experiment starts it requires the same data, which at this time can be read from the cache, resulting in faster access. We restarted the tools before each experiment to prevent the tools itself from reusing results obtained in previous experiments. This however does not prevent the OS from caching files that are (often) used.

In Section 4.3 we demonstrated the difference in tool performance and memory consumption, by interpreting a variety of charts. These interpretations were made by using common sense and visually inspecting the charts (and raw data). We believe that this method is admissible, because of the small standard deviation between the (20) repeated measurements and the fact that the differences between the tools are often substantial. There are methods available from the field of statistics, namely hypothesis testing [78], which can be applied to prove whether or not the difference between two sets of data is significant enough not to be a coincidence. For completeness we apply a statistical significance test called the $t$ Test to one of our data sets, see Table 4.10. The data represents the model check times of the PRISM$^H$ and PRISM$^S$ engines for verification of the property $\mathcal{P}_{\leq 0.5}[true\ \mathcal{U}^{\leq 10}\ fst]$ on the Tandem Queuing Network with queue size $n = 100$. The $t$ Test tests the significance of the difference between two sample means, see [31] for details. The $t$ Test results presented in 4.10 are obtained using SPSS [3], a predictive analytic software package.

Table 4.10: Example statistical significance test on the Tandem Queuing Network case study

(a) Model check time samples , TQN, queue size 100, $\mathcal{P}_{\leq 0.5}[true\ \mathcal{U}^{\leq 10}\ fst]$

| Measurement | **Model check time (sec.)** | |
|---|---|---|
| | *PRISM*$^H$ | *PRISM*$^S$ |
| *1* | 0.507 | 0.489 |
| *2* | 0.507 | 0.488 |
| *3* | 0.51 | 0.488 |
| *4* | 0.508 | 0.485 |
| *5* | 0.51 | 0.488 |
| *6* | 0.51 | 0.488 |
| *7* | 0.509 | 0.491 |
| *8* | 0.509 | 0.491 |
| *9* | 0.509 | 0.488 |
| *10* | 0.507 | 0.488 |
| *11* | 0.51 | 0.488 |
| *12* | 0.509 | 0.49 |
| *13* | 0.508 | 0.485 |
| *14* | 0.508 | 0.489 |
| *15* | 0.509 | 0.488 |
| *16* | 0.504 | 0.49 |
| *17* | 0.51 | 0.487 |
| *18* | 0.529 | 0.486 |
| *19* | 0.51 | 0.489 |
| *20* | 0.507 | 0.488 |
| *Mean* | $0.509 \pm 2$ | $0.4882 \pm 8$ |

(b) Results of $t$ Test

| $t$ Test for Equality of Means | | | | | | |
|---|---|---|---|---|---|---|
| $t$[a] | *df*[b] | *Sig.*[c] *(2-tailed)* | *Mean*[d] *difference* | *Std. Error*[e] *difference* | *95% confidence interval*[f] *of the difference* | |
| | | | | | lower | upper |
| -18.640 | 38 | 1.03E-20 | -0.0213 | 0.001143 | -0.023613 | -0.018987 |

[a] Statistic used for testing the null hypothesis that two population means are equal.
[b] Value associated with a test statistic that is used in determining the observed significance level.
[c] The probability of obtaining results as extreme as the one observed, and in either direction when the null hypothesis is true.
[d] The mean for one group minus the mean for the other group.
[e] The standard deviation of the sample differences.
[f] A range of values based on the paired difference. If the interval does not contain 0, the paired difference differs significantly from 0.

We will not go into details on every aspect of the $t$ Test results. The most important factor in the results is the 2-tailed[25] significance. If this value is less than 0.05 – in statistics, it is general practice to use the 5% level as significance boundary – we can reject the so called null hypothesis, which states: "there is no significant difference between the two group means". As mentioned earlier, and witnessed again in this example, the standard deviation between the collected samples is small, which makes comparing the mean values easier (if done by hand i. e. on intuition). Motivated by the small standard deviation, and in view of the $t$ Test results on two mean values that are relatively close together and yet still statistically significant, we believe it is safe to forgo the $t$ Test[26].

### 4.4.1 Analysis by probabilistic operator.

In the previous sections we presented the results per case study and property. Although it is interesting to see how each tool performs on the case studies, we are primarily interested in the overall performance of the tools in verifying the different probabilistic operators. We therefore group all performance data based on the probabilistic operators; steady-state $\mathcal{S}$, unbounded until $\mathcal{U}$, bounded until $\mathcal{U}^{\leq t}$, interval until $\mathcal{U}^{[t_1,t_2]}$, and nested properties. We opt to evaluate the overall performance per operator and therefore do not differentiate between PCTL and CSL properties. This gives us a greater data set to work with, in contrast with subdividing the grouped data. We are aware that the complexities for model checking e. g. interval until and bounded until for CSL and PCTL are different, but we look at them as time bounded reachability problems in general regardless of the underlying algorithms. Once grouped per operator we create graphs of the model check time and peak memory consumption of the tools for all case studies involving the specific operator. The results are shown in Figures 4.22 and 4.23. Note that the x-axis enumerates the case study performed with the specific type of property. The y-axis shows the (average) model check time or peak memory usage for each tool. The lines between the data points were merely added for better surveyability, they have no significance since the case studies are in 'random' order. We did however try to sort the data (in an increasing order) to make the graphs more readable.

**Model check time sorted per probabilistic operator.** Figure 4.22(a) demonstrates the performance of MRMC, PRISM$^H$, PRISM$^S$, and ETMCC on all experiments involving steady-state properties. It is clear that MRMC has the best performance, all of its data points lay below that of the other tools. The next best tool is PRISM using the sparse engine, which is faster than the hybrid engine. For those cases where ETMCC has not run out of memory, it is clearly the slowest tool.

Figure 4.22(b) shows the model check times for the bounded until property. For MRMC we see that for most of the experiments the data points are entangled with that of PRISM, meaning that sometimes MRMC is faster and sometimes PRISM is faster. In this situation, we conclude that there exists no significant difference between MRMC and PRISM$^S$, based on the available number of measurements. Studying the difference between PRISM$^H$ and PRISM$^S$, it is clear that verifying a bounded until property proceeds faster using the sparse engine. The sparse engine outperformed the hybrid engine in 79 situations, out of a total of 82. As for ETMCC, it performs the worst of all numerical tools. The statistical tool YMER is missing several data points. This is due to the fact that YMER does not support the PCTL bounded until operator (only CSL), also in several cases YMER reports to model check this property in 0 seconds, as was explained in Section 4.3.5 page 61. Nevertheless we can state that YMER performs better than any of the other tools, for instance case 43 shows YMER requiring an average of 0.9467 sec., whereas MRMC,

[25] 2-tailed means that we hypothesize that the difference between the groups of data may go in *either* direction (i. e. PRISM$^H$ may be faster than PRISM$^S$ or vice versa, we do not specify the direction just that there exists a significant difference).

[26] The decision to forgo the $t$ Test was based on a multitude of example runs, where each time the results matched our self-diagnosis.

PRISM$^S$, and PRISM$^H$ require 28.4 min., 16.4 min. and 25.5 min. respectively. VESTA displays an mediocre performance, in roughly half of the experiments it outperforms the numerical tools.

The model check times for the interval until property, see Figure 4.22(c), show that YMER has the overall best performance. The difference between both PRISM engines is clearly visible, namely PRISM$^S$ outperforms PRISM$^H$. The difference between MRMC and PRISM$^S$ is difficult to evaluate, in some cases MRMC is faster and in others PRISM$^S$. There exist no significant difference in either direction.

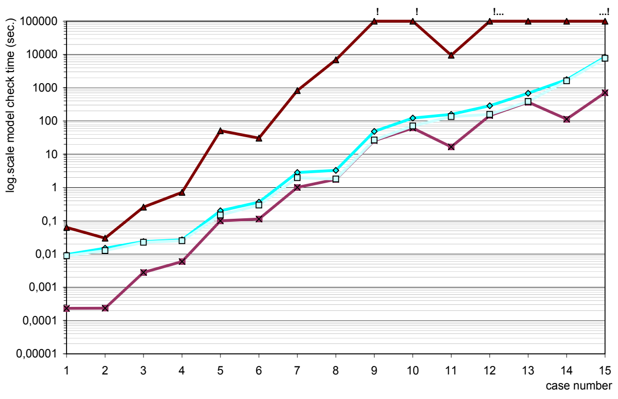
In Figure 4.22(d) we see the results for the unbounded until property. The results of both PRISM engines for this property are identical (that is to say within the standard deviation). As for MRMC, we see a mixed result. In the two most extreme cases (number 21 and 33) MRMC is 11.2 seconds slower and 194.9 seconds faster than PRISM. Because of the mixed results we can not conclude that either PRISM or MRMC is significantly faster for this property. However, the results do lean towards MRMC, since there are more data points (23 out of 33) where MRMC is faster than PRISM$^S$. If we examine ETMCC's performance we see that there are few available data points. This is due to the fact that this tool does not support DTMCs and sometimes runs out of memory. In those cases where verification succeeded, it frequently performed worse than the other numerical tools but still better than VESTA. VESTA has by far the worst performance when it comes to verifying the unbounded until property.

Figure 4.22(e) presents the performance on verification of the nested properties $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\,\mathcal{U}^{\leq 100}\ (n{=}70)]\,\mathcal{U}\ (n = 50)]$ and $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ \mathit{snd}]]$ for respectively the Birth-death and the Tandem Queuing Network case study. The results are unambiguous, MRMC outperforms both PRISM engines in all cases and the sparse engine is superior to the hybrid engine. ETMCC has the worst performance. There is no data available on VESTA, for the nested property in the Birth-death process it seemed to require an infinite amount of time and the experiment was aborted (see Section 4.3.3). The remaining nested property could not be verified with VESTA, since it does not support the steady-state operator. The same applies to YMER, which in addition has no support for the unbounded until operator (used in the Birth-death nested formula).

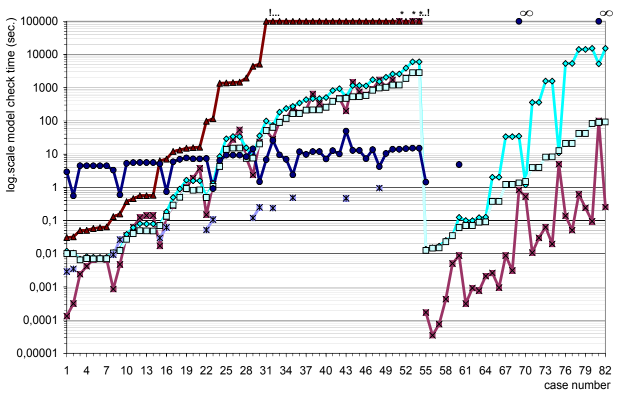
**Peak memory consumption sorted per probabilistic operator.** Figures 4.23(a) through 4.23(e) show the peak memory consumption of the tools during verification of the various probabilistic operators. In contrast with the model check time performance, the memory usage graphs assert a similar behaviour of the tools amongst the various properties. In general we can state that YMER is the best tool when is comes to memory consumption. It uses by far the least memory and has excellent scalability (i. e. we have seen in the individual case studies that YMERs memory consumption stays almost constant and appears independent of the model size). Unfortunately, YMER has a limited range of probabilistic operators at its disposal, which is apparent from the small amount of data points in the graphs. The other statistical tool in our test, VESTA, demonstrates a memory behaviour that far exceeds that of YMER, requiring up to 95 times as much memory for verification of the bounded until operator. Since YMER does not support the unbounded until operator, we can not compare YMER and VESTA on this property. Compared to the numerical tools we can state that VESTAs memory usage is fairly constant (i. e. the peak consumption does not differ greatly per case). For smaller model sizes VESTA requires more memory than the numerical tools, but as the models grow larger VETSA is more efficient and uses less memory than the numerical tools. Data for the remaining properties is not available. VESTA does not support the steady-state and bounded until property and verification of the nested properties was unsuccessful (they seemed to require an infinite amount of time and were aborted). If we compare MRMC and PRISM we see that in most cases, but not all, the (peak) memory consumption of MRMC is substantially lower than that of PRISM. As we know from the analysis in the individual case studies, MRMC has some problems with scalability. Although we can not see the model parameters in Figure 4.23, we know that for the largest model sizes MRMC often requires more memory than PRISM$^S$. Comparing the two prism engines learns that the hybrid engine is overall much more efficient (i. e. uses less memory) than the sparse engine. This applies to all properties, there are however a few situations, in particular for the bounded until properties, where the hybrid engines is less efficient than its sparse counterpart. This is believed to be caused by the fact that the memory efficiency of the (hybrid) MTBDD approach relies heavily on the structure and regularity of the model. This may lead to situations (e. g. poor regularity, many distinct values) where sparse matrix representation proves more efficient than MTBDDs. The last tool to be discussed is ETMMC. This tool displays a memory consumption similar to that of PRIMS, caused by the fact that they both use the JVM. However, ETMCC quickly runs out of memory. The tool produces the error: `java.lang.OutOfMemoryError:Java heap space"`, when trying to verify models with more than approximately 600,000 states.

- ⊠ mrmc
- ◆ prismh
- □ prisms
- ▲ etmcc
- ✳ ymer
- ● vesta

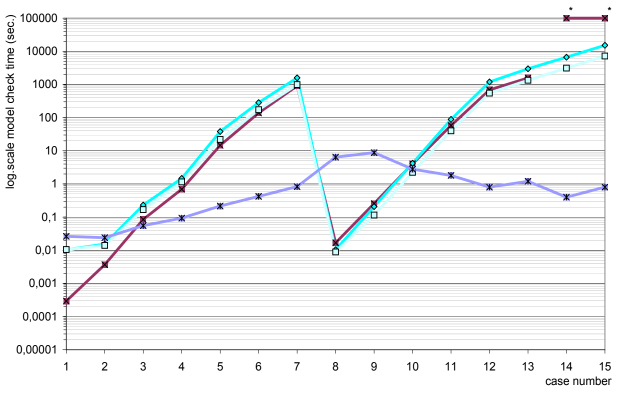
| | | |
|---|---|---|
| ? | ::= | *to fast to measure* |
| ! | ::= | *out of memory error* |
| ∞ | ::= | *run aborted after waiting 24H* |
| * | ::= | *timer overflow* |



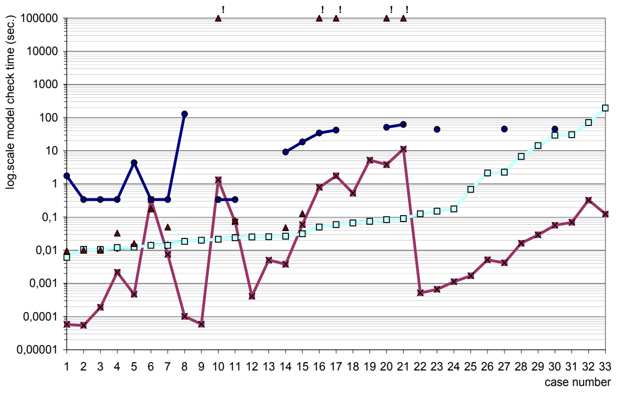
(a) steady-state



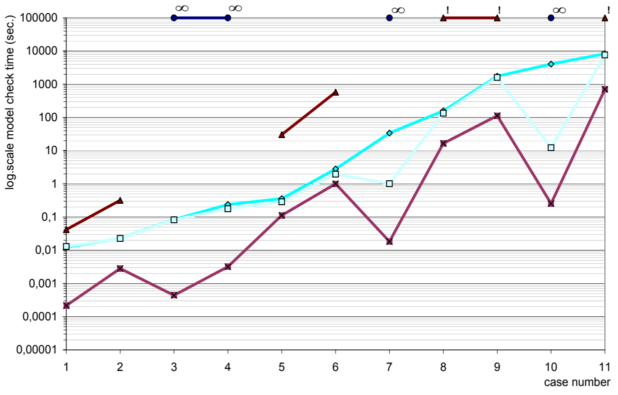
(b) bounded until



(c) interval until



(d) unbounded until

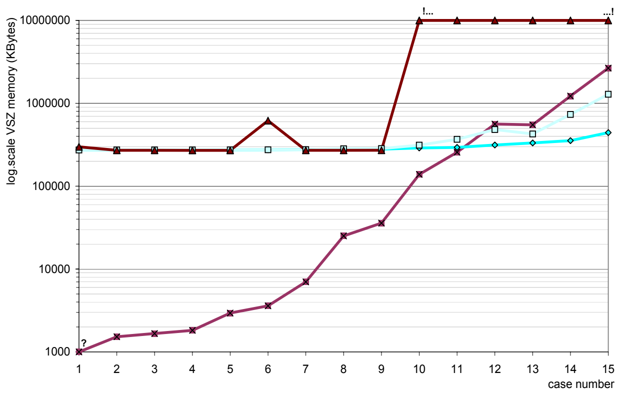


(e) nested

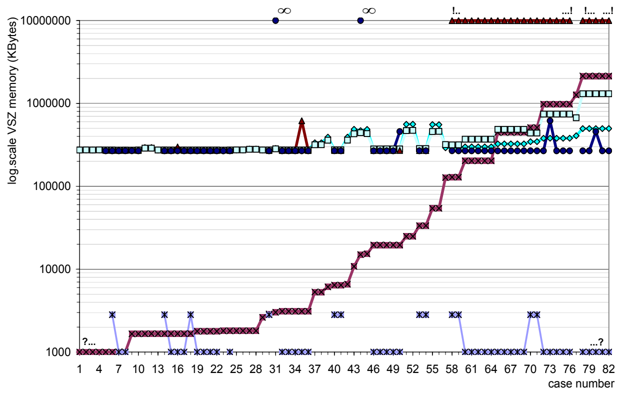
Figure 4.22: Model check times over all case studies (and model sizes) - arranged per property.

- ▨ mrmc
- ◇ prismh
- □ prisms
- ▲ etmcc
- ✳ ymer
- ● vesta

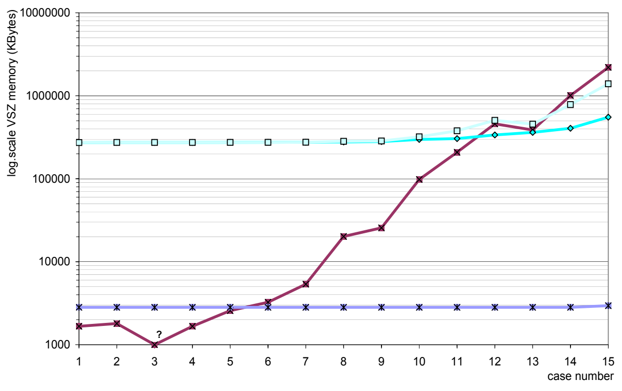
| | | |
|---|---|---|
| **?** | ::= | *to fast to measure* |
| **!** | ::= | *out of memory error* |
| $\infty$ | ::= | *run aborted after waiting 24H* |



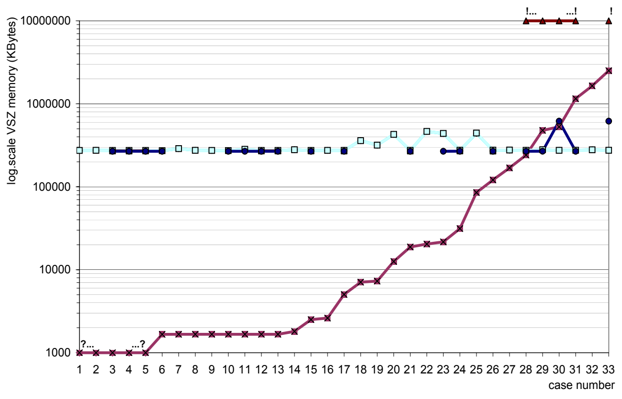
(a) steady-state



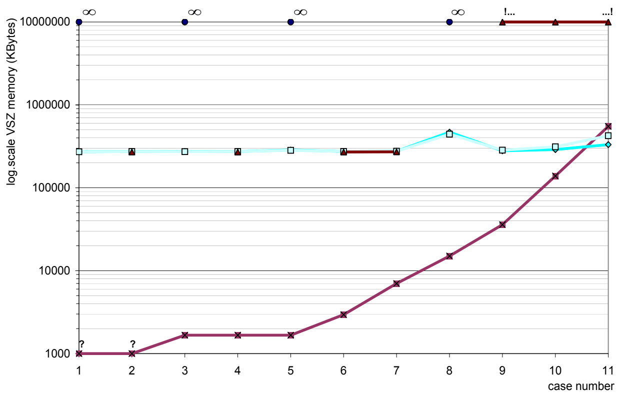
(b) bounded until



(c) interval until



(d) unbounded until



(e) nested

Figure 4.23: Peak VZS memory consumption over all case studies (and model sizes) - arranged per property.

### 4.4.2 Causes of performance differences

Up to now, the emphasis in this thesis has been on establishing whether or not (and to what extend) there exists a performance difference in speed and memory usage between the probabilistic model checker tools. From the preceding sections we know that differences exist and can be quite profound. The performance difference between the statistical and numerical tools is not surprising, and is caused by the different approach in model checking techniques i. e. sampling (from the initial state) versus numerical computations (for all states).

In the Leader Election case study we have seen a performance variance of several hours between the two top numerical tools MRMC and PRISM$^H$. Not much has been said on what might cause these differences. Although some performance difference between the tools might be caused by the difference in implementation language (C/C++ versus Java), we find it unlikely that this would be the sole cause, since both PRISM and MRMC have model checking engines implemented in C/C++. ETMCC on the other hand is fully developed in Java, which partially explains its poor performance. Since PRISM and MRMC are the top numerical tools in our study, we focus on uncovering the reasons for there performance differences.

In search for an explanation we focus on the PRISM tool and its inner workings. It is known that PRISM *always* builds an MTBDD, even with the sparse engine selected. In sparse mode, PRISM converts the MTBDD to a sparse matrix after performing some pre-computations[27]. This, along with the generation of the sparse matrix, may take a significant time and influence the model check time. On the other hand, there is no such influence for MRMC as it starts model checking on the pre-generated sparse matrix. Thus even if the model checking algorithms have the same performance the reported model check time for PRISM may be higher.

MTBDD size also plays a crucial role in PRISMs performance. If we take the Synchronous Leader Election case study as an example, we find that the considerable differences between PRISM and MRMC are due to the fact that the MTBDD is very large. It appears that the SLE protocol is an exceptionally bad example for symbolic methods i. e. the model is irregular resulting in a non-compact MTBDD. Naturally, this effect is most prominent on the larger model sizes, as witnessed in Table 4.11(a). It shows the relation between the state-space and the MTBDD size, for example the model $n = 4$, $k = 16$ contains 196,622 states (and 262,157 transitions), which requires 2.7 million MTBDD nodes. For the bounded until properties we witnessed model check times that differ up to several hours, e. g. for $n = 4$ and $k = 16$ we measured 0.25 sec for MRMC, 93 sec. for PRISM$^S$ and 4 hours for PRISM$^H$. The performance of the hybrid engine heavily depends on the size of the MTBDD, which as we have just seen is very large. This explains the poor performance of PRISM$^H$ on the SLE case study. The difference between MRMC and PRISM$^S$ is caused by the pre-computation step that is performed by PRISM on the MTBDD. The time consumed by the pre-computations is included in the model check time.

We have also experienced cases were memory usage can influence performance. If we look at the Cyclic Server Polling case study we note that using MTBDDs leads to a significant advantage in memory performance. As shown in Table 4.11(b), the CSP model can be represented as a highly compact MTBDD e. g. the model of 7 million states only requires 2,745 MTBDD nodes. If we study the average memory performance on verification of the largest model size ($n = 18$), we find that the memory usages for PRISM$^H$, PRISM$^S$, and MRMC are roughly 500MB, 1250 MB 2400MB, respectively, on our 2GB ram machine. Clearly the 2.4 GB, acquired by MRMC, does not fit in 2.0 GB RAM and thus swapping goes on, which is always a significant slow down. This is why MRMC may become considerably slower than both PRISM engines.

[27] In case of model checking time-unbounded until formula on DTMCs the pre-computations involve removing states that cannot ever reach the goal states.

Table 4.11: PRISM state-space and MTBDD size

(a) SLE case study

| n | k | #states | #trans. | #MTBDD nodes |
|---|---|---|---|---|
| 4 | 2 | 55 | 70 | 908 |
| | 4 | 782 | 1,037 | 10,801 |
| | 6 | 3,902 | 5,197 | 58,324 |
| | 8 | 12,302 | 16,397 | 165,625 |
| | 10 | 30,014 | 40,013 | 473,188 |
| | 12 | 62,222 | 82,957 | 929,667 |
| | 14 | 115,262 | 153,677 | 1,669,106 |
| | 16 | 196,622 | 262,157 | 2,762,663 |
| 8 | 2 | 1,803 | 2,058 | 7,857 |
| | 4 | 458,847 | 524,382 | 1,131,806 |

(b) CSP case study

| n | #states | #trans. | #MTBDD nodes |
|---|---|---|---|
| 3 | 36 | 84 | 112 |
| 6 | 572 | 2,208 | 367 |
| 9 | 6,912 | 36,864 | 765 |
| 12 | 73,728 | 503,808 | 1,282 |
| 15 | 737,280 | 6,144,000 | 1,942 |
| 16 | 1,572,864 | 13,893,632 | 2,188 |
| 17 | 3,342,336 | 31,195,136 | 2,469 |
| 18 | 7,077,888 | 69,599,232 | 2,745 |

**Reachability properties.** We have seen on more than one occasion that the PRISM hybrid and sparse engine have an identical performance in model check time as well as memory consumption. Looking back we can state that this happened in all cases where we verified an unbounded until property (see Figures 4.2(b),4.6(b),4.9(c), 4.13(d), 4.18). However, we learned that the indifference between the engines is not caused by the fact that we verify a property with the unbounded until operator. It is a result of the chosen probability bound $\bowtie p$ in $\mathcal{P}_{\bowtie p}[\Phi \ \mathcal{U} \ \Psi]$. In cases where the bound is extreme (i. e. either 0 or 1), PRISM will perform its computations solely on the MTBDD, even if the sparse engine is selected it will use the MTBDD without converting it to a sparse matrix. All our unbounded until properties happen to have an extreme probability bound, namely $\geq 1$. This explains why there is no performance difference between the sparse and hybrid engine for these properties.

We also noticed that VESTA has a rather poor performance for the properties with extreme bounds. The inefficiency of VESTA stems from the fact that it needs an excessive amount of sample paths to decide properties with bounds of the form $\geq 1$, as witnessed by e. g. , Tables B.4 and B.9. Generally, statistical tools have difficulties to decide whether the probability of some property meets a bound if the actual probability and the bound are close. Note that YMER does not support the unbounded until operator, and we could therefore not compare its performance to VESTA. As for ETMCC, its poor performance is caused by the less efficient sparse matrix representation.We do not have performance data on verification of unbounded until properties without extreme bounds, this would need to be investigated separately.

In the CPS case study we verified a time-bounded until property and observed what happens upon changing the time bound $t$. As expected, the memory usage is independent of $t$, in contrast to the model check time. The model check time required by MRMC is heavily influenced by $t$, e. g. , for $n$=15 the verification for $t$=20 is about four times longer than $t$=5. This is not surprising, as the time complexity of the underlying algorithm is linear in $t$, which is also true for ETMCC. Although this might not be apparent from our observation of the model check time graphs, the model check time for PRISM$^H$ and PRISM$^S$ is also linear in $t$. This fact is obscured by the initial overhead of the MTBDD construction, a careful analysis of the logfiles reveals that the time *per iteration* is independent from $t$. From $t$=30 on, the verification time is almost constant, due to a built-in steady-state detection [48]. The verification time for VESTA is rather constant for small $t$, this can be contributed to the fact that it collects the same amount of samples for different model sizes $n$, e. g. for $t = 5$ it needs $\approx$ 300,000 samples). Only for larger $t$ the amount of samples (and thus the model check time) increase slightly, e. g. for $t = 80$ the number of samples increase from 0.2M for $n = 3$ to 1.1M for $n = 18$. We have also used YMER to verify the time-bounded property and found its runtime to be extremely fast. This is caused by the fact that YMER instantly establishes that the initial state does not satisfy the premise of the implication $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq t}\ poll_1]$ and therefore stops. Like YMER, VESTA only checks the initial state (whereas the other tool check *all* states), but is unable to find the trivial satisfaction.

**Java, tools and memory consumption.** On increasing model parameters, the memory consumption of MRMC grows noticeably (as expected) whereas for PRISM$^S$ and PRISM$^H$ the memory usage seems constant for the smaller model ranges and only changes after reaching the larger model parameters. This is due to the fact that PRISM requires a large base memory consumption for; the JVM, the CUDD[28] package (around 40 MByte), and the MTBDD it generates from the input model. The large JVM base memory consumption is also the reason for the similarities in memory consumption between VESTA, PRISM and ETMCC.

We have witnessed on numerous occasions that ETMCC runs out of memory, it produces a "`java.lang.OutOfMemoryError:Java heap space`" error. Contrary to what one might expect, we did not detect an increase in ETMCCs VSZ memory consumption prior to the error. The VSZ memory usage remains fairly constant from the start of the tool, which is again caused by the fact that the JVM claims a large amount of base memory. We believe the large base memory chunk obscures the point where the fixed heap size limit is exceeded and the error is thrown.

[28] A explained in Chapter 3.1, the CUDD package [73] is a BDD/MTBDD library written in C.

## 4.5 User friendliness

The efficiency of a tool is not only defined by its speed and memory performance, we believe that user friendliness can be a contributing factor to the users perception of efficiency. A tool may perform fast computations, but if it offers a clumsy user-interface one would not be inclined to use it on a regular basis. During our experiments we have extensively used each of the probabilistic model checker tools and gained an insight in their user friendliness. Although it may seem difficult to measure user friendliness objectively, we aimed to present an honest overview.

As recognised by many people in the field, we find PRISM the most user friendly tool, having a reasonably powerful modelling language, simple installation instructions, a GUI and many additional features[29], such as the ability to plot the probability for different model parameter values. VESTA was less powerful in this respect, although it does have a nice GUI. The lack of a parallel composition operator, in its modelling language, on the other hand reduces it's usability, since one has to combine the various parallel components into a single model by hand, which is cumbersome and error-prone. Also, VESTAs supported syntax for model and property specification is not highly intuitive, the same applies to reported error message.

The syntax used by YMER strongly resembles that of PRISM, but not completely. Thus if one would want to model check a PRISM model with YMER (e. g. if a fast, but possible less accurate, response is desired), the model would have to be slightly transformed. This obviously also applies the other way around i. e. if more accuracy is required, one could verify YMER models using PRISM. Having no GUI, MRMC and YMER are less intuitive to use than VESTA and PRISM. On the other hand, MRMC is more appropriate as back-end verification engine as it has a simple input format. ETMCC offers a GUI, which simplifies the user interaction, but it offers no modelling language capabilities. Like MRMC, it works with pre-generated Markov chains and labellings. ETMCCs usefulness is limited due to the fact that for larger models it quickly runs out of memory.

In order to better asses the user friendliness, we decompose this aspect into three parts, namely:

1. *Ease of modelling.*
   This entails the supported modelling and property specification language (i. e. how hard is it to construct a model and specification accepted by the tool?)
2. *Ease of use.*
   This factor is mainly influence by the presence of a GUI. It also depends on available features, such as model export capabilities and the capability to adjust tool settings.
3. *Installation.*
   This relates to the difficulty in getting the tool up and running.

The tools are rated on their performance on each of the aspects listed above. Table 4.12 yields the results of our (informal) comparison. Here, ++ is the best, −− is the worst, and 0 is neutral.

Table 4.12: User friendliness assessment

| **user friendliness** | ETMCC | MRMC | PRISM | YMER | VESTA |
|---|---|---|---|---|---|
| ease of modelling | ++[a] | ++[a] | ++ | + | −− |
| ease of use | + | 0/+ | ++ | 0 | + |
| installation | + | 0/+ | ++ | −[b] | + |

++ is the best, −− is the worst, and 0 is neutral

[a] Exploiting the modelling facilities of PRISM.
[b] The required CUDD package files are not included in the distribution.

[29] The MRMC and PRISM projects are still active and new features are under development, new versions were released during creation of this thesis (MRMC[1], PRISM[2]).

# Chapter 5

# Conclusion

In this thesis we have studied five probabilistic model checker tools. Three of these utilise numerical techniques (PRISM$^{Sparse\text{ and }Hybrid}$, MRMC, ETMCC) and two (YMER, VESTA) are based on statistical methods (i. e. simulation and sampling). We made a tool by tool comparison, analysing model check times[1] and peak memory[2] usage. Using five[3] representative case studies of fully probabilistic systems, taken from the literature on performance evaluation and probabilistic model checking, we carefully constructed equivalent models for the individual tools and gathered performance data in an automated fashion in a controlled environment. We constructed multiple PCTL/CSL properties and realistic model sizes per case study, and verified each combination using the five tools, resulting in nearly 15,000 individual runs. Besides their performance, we also investigated the characteristics of each tool, comparing their implementation details, range of supported probabilistic models, model specification language, property specification language and supported algorithms and data structures. By ensuring that our experiments are repeatable, verifiable, statistically significant and free from external influences, our findings are based on a solid methodology.

From our experiments we learned that YMER is by far the best tool for verifying medium to large size models. It is the fastest tool and has a remarkably consistent (low) memory usage across various model sizes. Unfortunately YMER has a limited range of supported probabilistic operators at its disposal (no unbounded Until and steady-state operators), in addition it does not support discrete-time models (only CTMCs). Furthermore, being a statistical model checker YMER can not provide the same level of accuracy as can be achieved with numerical tools. The tool may report a wrong answer, and has done so during our experiments (in a few cases, as expected). We also learned that, at least for YMER, the order in which the CLS property is specified may influence performance. YMER outperforms the other statistical model checker VESTA. VESTA's memory consumption is also rather constant, but more in the order of PRISM's memory usage, which is mainly caused by the Java Virtual Machine (JVM). The model check time varies a lot. For certain nested properties, VESTA did not terminate within 24 h, even on a model with 100 states only. Overall we can state that, as expected, the statistical tools scale much better (performance wise) in relation to the state-space size than the numerical tools[4]. On the other hand, it is known [84] that for statistical tools, high accuracy comes at a greater price than for numerical tools. In addition, statistical tools have difficulties dealing with properties with a probability bound very close to the actual probability value and are not guaranteed to decide on probability bounds equal to the actual probability value.

Comparing the numerical tools we conclude that, as expected, PRISM$^S$ is usually faster than

[1] The time it takes a tool to verify a specific PCTL/CSL property.

[2] The maximum amount of memory consumed by the tool during its execution.

[3] The five case studies (of fully probabilistic systems) are; Synchronous Leader Election (SLE), Randomized Dining Philosophers (RDP), Birth-death process (BDP), Tandem Queuing Network (TQN) and Cyclic Server Polling System (CSP).

[4] Increasing the model size may increase the model check time by several orders of magnitude for the numerical tools, whilst the effect on statistical tools is minimal.

PRISM$^H$ at the cost of substantially greater memory usage. MRMC often outperforms PRISM$^S$ for models up to a few million states, this applies to the model check time as well as memory consumption and is especially true for steady-state and nested properties. The memory usage of PRISM is influenced by the overhead for MTBDD generation. On larger models, PRISM$^S$ and PRISM$^H$ perform better. This effect is more apparent whenever the MTBDD representation is compact. Because it uses MTBDDs, PRISM is able to check much larger models than the other numerical tools. For MRMC and ETMCC the model size is limited by the size of their model description files (i. e. *.tra* and *.lab*), which is bounded by 2GB due to system functions used for reading the files. The largest model size verified in our experiments contained 7 million states and approx. 69.5 million transitions. It is known that PRISM can handle models of e. g. 2,600 million states, provided that the model allows for a compact MTBDD representation, such as witnessed in the Cyclic Server Polling system case study. As for the remaining model checker tool, ETMCC, it has the worst performance. It is the slowest tool and it frequently runs out of memory in situations where the models could easily be checked by the other tools.

The conclusions regarding time performance have been based on the model check time only, we did not compare the tools on time spend on e. g. reading the input files, constructing the internal representation and writing results to file. The memory performance is based on the peak memory consumption during tool operation (i. e. from invocation to termination) and thus includes the model construction process. The results of the speed and memory performance of each tool are summarized in Table 5.1 and 5.2. We believe that in practice, one might often have to deal with model checking non trivial systems, which results in large state-space sizes. We therefore choose to let the model check times and memory performance on larger models weigh more heavily in our final evaluation.

Table 5.1: Speed performance

| **speed** | ETMCC | MRMC | PRISM$^S$ | PRISM$^H$ | YMER | VESTA |
|---|---|---|---|---|---|---|
| steady-state | − | ++ | + | 0/+[a] | N/A | N/A |
| bd. until | − | +[b] | +/++ | 0/+[a] | ++ | + |
| int. until | N/A | +/++ | +/++ | −[a] | ++ | N/A |
| unbd. until | − | +[b] | +/++ | +/++[a] | N/A | −/0 |
| nested | − | ++ | + | 0/+[a] | N/A[c] | −−[d] |

++ is the best, −− is the worst, 0 is neutral, and N/A means Not Applicable

[a] The time heavily depends on the MTBDD size.
[b] MRMC was faster in most cases, PRISM$^S$ on larger models.
[c] The property contained operators not supported by YMER.
[d] Based on one property, for which VESTA did not terminate.

Table 5.2: Memory performance

| **memory** | ETMCC | MRMC | PRISM$^S$ | PRISM$^H$ | YMER | VESTA |
|---|---|---|---|---|---|---|
| steady-state | − | +[a] | + | +/++[a b] | N/A | N/A |
| bd. until | − | +[a] | + | +/++[a b] | ++ | +[c] |
| int. until | N/A | +[a] | 0/+ | +/++[a b] | ++ | N/A |
| unbd. until | − | +[a] | +/++ | +/++[a b] | N/A | 0/+[c] |
| nested | − | +[a] | + | +/++[a b] | N/A | N/A[d] |

++ is the best, −− is the worst, 0 is neutral, and N/A means Not Applicable

[a] MRMC used least memory in most cases. For larger models PRISM$^S$ was between MRMC and PRISM$^H$ , and PRISM$^H$ was the best.
[b] The MTBDD size varied much with the case study.
[c] Fairly constant; inefficient for small models, efficient for large ones.
[d] Based on one property, for which VESTA did not terminate.

During our experiments we have extensively used each of the probabilistic model checker tools and gained an insight in their capabilities and user friendliness. We found a noticeable difference in the range of supported probabilistic operators between the tools and they lack a uniform model description language. The similarities between the tools are their support for the Linux platform and ability to model check Continuous Time Markov Chains (CTMCs), for which they all provide the bounded until operator. As recognised by many people in the field, we find PRISM the most user friendly tool, having a wide selection of probabilistic models and PCTL/CSL operators, an intuitive GUI and many additional features. MRMC is more appropriate as a fast back-end verification engine as it has a simple input format. In regards to the maximum verifiable model size, we discovered that, in addition to the RAM size, two factors restrict the model size: the size of the model description *.tra* files used by MRMC and ETMCC is limited to a maximum of 2 GB due to restrictions of the used system calls to write and read these files. In a few cases, we could not generate (and verify) our model as PRISM crashed due to a (known) problem of the CUDD package used for MTBDDs.

## 5.1 Recommendations

Based on our experience, we have the following suggestions for improving the tools. For YMER, it would be very useful if it supported more CSL/PCTL operators, so that its “slim and fast” engine becomes applicable to a wider class of model checking problems. Also, it would be nice for YMER to use exactly the same syntax as PRISM, improving the tool interoperability. For VESTA, we suggest to improve its running time. Also, its applicability would be enlarged by improving the modelling language by adding a parallel composition operator. For PRISM, a tight connection with YMER could be of relevance — ideally, a user would call the YMER model checker by pressing a single button. Besides the current model export functionally, it would be convenient if models could also be imported. This would eliminate model construction time. For MRMC, we suggest to improve the performance for larger models. For ETMCC, we do not provide any recommendations as it has been succeeded by MRMC.

### 5.1.1 Comparative research

On a more general note, for those who wish to perform a similar comparative research project we offer some wise words:

- Automation is the key word in this type of research. When performing experiments one generally needs to provide certain input parameters, which may or may not vary per experiment. Each experiment will generate a certain amount of data that often requires post processing e. g. filtering or some additional calculations. Automating this process as much as possible will not only ensure a consistent method of experiment execution, it also saves an enormous amount of time. Furthermore, if the process is automated it will make verification/duplication of the results feasible (especially for an outsider).
- In comparative research, it is essential that the environment (in which the items are compared) remains unchanged, except for the independent variables. The independent variable is the variable which is selected and manipulated by the experimenter to observe its effect on the dependent variable (i. e. the variable that is observed and measured) [31]. Also, ensure your comparison is fair and do not attempt to compare the incomparable (for example we made sure the verification parameters, such as the error bound $\epsilon$, of each tool were set to corresponding levels.
- Always think about the level of desired accuracy. It is pointless to present an average value with six decimal places, when the source data is only accurate to, for instance, two decimals. Repeat experiments to establish that obtained results were not due to mere chance.
- Know your statistics and sources of experimental errors. Make sure you calculate the right thing and that you calculate things right. For example, there are numerous statistical significance test available, whether or not a test is applicable depends on your data (e. g. is

the data divided into related groups? , is it continuous numerical data and if so can we assume that it is normally distributed?).

- Use reliable and multiple information sources. Different documentation on the same subject my reveal contradictive information.

We observed that there is little to none methodology available on how to set up an experimental research (e. g. tool comparison) project. This is in sheer contrast to domains such as physics and psychology, where there exist strict guidelines. On a basis level, we propose that a comparative research should at least meet the following three criteria:

1. *Repeatable/Verifiable.*
   A third party must be able to repeat any experiment and verify the results. This can be achieved by providing detailed documentation and by using e. g. open source models, openly available software and standard equipment. Automating the experiment process (e. g. by scripts) is also recommend.
2. *Statistically significant.*
   Gather a sufficient amount of data in order to make sure that your observations are not due to mere chance.
3. *Encapsulated.*
   With this we mean that experiments should be performed in an environment that is free from (unwanted) external influences.

## 5.2 Future Work

In this study we analysed the performance of five probabilistic model checker tools, naturally the study can be expanded to include more tools and even more diverse case studies. Besides DTMCs and CTMCs, it would be interesting to include reward models (which are supported by MRMC and the latest PRISM release). In our study we investigated the overall performance on verification of the different probabilistic operators. This can be augmented by investigating more details, such as differentiating between CSL and PCTL properties.

In our study we have mainly focussed on the model check time and peak memory consumption, there remains of course other interesting data that can be analyses, such as: *time per iteration*, *the total elapses time (including model construction* and *memory usage without the JVM effect.* Obviously, the *time per iteration* is not applicable for statistical tools.

We expect that, in the future, more tools will support distributed model checking, i. e. utilising multiple processor cores and/or multiple networked computers. A similar study as performed in this thesis may be repeated to include such tools. This will however require a different testing environment and new means of tracking model check time and memory consumption. From our selection of tools, YMER is the only one that is able to utilise multiple machines. It supports distributed acceptance sampling, i. e. the use of multiple machines to generate samples.

The statistical tools in this research always operated using fixed precision parameters (e. g. $\alpha$ and $\beta$), a different approach is to perform a series of experiment where these parameters vary (as done in for instance [84]).

On a final note we want to address the issue of significance testing. The field of statistics offers a wide variety of significance tests, such as the $t$ Test [31] and the Wilcoxon test [81]. Significance tests can detect whether or not there exist a significant difference between for instance the means of two data sets. These tests assume that data is gathered from some random population. Therefore, these test can not be applied on all aspects of our study, since we did *not* pick the case studies and model sizes randomly. When continuing our research, we believe it is prudent to investigate *if* and *when* statistical significance tests can be applied on the collected data.

# Appendices

# Appendix A

# Tool settings

This appendix lists the settings of each of the probabilistic model checker tools. These setting were not altered during the experiments. The table below displays the crucial settings, any parameter not displayed in the table can be assumed to have its default (tool distribution) value.

Table A.1: Tool settings

| **General settings num. tools: MRMC, ETMCC, PRISM** | |
|---|---|
| *Numerical computation alg.* | `Jacobi` |
| *Epsilon value for convergence check* | $10^{-6}$ |
| *Maximum number of iterations* | $10^{4}$ |
| *Steady-state detection* | `ON` |
| **Statistical tools: YMER, VESTA** | |
| $\alpha$ *(bound on false negative)* | `0.01` |
| $\beta$ *(bound on false positives)* | `0.01` |
| $\delta$ *indifference region* | `0.01` |
| Addition settings for YMER: | |
| sampling method (fixed number/seq.acc.sampling) | `seq.acc.sampling` |
| Addition settings for VESTA: | |
| *Maximum sample size* | $10^{8}$ |
| Settings related to unbounded until: | |
| *Stopping probability* | `0.05` |
| *Discount probability* | `1.0` |

# Appendix B

# Case studies: Model size and sample size

This appendix shows for each case study the model parameter(s) with the corresponding model size (i. e. number of states and transitions). When applicable it also lists the number of samples taken by the statistical tools. We use the following notations:

| | |
|---|---|
| *avg.*± *stdev.* | ::= average value over all runs in seconds ± the standard deviation in seconds. |
| - | ::= no value available (i. e. formula is not supported by the tool). |
| *n.s.* | ::= the tool has not specified the value. |
| ∞ | ::= tool has produced no answer after 24 hours of computation time. |

## B.1 DTMC

### B.1.1 Synchronous Leader Election

Table B.1: SLE - states and transitions

| n | k | #states | #transitions |
|---|---|---|---|
| 4 | 2 | 55 | 70 |
| | 6 | 3,902 | 5,197 |
| | 8 | 12,302 | 16,397 |
| | 10 | 30,014 | 40,013 |
| | 12 | 62,222 | 82,957 |
| | 14 | 115,262 | 153,677 |
| | 16 | 196,622 | 262,157 |
| 8 | 2 | 1,803 | 2,058 |
| | 4 | 458,847 | 524,382 |

### B.1.2 Randomized Dining Philosophers

Table B.2: Phil - states and transitions

| n | #states | #transitions |
|---|---|---|
| 3 | 770 | 2,845 |
| 4 | 10,022 | 47,432 |
| 6 | 594,790 | 4,170,946 |
| 7 | 5,454,562 | 44,070,594 |

### B.1.3 Birth-death process

Table B.3: Birth-death - states and transitions

| m | #states | #transitions |
|---:|---:|---:|
| 100 | 101 | 202 |
| 1,000 | 1,001 | 2,002 |
| 10,000 | 10,001 | 20,002 |
| 100,000 | 100,001 | 200,002 |

Table B.4: Birth-death - statistical tools #samples

| property | m | #samples |
|---|---|---|
| | | vesta |
| $\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq \frac{m}{2}}\ (n = \frac{m}{4})]$ | 100 | $143,349 \pm 35$ |
| | 1000 | $1,933,446 \pm 794$ |
| | 10000 | $\infty$ |
| | 100000 | $\infty$ |
| $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n = 50)]$ | 100 | $\infty$ |
| | 1000 | $\infty$ |
| | 10000 | $\infty$ |
| | 100000 | $\infty$ |
| $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ (n = m)]$ | 100 | $13,483,586 \pm 231,318$ |
| | 1000 | $4,674,346 \pm 5,113$ |
| | 10000 | $4,673,103 \pm 5,206$ |
| | 100000 | $4,673,632 \pm 5,105$ |

## B.2 CTMC

### B.2.1 Tandem Queuing Network

Table B.5: TQN - states and transitions

| n | #states | #transitions |
|---:|---:|---:|
| 2 | 15 | 33 |
| 10 | 231 | 729 |
| 50 | 5,151 | 17,649 |
| 100 | 20,301 | 70,299 |
| 255 | 130,816 | 455,939 |
| 511 | 523,776 | 1,829,379 |
| 1023 | 2,096,128 | 7,328,771 |

Table B.6: Tandem Queuing Network - statistical tools #samples

| **property** | **n** | **#samples** | |
|---|---|---|---|
| | | ymer | vesta |
| $\mathcal{P}_{<0.1}[\mathit{true}\,\mathcal{U}^{[0.5,2]}\ \mathit{full}]$ | 2 | $394 \pm 23$ | - |
| | 10 | 207.0 | - |
| | 50 | 207.0 | - |
| | 100 | 207.0 | - |
| | 255 | 207.0 | - |
| | 511 | 207.0 | - |
| | 1023 | 207.0 | - |
| $\mathcal{P}_{\leq 0.01}[\mathit{true}\,\mathcal{U}^{\leq 2}\ \mathit{full}]$ | 2 | $40 \pm 5$ | $15279 \pm 93$ |
| | 10 | 228.0 | $28300 \pm 80$ |
| | 50 | 228.0 | $81478 \pm 88$ |
| | 100 | 228.0 | $147815 \pm 79$ |
| | 255 | 228.0 | $353418 \pm 100$ |
| | 511 | 228.0 | $692832 \pm 102$ |
| | 1023 | 228.0 | $1371806 \pm 122$ |
| $\mathcal{P}_{\leq 0.5}[\mathit{true}\,\mathcal{U}^{\leq 10}\ \mathit{fst}]$ | 2 | 115.0 | $43802 \pm 98$ |
| | 10 | 115.0 | $186776 \pm 86$ |
| | 50 | 115.0 | $863870 \pm 86$ |
| | 100 | 115.0 | $1709112 \pm 77$ |
| | 255 | 115.0 | $4329504 \pm 88$ |
| | 511 | 115.0 | $8657521 \pm 71$ |
| | 1023 | 115.0 | $17313474 \pm 87$ |
| $\mathcal{P}_{\geq 1}[\mathit{snd}\,\mathcal{U}\ \mathit{sndn}]$ | 2 | - | 0 |
| | 10 | - | 0 |
| | 50 | - | 0 |
| | 100 | - | 0 |
| | 255 | - | 0 |
| | 511 | - | 0 |
| | 1023 | - | 0 |

### B.2.2 Cyclic Server Polling system

Table B.7: Polling - states and transitions

| **n** | **#states** | **#transitions** |
|---|---|---|
| 3 | 36 | 84 |
| 6 | 572 | 2,208 |
| 9 | 6,912 | 36,864 |
| 12 | 73,728 | 503,808 |
| 15 | 737,280 | 6,144,000 |
| 16 | 1,572,864 | 13,893,632 |
| 17 | 3,342,336 | 31,195,136 |
| 18 | 7,077,888 | 69,599,232 |

Table B.8: Cyclic Server Polling system - statistical tools #samples - part a

| **property** | **n** | **#samples** | |
|---|---|---|---|
| | | ymer | vesta |
| $\mathcal{P}_{\leq 0.99}[true\, \mathcal{U}^{[40,80]}\ serve_1]$ | 3 | 228.0 | - |
| | 6 | $207 \pm 23$ | - |
| | 9 | $48 \pm 29$ | - |
| | 12 | $24 \pm 10$ | - |
| | 15 | $10 \pm 9$ | - |
| | 16 | $14 \pm 5$ | - |
| | 17 | $4 \pm 2$ | - |
| | 18 | $8 \pm 3$ | - |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 5}\ poll_1]$ | 3 | n.s. | $160,493 \pm 154$ |
| | 6 | n.s. | $284,765 \pm 181$ |
| | 9 | n.s. | $324,589 \pm 307$ |
| | 12 | n.s. | $329,852 \pm 347$ |
| | 15 | n.s. | $330,506 \pm 320$ |
| | 16 | n.s. | $330,373 \pm 294$ |
| | 17 | n.s. | $330,294 \pm 346$ |
| | 18 | n.s. | $330,488 \pm 373$ |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 10}\ poll_1]$ | 3 | n.s. | $166,072 \pm 143$ |
| | 6 | n.s. | $348,730 \pm 222$ |
| | 9 | n.s. | $503,718 \pm 242$ |
| | 12 | n.s. | $596,501 \pm 277$ |
| | 15 | n.s. | $630,992 \pm 330$ |
| | 16 | n.s. | $634,901 \pm 504$ |
| | 17 | n.s. | $637,752 \pm 405$ |
| | 18 | n.s. | $639,204 \pm 313$ |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 20}\ poll_1]$ | 3 | n.s. | $166,087 \pm 144$ |
| | 6 | n.s. | $352,736 \pm 215$ |
| | 9 | n.s. | $539,353 \pm 253$ |
| | 12 | n.s. | $725,026 \pm 345$ |
| | 15 | n.s. | $901,620 \pm 341$ |
| | 16 | n.s. | $956,408 \pm 307$ |
| | 17 | n.s. | $1,007,298 \pm 380$ |
| | 18 | n.s. | $1,053,768 \pm 347$ |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 40}\ poll_1]$ | 3 | n.s. | $165,950 \pm 121$ |
| | 6 | n.s. | $352,783 \pm 260$ |
| | 9 | n.s. | $539,847 \pm 247$ |
| | 12 | n.s. | $726,546 \pm 294$ |
| | 15 | n.s. | $912,908 \pm 246$ |
| | 16 | n.s. | $975,716 \pm 332$ |
| | 17 | n.s. | $1,037,856 \pm 261$ |
| | 18 | n.s. | $1,100,436 \pm 359$ |

Table B.9: Cyclic Server Polling system - statistical tools #samples - part b

| **property** | **n** | **#samples** | |
|---|---|---|---|
| | | ymer | vesta |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\, \mathcal{U}^{\leq 80}\ poll_1]$ | 3 | n.s. | $165,972 \pm 114$ |
| | 6 | n.s. | $352,815 \pm 293$ |
| | 9 | n.s. | $539,563 \pm 295$ |
| | 12 | n.s. | $726,599 \pm 270$ |
| | 15 | n.s. | $912,938 \pm 407$ |
| | 16 | n.s. | $975,645 \pm 297$ |
| | 17 | n.s. | $1,037,376 \pm 543$ |
| | 18 | n.s. | $1,100,126 \pm 496$ |
| $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\, \mathcal{U}\ poll_1]$ | 3 | n.s. | $33,963 \pm 248$ |
| | 6 | n.s. | $150,300 \pm 775$ |
| | 9 | n.s. | $394,236 \pm 1,921$ |
| | 12 | n.s. | $841,445 \pm 4,574$ |
| | 15 | n.s. | $1,604,517 \pm 8,010$ |
| | 16 | n.s. | $1,956,313 \pm 9,307$ |
| | 17 | n.s. | $2,379,633 \pm 10,533$ |
| | 18 | n.s. | $2,871,492 \pm 10,863$ |
| $\neg(poll_1 \wedge poll_2)$ | 3 | n.s. | 0 |
| | 6 | n.s. | 0 |
| | 9 | n.s. | 0 |
| | 12 | n.s. | 0 |
| | 15 | n.s. | 0 |
| | 16 | n.s. | 0 |
| | 17 | n.s. | 0 |
| | 18 | n.s. | 0 |

# Appendix C

# Model check times

This appendix shows the model check times for each case study. We use the following notations:

| | | |
|---|---|---|
| *avg.± stdev.* | ::= | average time over all runs in seconds ± the standard deviation in seconds. |
| - | ::= | no time measurement available (i. e. formula is not supported by the tool). |
| *n.s.* | ::= | the tool has not specified the value. |

## C.1 Synchronous Leader Election

Table C.1: Synchronous Leader Election - model check time

| property | n | k | model check time (sec.) | | | additional measurements | |
|---|---|---|---|---|---|---|---|
| | | | mrmc | prism hybrid | prism sparse | prism hybrid | prism sparse |
| $\mathcal{P}_{\geq 0.85}[true\ \mathcal{U}^{\leq 5}\ leaderelected]$ | 4 | 2 | $0.0000348 \pm 13$ | $0.0150 \pm 3$ | $0.0147 \pm 6$ | | |
| | | 4 | $0.0003129 \pm 17$ | $0.0997 \pm 16$ | $0.0711 \pm 5$ | | |
| | | 6 | $0.000949 \pm 2$ | $2.016 \pm 8$ | $0.3808 \pm 19$ | | |
| | | 8 | $0.003111 \pm 15$ | $33.58 \pm 12$ | $1.209 \pm 16$ | | |
| | | 10 | $0.0107 \pm 3$ | $361.4 \pm 3$ | $3.89 \pm 7$ | | |
| | | 12 | $0.01966 \pm 9$ | $1583.5 \pm 16$ | $8.25 \pm 15$ | | |
| | | 14 | $0.05092 \pm 16$ | $5376 \pm 12$ | $20.9 \pm 4$ | | |
| | | 16 | $0.0946 \pm 15$ | $15299 \pm 10$ | $83 \pm 39$ | | |
| | 8 | 2 | $0.00077 \pm 4$ | $0.122 \pm 2$ | $0.0892 \pm 6$ | | |
| | | 4 | $0.2413 \pm 9$ | $14237 \pm 34$ | $41.7 \pm 14$ | $\mathcal{P}_{>0.99}[true\ \mathcal{U}^{\leq 400}\ leaderelected]$ | |
| $\mathcal{P}_{\geq 0.99}[true\ \mathcal{U}^{\leq 40}\ leaderelected]$ | 4 | 2 | $0.000076 \pm 2$ | $0.017 \pm 4$ | $0.01490 \pm 14$ | $0.0155 \pm 2$ | $0.0151 \pm 3$ |
| | | 4 | $0.000929 \pm 3$ | $0.0990 \pm 4$ | $0.0712 \pm 4$ | $0.1038 \pm 2$ | $0.0736 \pm 3$ |
| | | 6 | $0.002632 \pm 5$ | $2.018 \pm 9$ | $0.380 \pm 2$ | $2.14 \pm 4$ | $0.3946 \pm 13$ |
| | | 8 | $0.00878 \pm 4$ | $33.64 \pm 17$ | $1.208 \pm 13$ | $33.70 \pm 16$ | $1.250 \pm 12$ |
| | | 10 | $0.03018 \pm 11$ | $361.8 \pm 4$ | $3.94 \pm 7$ | $365.4 \pm 5$ | $4.00 \pm 7$ |
| | | 12 | $0.06290 \pm 16$ | $1585.3 \pm 18$ | $8.19 \pm 17$ | $1591.4 \pm 11$ | $8.67 \pm 14$ |
| | | 14 | $0.1391 \pm 4$ | $5377 \pm 36$ | $20.79 \pm 12$ | $5375 \pm 18$ | $21.7 \pm 2$ |
| | | 16 | $0.2534 \pm 13$ | $15297 \pm 77$ | $93.0 \pm 13$ | $15282 \pm 20$ | $92.5 \pm 15$ |
| | 8 | 2 | $0.00211 \pm 14$ | $0.1257 \pm 14$ | $0.0908 \pm 3$ | $0.1333 \pm 14$ | $0.0962 \pm 3$ |
| | | 4 | $0.6151 \pm 15$ | $14233 \pm 21$ | $41.7 \pm 6$ | $14242 \pm 128$ | $44.04 \pm 19$ |
| $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ leaderelected]$ | 4 | 2 | $0.000059 \pm 3$ | $0.0198 \pm 6$ | $0.02010 \pm 14$ | | |
| | | 4 | $0.000518 \pm 3$ | $0.1255 \pm 6$ | $0.1257 \pm 3$ | | |
| | | 6 | $0.001703 \pm 16$ | $0.685 \pm 3$ | $0.686 \pm 2$ | | |
| | | 8 | $0.00515 \pm 3$ | $2.13 \pm 2$ | $2.15 \pm 2$ | | |
| | | 10 | $0.0164 \pm 5$ | $6.77 \pm 17$ | $6.74 \pm 18$ | | |
| | | 12 | $0.02906 \pm 6$ | $13.92 \pm 15$ | $14.3 \pm 3$ | | |
| | | 14 | $0.0693 \pm 3$ | $30.2 \pm 14$ | $30.6 \pm 11$ | | |
| | | 16 | $0.1245 \pm 3$ | $193.1 \pm 10$ | $195 \pm 4$ | | |
| | 8 | 2 | $0.00113 \pm 8$ | $0.1760 \pm 9$ | $0.1772 \pm 5$ | | |
| | | 4 | $0.3264 \pm 6$ | $70.9 \pm 6$ | $71.2 \pm 5$ | | |

## C.2 Randomized Dining Philosophers

Table C.2: Randomized Dining Philosophers - model check time

| **property** | **n** | **model check time (sec.)** | | |
|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse |
| $\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq 20}\ \mathit{eat}]$ | 3 | $0.00043 \pm 3$ | $0.025 \pm 2$ | $0.0228 \pm 2$ |
| | 4 | $0.00507 \pm 3$ | $0.0347 \pm 4$ | $0.0347 \pm 2$ |
| | 6 | $0.5209 \pm 19$ | $1.185 \pm 5$ | $1.478 \pm 4$ |
| | 7 | $4.99 \pm 5$ | $10.89 \pm 2$ | $12.63 \pm 5$ |
| $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ \mathit{eat}]$ | 3 | $0.00041 \pm 5$ | $0.02510 \pm 14$ | $0.0252 \pm 2$ |
| | 4 | $0.005089 \pm 18$ | $0.0258 \pm 2$ | $0.0256 \pm 3$ |
| | 6 | $0.5268 \pm 12$ | $0.0669 \pm 2$ | $0.0668 \pm 2$ |
| | 7 | $5.261 \pm 19$ | $0.0745 \pm 4$ | $0.0744 \pm 5$ |

## C.3 Birth-death process

Table C.3: Birth-death process - model check time

| **property** | **m** | **model check time (sec.)** | | | |
|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | vesta |
| $\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq \frac{m}{2}}\ (n = \frac{m}{4})]$ | 100 | $0.000171 \pm 13$ | $0.0136 \pm 4$ | $0.0127 \pm 2$ | $1.4335 \pm 14$ |
| | 1000 | $0.0088 \pm 4$ | $0.1241 \pm 4$ | $0.0608 \pm 3$ | $4.831 \pm 3$ |
| | 10000 | $0.826 \pm 5$ | $34.84 \pm 15$ | $1.352 \pm 5$ | $\infty$[a] |
| | 100000 | $100.6 \pm 6$ | $5249 \pm 11$ | $90.0 \pm 6$ | $\infty$ |
| $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[\mathit{true}\ \mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n = 50)]$ | 100 | $0.000440 \pm 14$ | $0.0841 \pm 4$ | $0.0824 \pm 3$ | $\infty$ |
| | 1000 | $0.00320 \pm 17$ | $0.2389 \pm 9$ | $0.1774 \pm 6$ | $\infty$ |
| | 10000 | $0.01840 \pm 9$ | $33.92 \pm 13$ | $1.0114 \pm 19$ | $\infty$ |
| | 100000 | $0.2561 \pm 8$ | $4061 \pm 6$ | $12.24 \pm 4$ | $\infty$ |
| $\mathcal{P}_{\geq 1}[\mathit{true}\ \mathcal{U}\ (n = m)]$ | 100 | $0.000102 \pm 6$ | $0.01900 \pm 15$ | $0.0185 \pm 2$ | $128 \pm 2$ |
| | 1000 | $0.00066 \pm 3$ | $0.1509 \pm 5$ | $0.1510 \pm 10$ | $44.22 \pm 4$ |
| | 10000 | $0.004162 \pm 16$ | $2.298 \pm 8$ | $2.263 \pm 7$ | $45.23 \pm 6$ |
| | 100000 | $0.0567 \pm 5$ | $29.63 \pm 16$ | $29.4 \pm 4$ | $45.1 \pm 3$ |

[a] $\infty$ = experiment aborted, no result produces after continuously operating for $\approx$ 24 hours.

## C.4 Tandem Queuing Network

Table C.4: Tandem Queuing Network - model check time

| **property** | **n** | **model check time (sec.)** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $\mathcal{S}_{<0.01}[full]$ | 2 | 0.000235 ± 8 | 0.015 ± 4 | 0.0127 ± 3 | 0.03 | - | - |
| | 10 | 0.00280 ± 12 | 0.0236 ± 16 | 0.0225 ± 6 | 0.257 ± 5 | - | - |
| | 50 | 0.1125 ± 3 | 0.3631 ± 13 | 0.2948 ± 6 | 30.4 ± 2 | - | - |
| | 100 | 1.011 ± 8 | 2.815 ± 5 | 1.965 ± 7 | 826 ± 10 | - | - |
| | 255 | 16.64 ± 3 | 157.83 ± 12 | 135.25 ± 7 | 9509 ± 577 | - | - |
| | 511 | 114.2 ± 3 | 1746.5 ± 8 | 1606.4 ± 7 | `out mem.` | - | - |
| | 1023 | 708.7 ± 9 | 8311 ± 46 | 7681 ± 11 | `out mem.` | - | - |
| $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ snd]]$ | 2 | 0.000216 ± 8 | 0.0123 ± 2 | 0.0130 ± 4 | 0.042 ± 3 | - | - |
| | 10 | 0.00282 ± 8 | 0.02290 ± 14 | 0.0227 ± 4 | 0.32 | - | - |
| | 50 | 0.11288 ± 16 | 0.3592 ± 10 | 0.2883 ± 8 | 30.45 ± 5 | - | - |
| | 100 | 1.008 ± 4 | 2.813 ± 6 | 1.964 ± 5 | 579 ± 77 | - | - |
| | 255 | 16.521 ± 19 | 157.80 ± 8 | 135.23 ± 7 | `out mem.` | - | - |
| | 511 | 113.44 ± 11 | 1746.4 ± 5 | 1606.9 ± 4 | `out mem.` | - | - |
| | 1023 | 706.9 ± 7 | 8301 ± 10 | 7679 ± 17 | `out mem.` | - | - |
| $\mathcal{P}_{<0.1}[true\,\mathcal{U}^{[0.5,2]}\ full]$ | 2 | 0.000294 ± 14 | 0.0104 ± 2 | 0.0105 ± 4 | - | 0.0263 ± 11 | - |
| | 10 | 0.00369 ± 4 | 0.0154 ± 2 | 0.01400 ± 15 | - | 0.0240 ± 6 | - |
| | 50 | 0.0849 ± 2 | 0.2309 ± 6 | 0.1685 ± 4 | - | 0.0553 ± 3 | - |
| | 100 | 0.695 ± 4 | 1.467 ± 3 | 1.155 ± 7 | - | 0.0935 ± 3 | - |
| | 255 | 14.644 ± 13 | 38.18 ± 4 | 21.934 ± 19 | - | 0.2163 ± 7 | - |
| | 511 | 138.54 ± 14 | 284 ± 3 | 173.37 ± 14 | - | 0.4214 ± 9 | - |
| | 1023 | 911.1 ± 10 | 1583.8 ± 17 | 994.5 ± 4 | - | 0.832 ± 3 | - |
| $\mathcal{P}_{\leq 0.01}[true\,\mathcal{U}^{\leq 2}\ full]$ | 2 | 0.000315 ± 6 | 0.0105 ± 4 | 0.01000 ± 15 | 0.032 ± 3 | 0.0035 ± 5 | 0.5550 ± 6 |
| | 10 | 0.004772 ± 16 | 0.01320 ± 19 | 0.0126 ± 2 | 0.157 ± 2 | 0.0264 ± 2 | 0.5956 ± 7 |
| | 50 | 0.1324 ± 3 | 0.1823 ± 3 | 0.1348 ± 4 | 7.09 ± 16 | 0.0622 ± 3 | 0.7373 ± 6 |
| | 100 | 1.237 ± 6 | 1.298 ± 3 | 1.047 ± 5 | 116.2 ± 11 | 0.1064 ± 4 | 0.9127 ± 10 |
| | 255 | 23.25 ± 3 | 35.69 ± 4 | 20.220 ± 17 | 5161 ± 22 | 0.2506 ± 6 | 1.470 ± 3 |
| | 511 | 248.26 ± 16 | 273.1 ± 3 | 164.76 ± 18 | `out mem.` | 0.4835 ± 7 | 2.382 ± 3 |
| | 1023 | 1705.8 ± 8 | 1531 ± 5 | 983 ± 3 | `out mem.` | 0.9467 ± 17 | 4.208 ± 6 |
| $\mathcal{P}_{\leq 0.5}[true\,\mathcal{U}^{\leq 10}\ fst]$ | 2 | 0.000132 ± 5 | 0.012 ± 4 | 0.01010 ± 14 | 0.031 ± 2 | 0.00290 ± 14 | 2.920 ± 19 |
| | 10 | 0.00086 ± 4 | 0.0107 ± 2 | 0.0105 ± 2 | 0.131 ± 2 | 0.0093 ± 2 | 3.293 ± 4 |
| | 50 | 0.01732 ± 6 | 0.07285 ± 17 | 0.0701 ± 4 | 6.343 ± 4 | 0.0298 ± 2 | 5.098 ± 4 |
| | 100 | 0.1498 ± 8 | 0.509 ± 2 | 0.4882 ± 8 | 98.2 ± 6 | 0.0519 ± 2 | 7.367 ± 8 |
| | 255 | 2.353 ± 7 | 8.377 ± 7 | 7.588 ± 7 | 4440 ± 13 | 0.1195 ± 4 | 14.458 ± 13 |
| | 511 | 28.42 ± 4 | 72.80 ± 10 | 64.51 ± 2 | `out mem.` | 0.2366 ± 11 | 26.027 ± 13 |
| | 1023 | 198.37 ± 19 | 529.4 ± 12 | 477.8 ± 10 | `out mem.` | 0.4675 ± 13 | 49.18 ± 4 |
| $\mathcal{P}_{\geq 1}[snd\,\mathcal{U}\ sndn]$ | 2 | 0.000054 ± 3 | 0.0105 ± 2 | 0.0106 ± 2 | 0.01 | - | 0.3389 ± 5 |
| | 10 | 0.000190 ± 3 | 0.01080 ± 19 | 0.0105 ± 2 | 0.01 | - | 0.3388 ± 4 |
| | 50 | 0.002167 ± 5 | 0.0113 ± 2 | 0.0121 ± 13 | 0.033 ± 3 | - | 0.3385 ± 3 |
| | 100 | 0.00750 ± 15 | 0.0148 ± 6 | 0.0140 ± 7 | 0.050 ± 3 | - | 0.3387 ± 4 |
| | 255 | 0.0715 ± 8 | 0.0250 ± 9 | 0.0239 ± 8 | 0.078 ± 3 | - | 0.3381 ± 4 |
| | 511 | 0.3395 ± 18 | 0.0144 ± 5 | 0.0141 ± 7 | 0.177 ± 2 | - | 0.3386 ± 4 |
| | 1023 | 1.334 ± 4 | 0.0214 ± 6 | 0.0215 ± 6 | `out mem.` | - | 0.3384 ± 4 |

## C.5 Cyclic Server Polling system

Table C.5: Cyclic Server Polling system - model check time - part a

| **property** | **n** | **model check time (sec.)** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $\mathcal{S}_{<0.2}[busy_1\ \mathcal{U}\ serve_1]$ | 3 | 0.000233 $\pm$ 11 | 0.0096 $\pm$ 2 | 0.00885 $\pm$ 17 | 0.063 $\pm$ 6 | - | - |
| | 6 | 0.00594 $\pm$ 18 | 0.0268 $\pm$ 4 | 0.0251 $\pm$ 4 | 0.716 $\pm$ 7 | - | - |
| | 9 | 0.0994 $\pm$ 5 | 0.1996 $\pm$ 6 | 0.1490 $\pm$ 5 | 51.20 $\pm$ 14 | - | - |
| | 12 | 1.761 $\pm$ 5 | 3.274 $\pm$ 11 | 1.807 $\pm$ 3 | 6908 $\pm$ 157 | - | - |
| | 15 | 25.72 $\pm$ 4 | 48.8 $\pm$ 3 | 26.75 $\pm$ 3 | `out mem.` | - | - |
| | 16 | 61.26 $\pm$ 7 | 122.7 $\pm$ 4 | 70.89 $\pm$ 13 | `out mem.` | - | - |
| | 17 | 145.8 $\pm$ 10 | 287 $\pm$ 6 | 158.2 $\pm$ 6 | `out mem.` | - | - |
| | 18 | 369 $\pm$ 2 | 680 $\pm$ 15 | 385.0 $\pm$ 16 | `out mem.` | - | - |
| $\mathcal{P}_{\leq 0.99}[true\ \mathcal{U}^{[40,80]}\ serve_1]$ | 3 | 0.0168 $\pm$ 7 | 0.0104 $\pm$ 2 | 0.00890 $\pm$ 14 | - | 6.44 $\pm$ 5 | - |
| | 6 | 0.2568 $\pm$ 7 | 0.2078 $\pm$ 7 | 0.1155 $\pm$ 12 | - | 8.8 $\pm$ 10 | - |
| | 9 | 4.051 $\pm$ 16 | 4.216 $\pm$ 11 | 2.240 $\pm$ 11 | - | 2.8 $\pm$ 16 | - |
| | 12 | 57.93 $\pm$ 4 | 88.73 $\pm$ 16 | 40.06 $\pm$ 13 | - | 1.8 $\pm$ 7 | - |
| | 15 | 688.2 $\pm$ 6 | 1196.4 $\pm$ 15 | 548.2 $\pm$ 5 | - | 0.8 $\pm$ 8 | - |
| | 16 | 1574 $\pm$ 2 | 2979 $\pm$ 9 | 1339.5 $\pm$ 14 | - | 1.2 $\pm$ 4 | - |
| | 17 | `timer overfl.` | 6644 $\pm$ 40 | 3115 $\pm$ 12 | - | 0.4 $\pm$ 2 | - |
| | 18 | `timer overfl.` | 15207 $\pm$ 177 | 7208 $\pm$ 31 | - | 0.8 $\pm$ 3 | - |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 5}\ poll_1]$ | 3 | 0.00242 $\pm$ 11 | 0.0069 $\pm$ 2 | 0.0065 $\pm$ 2 | 0.05 | 0.0 | 4.47 $\pm$ 9 |
| | 6 | 0.0350 $\pm$ 4 | 0.0383 $\pm$ 3 | 0.0273 $\pm$ 5 | 0.3695 $\pm$ 10 | 0.0 | 5.254 $\pm$ 6 |
| | 9 | 0.2748 $\pm$ 16 | 0.4955 $\pm$ 16 | 0.2930 $\pm$ 14 | 12.095 $\pm$ 10 | 0.0 | 5.872 $\pm$ 5 |
| | 12 | 4.223 $\pm$ 10 | 8.479 $\pm$ 14 | 4.291 $\pm$ 8 | 1367 $\pm$ 6 | 0.00005 $\pm$ 10 | 6.372 $\pm$ 6 |
| | 15 | 51.43 $\pm$ 14 | 100.02 $\pm$ 19 | 50.83 $\pm$ 7 | `out mem.` | 0.0 | 6.833 $\pm$ 6 |
| | 16 | 117.37 $\pm$ 19 | 237.4 $\pm$ 5 | 119.43 $\pm$ 17 | `out mem.` | 0.0 | 6.975 $\pm$ 6 |
| | 17 | 266.2 $\pm$ 3 | 506 $\pm$ 7 | 259.6 $\pm$ 11 | `out mem.` | 0.0 | 7.127 $\pm$ 8 |
| | 18 | 606 $\pm$ 11 | 1133 $\pm$ 7 | 579 $\pm$ 3 | `out mem.` | 0.0 | 7.279 $\pm$ 9 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 10}\ poll_1]$ | 3 | 0.00421 $\pm$ 19 | 0.0082 $\pm$ 9 | 0.00720 $\pm$ 19 | 0.0510 $\pm$ 14 | 0.0 | 4.461 $\pm$ 16 |
| | 6 | 0.0631 $\pm$ 5 | 0.06095 $\pm$ 18 | 0.0405 $\pm$ 3 | 0.457 $\pm$ 18 | 0.0 | 5.543 $\pm$ 10 |
| | 9 | 0.512 $\pm$ 3 | 0.892 $\pm$ 4 | 0.504 $\pm$ 2 | 13.3 $\pm$ 2 | 0.0 | 6.966 $\pm$ 9 |
| | 12 | 7.656 $\pm$ 18 | 15.49 $\pm$ 3 | 7.51 $\pm$ 2 | 1960 $\pm$ 144 | 0.0 | 8.353 $\pm$ 6 |
| | 15 | 93.46 $\pm$ 14 | 183.5 $\pm$ 3 | 89.90 $\pm$ 13 | `out mem.` | 0.0 | 9.484 $\pm$ 10 |
| | 16 | 213.7 $\pm$ 4 | 437.3 $\pm$ 7 | 212.2 $\pm$ 3 | `out mem.` | 0.0 | 9.783 $\pm$ 11 |
| | 17 | 476.0 $\pm$ 8 | 933 $\pm$ 2 | 461 $\pm$ 2 | `out mem.` | 0.0 | 10.123 $\pm$ 8 |
| | 18 | 1073.4 $\pm$ 19 | 2063 $\pm$ 32 | 1030 $\pm$ 4 | `out mem.` | 0.0 | 10.46 $\pm$ 8 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 20}\ poll_1]$ | 3 | 0.0069 $\pm$ 3 | 0.0077 $\pm$ 2 | 0.0070 | 0.061 $\pm$ 5 | 0.0 | 4.451 $\pm$ 6 |
| | 6 | 0.1217 $\pm$ 8 | 0.0801 $\pm$ 10 | 0.0488 $\pm$ 4 | 0.5580 $\pm$ 19 | 0.0 | 5.58 $\pm$ 2 |
| | 9 | 0.993 $\pm$ 4 | 1.643 $\pm$ 10 | 0.919 $\pm$ 4 | 15.14 $\pm$ 11 | 0.0 | 7.7 $\pm$ 3 |
| | 12 | 14.54 $\pm$ 9 | 28.96 $\pm$ 4 | 13.71 $\pm$ 3 | 1397.1 $\pm$ 12 | 0.00005 $\pm$ 10 | 9.309 $\pm$ 5 |
| | 15 | 174.8 $\pm$ 2 | 345.1 $\pm$ 8 | 164.9 $\pm$ 3 | `out mem.` | 0.0 | 11.840 $\pm$ 10 |
| | 16 | 395.1 $\pm$ 7 | 821.9 $\pm$ 17 | 389.5 $\pm$ 6 | `out mem.` | 0.0 | 12.744 $\pm$ 10 |
| | 17 | 895.1 $\pm$ 8 | 1751 $\pm$ 4 | 845 $\pm$ 2 | `out mem.` | 0.0 | 13.667 $\pm$ 11 |
| | 18 | 1994 $\pm$ 3 | 3902 $\pm$ 86 | 1896 $\pm$ 15 | `out mem.` | 0.0 | 14.596 $\pm$ 10 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 40}\ poll_1]$ | 3 | 0.0068 $\pm$ 3 | 0.0078 $\pm$ 2 | 0.00695 $\pm$ 10 | 0.058 $\pm$ 2 | 0.0 | 4.456 $\pm$ 4 |
| | 6 | 0.1416 $\pm$ 8 | 0.0787 $\pm$ 8 | 0.0484 $\pm$ 6 | 0.559 $\pm$ 11 | 0.0 | 5.562 $\pm$ 6 |
| | 9 | 1.900 $\pm$ 8 | 1.572 $\pm$ 5 | 0.822 $\pm$ 5 | 15.60 $\pm$ 3 | 0.0 | 7.187 $\pm$ 6 |
| | 12 | 27.90 $\pm$ 4 | 33.81 $\pm$ 6 | 15.25 $\pm$ 6 | 1433 $\pm$ 22 | 0.0 | 9.318 $\pm$ 7 |
| | 15 | 333.4 $\pm$ 15 | 468.7 $\pm$ 15 | 214.9 $\pm$ 4 | `out mem.` | 0.0 | 11.933 $\pm$ 12 |
| | 16 | 753.2 $\pm$ 15 | 1168 $\pm$ 2 | 536.1 $\pm$ 15 | `out mem.` | 0.0 | 12.924 $\pm$ 11 |
| | 17 | 1710.1 $\pm$ 16 | 2591 $\pm$ 15 | 1206 $\pm$ 7 | `out mem.` | 0.0 | 13.988 $\pm$ 9 |
| | 18 | `timer overfl.` | 6019 $\pm$ 117 | 2828 $\pm$ 17 | `out mem.` | 0.0 | 15.082 $\pm$ 11 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 80}\ poll_1]$ | 3 | 0.0068 $\pm$ 3 | 0.0078 $\pm$ 4 | 0.0070 | 0.064 $\pm$ 5 | 0.0 | 4.454 $\pm$ 7 |
| | 6 | 0.1418 $\pm$ 5 | 0.0794 $\pm$ 3 | 0.0482 $\pm$ 5 | 0.58 $\pm$ 3 | 0.0 | 5.556 $\pm$ 5 |
| | 9 | 3.714 $\pm$ 18 | 1.573 $\pm$ 5 | 0.830 $\pm$ 6 | 16.2 $\pm$ 2 | 0.0 | 7.184 $\pm$ 7 |
| | 12 | 54.25 $\pm$ 6 | 33.76 $\pm$ 4 | 15.28 $\pm$ 4 | 1472 $\pm$ 4 | 0.0 | 9.320 $\pm$ 7 |
| | 15 | 645.3 $\pm$ 6 | 467.4 $\pm$ 11 | 214.6 $\pm$ 4 | `out mem.` | 0.0 | 11.932 $\pm$ 8 |
| | 16 | 1462 $\pm$ 3 | 1166 $\pm$ 2 | 533.5 $\pm$ 8 | `out mem.` | 0.0 | 12.920 $\pm$ 9 |
| | 17 | `timer overfl.` | 2589 $\pm$ 52 | 1208 $\pm$ 2 | `out mem.` | 0.0 | 13.994 $\pm$ 13 |
| | 18 | `timer overfl.` | 6032 $\pm$ 121 | 2829 $\pm$ 51 | `out mem.` | 0.0 | 15.075 $\pm$ 13 |

Table C.6: Cyclic Server Polling system - model check time - part b

| **property** | **n** | **model check time (sec.)** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\, \mathcal{U}\ poll_1]$ | 3 | $0.000058 \pm 2$ | $0.0063 \pm 2$ | $0.00610 \pm 14$ | $0.0095 \pm 10$ | - | $1.752 \pm 10$ |
| | 6 | $0.00047 \pm 2$ | $0.0124 \pm 6$ | $0.0124 \pm 4$ | $0.016 \pm 2$ | - | $4.37 \pm 2$ |
| | 9 | $0.003786 \pm 7$ | $0.0263 \pm 2$ | $0.0266 \pm 2$ | $0.0480 \pm 19$ | - | $9.22 \pm 4$ |
| | 12 | $0.0597 \pm 9$ | $0.0315 \pm 5$ | $0.0315 \pm 6$ | $0.128 \pm 11$ | - | $18.46 \pm 14$ |
| | 15 | $0.790 \pm 2$ | $0.0505 \pm 4$ | $0.0500 \pm 2$ | `out mem.` | - | $34.45 \pm 18$ |
| | 16 | $1.748 \pm 5$ | $0.0591 \pm 4$ | $0.0593 \pm 5$ | `out mem.` | - | $41.99 \pm 17$ |
| | 17 | $3.842 \pm 11$ | 0.083 | 0.084 | `out mem.` | - | $51.2 \pm 2$ |
| | 18 | $11.3 \pm 4$ | $0.090 \pm 3$ | $0.0903 \pm 14$ | `out mem.` | - | $62.3 \pm 3$ |
| $\neg(poll_1 \wedge poll_2)$ | 3 | n.s. | $0.007 \pm 3$ | $0.0053 \pm 2$ | 0.0 | $0.00005 \pm 10$ | $0.00005 \pm 10$ |
| | 6 | n.s. | $0.0048 \pm 2$ | $0.0047 \pm 3$ | 0.0 | 0.0 | $0.00015 \pm 17$ |
| | 9 | n.s. | $0.0047 \pm 2$ | $0.0044 \pm 2$ | 0.0 | 0.0 | 0.0 |
| | 12 | n.s. | $0.0047 \pm 2$ | $0.0047 \pm 2$ | $0.0010 \pm 14$ | 0.0 | $0.00005 \pm 10$ |
| | 15 | n.s. | $0.00480 \pm 19$ | $0.0048 \pm 3$ | `out mem.` | 0.0 | $0.00005 \pm 10$ |
| | 16 | n.s. | $0.0048 \pm 2$ | $0.00480 \pm 19$ | `out mem.` | 0.0 | 0.0 |
| | 17 | n.s. | $0.0055 \pm 2$ | $0.0055 \pm 2$ | `out mem.` | 0.0 | 0.0 |
| | 18 | n.s. | $0.0048 \pm 3$ | $0.0046 \pm 2$ | `out mem.` | 0.0 | $0.00005 \pm 10$ |

# Appendix D

# Peak Memory consumption

This appendix shows the peak VSZ (Virtual Memory Size) memory consumption (in Kilobytes) for each case study. We use the following notations:

| | | |
|---|---|---|
| - | ::= | no memory measurement available (i. e. formula is not supported by the tool). |
| *n.m.* | ::= | the value could not be measured (i. e. the model check time was to short for an accurate memory measurement). |

## D.1 Synchronous Leader Election

Table D.1: Synchronous Leader Election - peak VSZ memory

| | | | **VSZ memory in Kbytes** | | |
|---|---|---|---|---|---|
| **property** | **n** | **k** | mrmc | prism hybrid | prism sparse |
| $\mathcal{P}_{\geq 0.85}[true\ \mathcal{U}^{\leq 5}\ leaderelected]$ | 4 | 2 | *n.m.* | 273,980 | 274,092 |
| | | 4 | *n.m.* | 274,876 | 274,996 |
| | | 6 | 1,808 | 281,456 | 279,464 |
| | | 8 | 1,664 | 293,480 | 288,176 |
| | | 10 | 5,300 | 335,012 | 317,260 |
| | | 12 | 6,112 | 391,592 | 359,924 |
| | | 14 | 10,776 | 487,332 | 429,960 |
| | | 16 | 24,872 | 559,864 | 469,940 |
| | 8 | 2 | 1,804 | 274,924 | 274,908 |
| | | 4 | 54,180 | 555,968 | 457,628 |
| $\mathcal{P}_{\geq 0.99}[true\ \mathcal{U}^{\leq 40}\ leaderelected]$ | 4 | 2 | *n.m.* | 273,980 | 273,984 |
| | | 4 | *n.m.* | 274,876 | 274,972 |
| | | 6 | 1,808 | 281,456 | 279,596 |
| | | 8 | 1,668 | 293,588 | 288,080 |
| | | 10 | 5,300 | 334,416 | 317,520 |
| | | 12 | 6,600 | 391,592 | 360,236 |
| | | 14 | 15,288 | 485,440 | 432,432 |
| | | 16 | 24,872 | 561,740 | 472,536 |
| | 8 | 2 | 1,808 | 274,908 | 274,908 |
| | | 4 | 54,180 | 554,092 | 459,508 |
| $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ leaderelected]$ | 4 | 2 | *n.m.* | 273,984 | 273,960 |
| | | 4 | *n.m.* | 274,960 | 274,880 |
| | | 6 | 1,808 | 279,404 | 279,400 |
| | | 8 | 1,668 | 288,304 | 288,060 |
| | | 10 | 7,296 | 317,276 | 317,272 |
| | | 12 | 7,088 | 360,096 | 359,692 |
| | | 14 | 12,584 | 427,608 | 427,604 |
| | | 16 | 20,396 | 464,136 | 464,220 |
| | 8 | 2 | 1,668 | 274,912 | 274,908 |
| | | 4 | 85,120 | 441,724 | 444,068 |

## D.2 Randomized Dining Philosophers

Table D.2: Randomized Dining Philosophers - peak VSZ memory

| **property** | **n** | **VSZ memory in Kbytes** | | |
|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse |
| $\mathcal{P}_{\geq 0.9}[true\ \mathcal{U}^{\leq 20}\ eat]$ | 3 | $1,668$ | $274,112$ | $274,116$ |
| | 4 | $2,616$ | $274,700$ | $274,376$ |
| | 6 | $127,640$ | $291,608$ | $318,564$ |
| | 7 | $1,265,648$ | $407,576$ | $671,392$ |
| $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ eat]$ | 3 | $1,668$ | $274,204$ | $274,104$ |
| | 4 | $2,616$ | $274,380$ | $274,368$ |
| | 6 | $168,864$ | $276,776$ | $276,932$ |
| | 7 | $1,644,692$ | $279,376$ | $279,104$ |

## D.3 Birth-death process

Table D.3: Birth-death process - peak VSZ memory

| **property** | **m** | **VSZ memory in Kbytes** | | | |
|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | vesta |
| $\mathcal{P}_{\geq 0.9}[true\ \mathcal{U}^{\leq \frac{m}{2}}\ (n = \frac{m}{4})]$ | 100 | $1,664$ | $271,988$ | $272,100$ | $267,996$ |
| | 1000 | $1,800$ | $272,932$ | $272,808$ | $267,404$ |
| | 10000 | $3,020$ | $285,760$ | $283,132$ | - |
| | 100000 | $15,016$ | $468,704$ | $443,460$ | - |
| $\mathcal{P}_{\geq 1}[\mathcal{P}_{\geq 0.9}[true\ \mathcal{U}^{\leq 100}\ (n{=}70)]\ \mathcal{U}\ (n = 50)]$ | 100 | $n.m.$ | $271,988$ | $271,976$ | - |
| | 1000 | $1,668$ | $273,224$ | $272,952$ | - |
| | 10000 | $1,668$ | $285,468$ | $283,032$ | - |
| | 100000 | $15,016$ | $468,824$ | $443,312$ | - |
| $\mathcal{P}_{\geq 1}[true\ \mathcal{U}\ (n = m)]$ | 100 | $1,664$ | $271,988$ | $272,100$ | $267,976$ |
| | 1000 | $1,668$ | $273,012$ | $272,868$ | $267,976$ |
| | 10000 | $1,668$ | $282,740$ | $282,736$ | $267,976$ |
| | 100000 | $21,672$ | $438,424$ | $439,192$ | $267,976$ |

## D.4 Tandem Queuing Network

Table D.4: Tandem Queuing Network - peak VSZ memory

| **property** | **n** | **VSZ memory in Kbytes** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $\mathcal{S}_{<0.01}[\mathit{full}]$ | 2 | 1,524 | 273,860 | 273,868 | 270,868 | - | - |
| | 10 | 1,668 | 273,860 | 273,836 | 270,868 | - | - |
| | 50 | 2,944 | 274,520 | 274,528 | 270,496 | - | - |
| | 100 | 7,000 | 276,088 | 276,004 | 270,648 | - | - |
| | 255 | 35,864 | 279,416 | 284,220 | 271,468 | - | - |
| | 511 | 138,632 | 289,236 | 312,352 | out mem. | - | - |
| | 1023 | 549,920 | 332,668 | 426,696 | out mem. | - | - |
| $\mathcal{S}_{>0.2}[\mathcal{P}_{>0.1}[\mathcal{X}\ \mathit{snd}]]$ | 2 | *n.m.* | 273,860 | 273,836 | 270,868 | - | - |
| | 10 | 1,668 | 273,868 | 273,840 | 270,352 | - | - |
| | 50 | 2,948 | 274,516 | 274,516 | 270,488 | - | - |
| | 100 | 7,008 | 276,100 | 276,004 | 271,864 | - | - |
| | 255 | 35,920 | 279,416 | 284,224 | out mem. | - | - |
| | 511 | 138,808 | 289,264 | 312,112 | out mem. | - | - |
| | 1023 | 550,528 | 331,752 | 425,032 | out mem. | - | - |
| $\mathcal{P}_{<0.1}[\mathit{true}\,\mathcal{U}^{[0.5,2]}\ \mathit{full}]$ | 2 | *n.m.* | 273,868 | 273,836 | - | 2,820 | - |
| | 10 | 1,668 | 273,868 | 273,836 | - | 2,824 | - |
| | 50 | 2,576 | 274,380 | 274,400 | - | 2,824 | - |
| | 100 | 5,364 | 275,964 | 275,956 | - | 2,824 | - |
| | 255 | 25,620 | 280,884 | 285,640 | - | 2,824 | - |
| | 511 | 97,976 | 297,824 | 320,300 | - | 2,824 | - |
| | 1023 | 386,952 | 362,080 | 455,464 | - | 2,824 | - |
| $\mathcal{P}_{\leq 0,01}[\mathit{true}\,\mathcal{U}^{\leq 2}\ \mathit{full}]$ | 2 | 1,660 | 273,860 | 273,868 | 270,868 | *n.m.* | 267,984 |
| | 10 | 1,672 | 273,860 | 273,836 | 270,868 | 2,824 | 267,984 |
| | 50 | 2,880 | 274,380 | 274,356 | 270,432 | 2,824 | 267,984 |
| | 100 | 6,428 | 275,804 | 275,792 | 270,572 | 2,824 | 267,984 |
| | 255 | 33,492 | 279,788 | 285,128 | 270,284 | 2,824 | 267,984 |
| | 511 | 129,116 | 293,208 | 316,276 | out mem. | 2,824 | 267,984 |
| | 1023 | 511,676 | 345,984 | 439,592 | out mem. | 2,824 | 267,984 |
| $\mathcal{P}_{\leq 0,5}[\mathit{true}\,\mathcal{U}^{\leq 10}\ \mathit{fst}]$ | 2 | *n.m.* | 273,868 | 273,836 | 271,384 | *n.m.* | 267,652 |
| | 10 | 1,668 | 273,868 | 273,844 | 270,948 | 2,824 | 267,984 |
| | 50 | 2,536 | 273,996 | 274,352 | 270,432 | 2,824 | 267,984 |
| | 100 | 5,204 | 275,800 | 275,788 | 270,532 | 2,824 | 267,988 |
| | 255 | 24,596 | 279,764 | 285,116 | 270,268 | 2,824 | 267,984 |
| | 511 | 93,884 | 292,848 | 316,180 | out mem. | 2,824 | 267,988 |
| | 1023 | 370,564 | 345,308 | 439,024 | out mem. | 2,824 | 267,988 |
| | 2 | *n.m.* | 273,868 | 273,836 | 271,384 | *n.m.* | 267,652 |
| | 10 | *n.m.* | 273,868 | 273,844 | 270,948 | 2,824 | 267,984 |
| | 50 | 1668 | 273,996 | 274,352 | 270,432 | 2,824 | 267,984 |
| | 100 | 6,424 | 275,800 | 275,788 | 270,532 | 2,824 | 267,988 |
| | 255 | 33,448 | 279,764 | 285,116 | 270,268 | 2,824 | 267,984 |
| | 511 | 129,012 | 292,848 | 316,180 | out mem. | 2,824 | 267,988 |
| | 1023 | 511,520 | 345,308 | 439,024 | out mem. | 2,824 | 267,988 |
| $\mathcal{P}_{\geq 1}[\mathit{snd}\,\mathcal{U}\ \mathit{sndn}]$ | 2 | *n.m.* | 273,860 | 273,868 | 270,860 | - | 267,972 |
| | 10 | *n.m.* | 273,860 | 273,872 | 270,380 | - | 267,992 |
| | 50 | 1,668 | 273,988 | 274,076 | 270,868 | - | 267,972 |
| | 100 | 5,044 | 274,348 | 274,356 | 270,352 | - | 267,988 |
| | 255 | 31,324 | 275,280 | 275,184 | 270,928 | - | 267,992 |
| | 511 | 120,860 | 276,348 | 276,324 | 269,960 | - | 267,988 |
| | 1023 | 478,108 | 279,472 | 279,452 | out mem. | - | 267,988 |

## D.5 Cyclic Server Polling system

Table D.5: Cyclic Server Polling system - peak VSZ memory - part a

| **property** | **n** | **VSZ memory in Kbytes** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $\mathcal{S}_{<0.2}[busy_1\ \mathcal{U}\ serve_1]$ | 3 | *n.m.* | 272.772 | 273.232 | 298.228 | - | - |
| | 6 | 1.820 | 273.864 | 273.900 | 270.384 | - | - |
| | 9 | 3.596 | 274.936 | 274.832 | 617.716 | - | - |
| | 12 | 25.060 | 277.228 | 282.672 | 270.804 | - | - |
| | 15 | 256.716 | 293.652 | 367.656 | `out mem.` | - | - |
| | 16 | 561.232 | 313.904 | 482.096 | `out mem.` | - | - |
| | 17 | 1.223.528 | 354.424 | 733.056 | `out mem.` | - | - |
| | 18 | 2.658.048 | 442.992 | 1.285.452 | `out mem.` | - | - |
| $\mathcal{P}_{\leq 0.99}[true\ \mathcal{U}^{[40,80]}\ serve_1]$ | 3 | 1.668 | 272.928 | 273.088 | - | 2.824 | - |
| | 6 | 1.800 | 273.860 | 273.996 | - | 2.824 | - |
| | 9 | 3.248 | 274.960 | 274.860 | - | 2.824 | - |
| | 12 | 20.168 | 277.752 | 282.820 | - | 2.824 | - |
| | 15 | 208.832 | 304.592 | 378.608 | - | 2.820 | - |
| | 16 | 459.044 | 337.344 | 505.424 | - | 2.824 | - |
| | 17 | 1.006.428 | 406.148 | 783.980 | - | 2.824 | - |
| | 18 | 2.198.012 | 552.484 | 1.394.964 | - | 2.956 | - |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 5}\ poll_1]$ | 3 | *n.m.* | 272.856 | 273.232 | 270.880 | *n.m.* | 268.052 |
| | 6 | 1.780 | 274.348 | 273.896 | 270.340 | *n.m.* | 267.980 |
| | 9 | 3.104 | 274.648 | 274.648 | 617.716 | *n.m.* | 267.980 |
| | 12 | 19.572 | 277.196 | 282.400 | 270.904 | *n.m.* | 267.980 |
| | 15 | 202.940 | 298.704 | 369.584 | `out mem.` | *n.m.* | 267.980 |
| | 16 | 446.496 | 325.052 | 486.220 | `out mem.` | *n.m.* | 267.980 |
| | 17 | 979.664 | 379.936 | 743.292 | `out mem.` | *n.m.* | 267.980 |
| | 18 | 2.141.688 | 497.084 | 1.308.948 | `out mem.` | *n.m.* | 267.980 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 10}\ poll_1]$ | 3 | *n.m.* | 273.144 | 272.768 | 270.948 | *n.m.* | 267.980 |
| | 6 | 1.780 | 273.988 | 273.896 | 270.872 | *n.m.* | 267.980 |
| | 9 | 3.104 | 274.640 | 274.736 | 270.496 | *n.m.* | 267.980 |
| | 12 | 19.572 | 277.284 | 282.400 | 270.508 | *n.m.* | 267.980 |
| | 15 | 202.940 | 298.608 | 369.584 | `out mem.` | *n.m.* | 267.980 |
| | 16 | 446.496 | 325.048 | 486.220 | `out mem.` | *n.m.* | 267.980 |
| | 17 | 979.664 | 379.936 | 743.312 | `out mem.` | *n.m.* | 267.980 |
| | 18 | 2.141.688 | 497.076 | 1.308.948 | `out mem.` | *n.m.* | 267.980 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 20}\ poll_1]$ | 3 | 1.668 | 272.892 | 273.860 | 270.872 | *n.m.* | 267.980 |
| | 6 | 1.780 | 273.988 | 273.896 | 270.856 | *n.m.* | 267.980 |
| | 9 | 3.104 | 274.640 | 274.644 | 270.432 | *n.m.* | 267.980 |
| | 12 | 19.572 | 277.284 | 282.400 | 271.000 | *n.m.* | 267.980 |
| | 15 | 202.940 | 298.932 | 369.584 | `out mem.` | *n.m.* | 267.980 |
| | 16 | 446.496 | 325.052 | 486.308 | `out mem.` | *n.m.* | 267.980 |
| | 17 | 979.664 | 380.064 | 743.312 | `out mem.` | *n.m.* | 617.716 |
| | 18 | 2.141.688 | 497.208 | 1.308.948 | `out mem.` | *n.m.* | 267.980 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 40}\ poll_1]$ | 3 | 1.668 | 273.860 | 273.368 | 298.228 | *n.m.* | 267.980 |
| | 6 | 1.800 | 273.988 | 273.896 | 270.572 | *n.m.* | 267.980 |
| | 9 | 3.104 | 274.732 | 274.644 | 270.492 | *n.m.* | 267.980 |
| | 12 | 19.572 | 277.284 | 282.492 | 271.004 | *n.m.* | 267.980 |
| | 15 | 202.940 | 298.744 | 369.664 | `out mem.` | *n.m.* | 267.980 |
| | 16 | 446.496 | 324.960 | 486.220 | `out mem.` | *n.m.* | 267.980 |
| | 17 | 979.664 | 379.468 | 743.308 | `out mem.` | *n.m.* | 267.980 |
| | 18 | 2.141.688 | 496.648 | 1.308.948 | `out mem.` | *n.m.* | 457.972 |
| $busy_1 \implies \mathcal{P}_{\geq 0.5}[true\ \mathcal{U}^{\leq 80}\ poll_1]$ | 3 | 1.668 | 273.952 | 273.860 | 270.948 | *n.m.* | 267.980 |
| | 6 | 1.780 | 273.860 | 273.896 | 270.856 | *n.m.* | 267.980 |
| | 9 | 3.104 | 274.660 | 274.644 | 270.504 | *n.m.* | 267.980 |
| | 12 | 19.572 | 277.416 | 282.400 | 271.016 | *n.m.* | 457.972 |
| | 15 | 202.940 | 298.840 | 369.588 | `out mem.` | *n.m.* | 267.980 |
| | 16 | 446.496 | 324.960 | 486.224 | `out mem.` | *n.m.* | 267.980 |
| | 17 | 979.796 | 380.068 | 743.312 | `out mem.` | *n.m.* | 267.980 |
| | 18 | 2.141.688 | 497.216 | 1.308.948 | `out mem.` | *n.m.* | 267.980 |

Table D.6: Cyclic Server Polling system - peak VSZ memory - part b

| **property** | **n** | **VSZ memory in Kbytes** | | | | | |
|---|---|---|---|---|---|---|---|
| | | mrmc | prism hybrid | prism sparse | etmcc | ymer | vesta |
| $busy_1 \implies \mathcal{P}_{\geq 1.0}[true\, \mathcal{U}\ poll_1]$ | 3 | *n.m.* | 272.800 | 273.860 | 270.856 | - | 267.976 |
| | 6 | 1.668 | 273.956 | 273.896 | 270.856 | - | 267.976 |
| | 9 | 2.508 | 273.956 | 273.944 | 270.932 | - | 267.976 |
| | 12 | 18.824 | 273.832 | 274.212 | 270.264 | - | 267.976 |
| | 15 | 241.080 | 274.760 | 274.892 | `out mem.` | - | 267.976 |
| | 16 | 527.824 | 275.032 | 274.988 | `out mem.` | - | 617.716 |
| | 17 | 1.152.504 | 275.184 | 275.144 | `out mem.` | - | 267.976 |
| | 18 | 2.507.400 | 275.316 | 275.304 | `out mem.` | - | 617.716 |
| $\neg(poll_1 \wedge poll_2)$ | 3 | *n.m.* | 273.236 | 272.768 | 270.340 | *n.m.* | 267.992 |
| | 6 | 1.668 | 272.772 | 272.768 | 270.872 | *n.m.* | 267.992 |
| | 9 | 2.508 | 273.976 | 273.972 | 270.900 | *n.m.* | 268.060 |
| | 12 | 14.948 | 273.832 | 273.832 | 617.716 | *n.m.* | 267.992 |
| | 15 | 156.956 | 274.988 | 274.896 | `out mem.` | *n.m.* | 267.880 |
| | 16 | 348.416 | 275.040 | 274.932 | `out mem.` | *n.m.* | 267.992 |
| | 17 | 771.248 | 274.736 | 274.648 | `out mem.` | *n.m.* | 267.992 |
| | 18 | 1.700.312 | 275.284 | 275.104 | `out mem.` | *n.m.* | 267.992 |

# Bibliography

[1] MRMC website. November, 2006, http://www.cs.utwente.nl/~zapreevis/mrmc/.

[2] PRISM website. November, 2006, http://www.cs.bham.ac.uk/~dxp/prism/.

[3] SPSS - Statistical Package for the Social Sciences, version 12.0. February, 2007, http://www.spss.com/.

[4] VESTA website. March, 2007, http://osl.cs.uiuc.edu/~ksen/vesta2/.

[5] Gul A. Agha, José Meseguer, and Koushik Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, 2005.

[6] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[7] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[8] Suzana Andova, Holger Hermanns, and Joost-Pieter Katoen. Discrete-time rewards model-checked. In Kim G. Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems : First International Workshop (FORMATS'03)*, volume 2791 of *LNCS*, pages 88–104, Berlin, 2003. Springer.

[9] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert K. Brayton. Verifying continuous time Markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276, Berlin, 1996. Springer.

[10] C. Baier. *On algorithmic verification methods for probabilistic systems.* habilitation thesis, University of Mannheim, 1998.

[11] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time Markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

[12] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model checking continuous-time Markov chains by transient analysis. In E. Allen Emerson and A. Prasad Sistla, editors, *Proc. 12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 358–372, Berlin, 2000. Springer.

[13] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the logical characterisation of performability properties. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP '00)*, volume 1853 of *LNCS*, pages 780–792, Berlin, 2000. Springer.

[14] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In Jos C. M. Baeten and Sjouke Mauw, editors, *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 146–161, Berlin, 1999. Springer.

[15] Christel Baier and Marta Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[16] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *LNCS*, pages 499–513, Berlin, 1995. Springer.

[17] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. *INFORMS J. on Computing*, 12(3):203–222, 2000.

[18] Byron Changuion, Ian Davies, and Micheal Nelte. DaNAMiCS - a petri net editor. March, 2007, `http://www.cs.uct.ac.za/Research/DNA/microweb/danamics/DNAFrameH.html`.

[19] Frank Ciesinski and Marcus Größer. On probabilistic computation tree logic. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems*, volume 2925 of *LNCS*, pages 147–188, Berlin, 2004. Springer.

[20] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[21] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, London, UK, 1999. ISBN 0-262-03270-8.

[22] David R. Cox. A use of complex probabilities in the theory of stochastic processes. In *Proc. Cambridge Philosophical Society*, volume 51, pages 313–319, 1955.

[23] Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. *International Journal on Software Tools for Technology Transfer (STTT'04)*, 5(2-3):221–236, 2004.

[24] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In Susanne Graf and Michael I. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 395–410, Berlin, 2000. Springer.

[25] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.

[26] Ronald A. Fisher. Applications of student's distribution. *Metron*, 5:90–104, 1925.

[27] Wan Fokkink and Jun Pang. Simplifying itai-rodeh leader election for anonymous rings. *Electronic Notes in Theoretical Computer Science*, 128(6):53–68, 2005.

[28] Bennett L. Fox and Peter W. Glynn. Computing poisson probabilities. *Communications of the ACM*, 31(4):440–445, 1988.

[29] Nissim Francez. *Fairness*. Springer-Verlag, NY, USA, 1986. ISBN 0-387-96235-2.

[30] M. Fujita, P.C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Form. Methods Syst. Des.*, 10(2-3):149–169, 1997.

[31] Laurence G. Grimm. *Statistical Applications for the Behavioral Sciences.* John Wiley & Sons Inc., NY, USA, 1993. ISBN 0-471-50982-5.

[32] Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar. On randomization in sequential and distributed algorithms. *ACM Comput. Surv.*, 26(1):7–86, 1994.

[33] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[34] Boudewijn Haverkort, Lucia Cloth, Holger Hermanns, Joost-Pieter Katoen, and Christel Baier. Model-checking performability properties. In *International Conference on Dependable Systems and Networks (DSN'02)*, pages 103–112, Washington D. C., 2002. IEEE CS Press.

[35] Holger Hermanns, Ulrich Herzog, Ulrich Klehmet, Vassilis Mertsiotakis, and Markus Siegle. Compositional performance modelling with the TIPPtool. *Performance Evaluation*, 39(1-4):5–35, 2000.

[36] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A Markov chain model checker. In Susanne Graf and Michael I. Schwartzbach, editors, *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 347–362, Berlin, 2000. Springer.

[37] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. Towards model checking stochastic process algebra. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *2nd International Conference on Integrated Formal Methods (IFM'00)*, volume 1945 of *LNCS*, pages 420–439, Berlin, 2000. Springer.

[38] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A tool for model-checking Markov chains. *Int. J. on Softw. for Technology Transfer (STTT)*, 4(2):153–172, 2003.

[39] Holger Hermanns, Joachim Meyer-Kayser, and Markus Siegle. Multi-terminal binary decision diagrams to represent and analyse continuous-time Markov chains. In B. Plateau, W. J. Stewart, and M. Silva, editors, *Proc. 3rd Int. Workshop on the Num. Sol. of Markov Chains*, pages 188–207, Spain, 1999. Prensas Universitarias de Zaragoza.

[40] Jane Hillston. *Compositional Approach to Performance Modelling.* Cambridge University Press, Cambridge, UK, 1996. ISBN 0-521-57189-8.

[41] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In Holger Hermanns and Jens Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444, Berlin, 2006. Springer.

[42] Robert V. Hogg and Allen T. Craig. *Introduction to Mathematical Statistics.* Macmillan, NY, USA, fourth edition, 1978. ISBN 0-02-355710-9.

[43] Oliver C. Ibe and Kishor S. Trivedi. Stochastic Petri Net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[44] Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.

[45] Arne Jensen. Markoff chains as an aid in the study of Markoff processes. *Skandinavisk Aktuarietidskrift*, 36:87–91, 1953.

[46] Samuel Karlin and James L. McGregor. The differential equations of birth-and-death processes, and the Stieltjes moment problem. *Transactions of the American Mathematical Society*, 85(2):489–546, 1957.

[47] Joost-Pieter Katoen, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Faster and symbolic CTMC model checking. In Luca de Alfaro and Stephen Gilmore, editors, *Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01)*, volume 2165 of *LNCS*, pages 23–38, Berlin, 2001. Springer.

[48] Joost-Pieter Katoen and Ivan S. Zapreev. Safe on-the-fly steady-state detection for time-bounded reachability. In *Third International Conference on the Quantitative Evaluation of Systems (QEST'06)*, pages 301–310, Washington D. C., 2006. IEEE Computer Society.

[49] Vidyadhar G. Kulkarni. *Modeling and analysis of stochastic systems.* Chapman & Hall, London, UK, 1995. ISBN 0-412-04991-0.

[50] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, *Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02)*, volume 2324 of *LNCS*, pages 200–204, Berlin, 2002. Springer.

[51] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 2.0: A tool for probabilistic model checking. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, pages 322–323, Washington D. C., 2004. IEEE Computer Society.

[52] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Quantitative analysis with the probabilistic model checker PRISM. *Electronic Notes in Theoretical Computer Science*, 153(2):5–31, 2005.

[53] Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using Digital Clocks. In Kim Guldstrand Larsen and Peter Niebert, editors, *1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*, pages 105–120, Berlin, 2003. Springer.

[54] Richard Lassaigne and Sylvain Peyronnet. Approximate verification of probabilistic systems. In Holger Hermanns and Roberto Segala, editors, *Process Algebra and Probabilistic Methods. Performance Modeling and Verification (PAPM-PROBMIV'02)*, volume 2399 of *LNCS*, pages 213–214, Berlin, 2002. Springer.

[55] Paola Lecca and Corrado Priami. Cell cycle control in eukaryotes: A BioSpi model. In *Proc. Workshop on Concurrent Models in Molecular Biology (BioConcur'03)*, ENTCS, 2003.

[56] Daniel J. Lehmann and Michael O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *Proc. 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, New York, 1981. ACM Press.

[57] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic, Dordrecht, NL, 1993. ISBN 0-7923-9380-5.

[58] Sri Gopal Mohanty, Aliakbar Montazer-Haghighi, and R. Trueblood. On the transient behavior of a finite birth-death process with an application. *Computers and Operations Research*, 20(3):239–248, 1993.

[59] Jogesh K. Muppala and Kishor S. Trivedi. Numerical transient solution of finite markovian queueing systems. In U. Narayan Bhat and Ishwar V. Basawa, editors, *Queueing and Related Models*, pages 262–284, USA, 1992. Oxford University Press.

[60] Gethin Norman and Vitaly Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[61] James Norris. *Markov chains.* Cambridge series on statistical and probabilistic mathematics no. 2. Cambridge University Press, Cambridge, UK, 1997. ISBN 0-521-48181-3.

[62] H.A. Oldenkamp. Probabilistic model checking: A comparison of tools. MSc thesis, University of Twente, Enschede, The Netherlands, 2007. http://www.cs.utwente.nl/~oldenkampha.

[63] David Anthony Parker. Implementation of symbolic model checking for probabilistic systems. Master's thesis, the University of Birmingham, 2002.

[64] Bernard Philippe, Youcef Saad, and William Stewart. Numerical methods in Markov chain modelling. *Operations Research*, 40(6):1156–1179, 1992.

[65] Sergio Pissanetzky. *Sparse Matrix Technology.* Academic Press, London, UK, 1984. ISBN 0-12-557580-7.

[66] Amir Pnueli and Lenore D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.

[67] Muhammad A. Qureshi and William H. Sanders. A new methodology for calculating distributions of reward accumulated during a finite interval. In *Proc. of the 26th International Symposium on Fault-Tolerant Computing (FTCS'96)*, pages 116–125, Japan, 1996.

[68] Youcef Saad. *NumericalMethods for Large Eigenvalue Problems.* Manchester University Press, Manchester, UK, 1992. ISBN 0-7190-3386-1.

[69] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In Rajeev Alur and Doron Peled, editors, *Proc. 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215, Berlin, 2004. Springer.

[70] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In Kousha Etessami and Sriram K. Rajamani, editors, *Proc. 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 266–280, Berlin, 2005. Springer.

[71] Gerald S. Shedler. *Regenerative stochastic simulation.* Acedemic Press, London, UK, 1993. ISBN 0-12-639360-5.

[72] Matthew Simmons. *Automata Theory.* World Scientific Publishing, Singapore, 1999. ISBN 981-023753-7.

[73] Fabio Somenzi. *CUDD: CU Decision Diagram package.* Public software, Colorado University, Boulder, 1997.

[74] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, New Jersey, UK, 1994. ISBN 0-691-03699-3.

[75] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Compting*, 1(2):146–160, 1972.

[76] Henk C. Tijms. *A First Course in Stochastic Models.* John Wiley & Sons Ltd, West Sussex, UK, 2003. ISBN 0-471-49881-5.

[77] Henk C. Tijms and R. Veldman. A fast algorithm for the transient reward distribution in continuous-time Markov chains. In *Operation Research Letters*, volume 26, pages 155–158, 2000.

[78] Kishor S. Trivedi. *Probability & statistics with reliability, queuing, and computer science applications.* Prentice-Hall, New Jersey, USA, 1982. ISBN 0-13-711564-4.

[79] Abraham Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[80] Ward Whitt. Continuity of generalized semi-markov processes. *Mathematics of Operations Research*, 5:494–501, 1980.

[81] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[82] Håkan L. S. Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2005. CMU-CS-05-105, http://www.cs.cmu.edu/˜lorens/papers/.

[83] Håkan L. S. Younes. Ymer: A statistical model checker. In Kousha Etessami and Sriram K. Rajamani, editors, *Proc. 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 429–433, Berlin, 2005. Springer.

[84] Håkan L. S. Younes, Marta Kwiatkowska, Gethin Norman, and David Parker. Numerical vs. statistical probabilistic model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):216–228, 2006.

[85] Håkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proc. 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 223–235, Berlin, 2002. Springer.

[86] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation*, 204(9):1368–1409, 2006.

[87] David M. Young. *Iterative Solution of Large Linear Systems*. Acedemic Press, NY, USA, 1971. ISBN 0-12-773050-8.