

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

Bachelor Thesis

Symbolic Model Checking of Probabilistic Systems using Multi-Terminal Binary Decision Diagrams

Frederick Prinz

September 24, 2014

First Reviewer:
Prof. Dr. Ir. Joost-Pieter Katoen
Second Reviewer:
Prof. Dr. Erika Ábrahám

Advisor:
Dipl.-Inform. Christian Dehnert

Erklärung

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

Aachen, 24.September, 2014

Frederick Prinz

Abstract

Model checking is a method to verify probabilistic systems with respect to their specified requirements. In particular, we consider symbolic model checking, where the systems are represented by a symbolic data structure, e.g. multi-terminal binary decision diagrams (MTBDDs). In this thesis we present the theoretical background and the conceptual view of symbolic model checking. Furthermore, we show the competitiveness of our implementation in the model checking tool STORM. Thereby, we provide experimental results for common probabilistic models and compare the results to the well-known model checker PRISM [1]. Beside the general performance, we test a new variant for determining all reachable states in the given model. The algorithm is called Chaining and we measure the time and memory requirements of this algorithm compared to the standard breath first search (BFS). In addition, we consider the variable ordering in the given probabilistic program, which specifies the probabilistic system. We show that the variable ordering in the program significantly influences the size of the MTBDD and the time required for model checking, respectively. We provide a first heuristic to order these variables, such that the MTBDD size is relatively small. Different case studies prove the efficiency of this heuristic.

Contents

1. Introduction	1
1.1. Motivation	1
1.2. Outline	2
1.3. Related Work	3
2. Preliminaries	5
2.1. Markov Models	5
2.1.1. Discrete-time Markov Chain	5
2.1.2. Discrete-time Markov Decision Process	7
2.2. Probabilistic Computation Tree Logic	10
2.3. Modeling Language	13
2.4. MTBDD	17
3. From Probabilistic Programs to MTBDDs	21
3.1. Model Representation	21
3.1.1. Variables	21
3.1.2. Expressions	23
3.1.3. Commands	25
3.1.4. Synchronising Actions and Non-determinism	26
3.2. Model Checking Algorithms	29
3.2.1. State Formulas	29
3.2.2. Path Formulas	30
4. Evaluation	35
4.1. Performance	36
4.2. Chaining	38
4.3. Variable Ordering	40
5. Conclusion	47
5.1. Summary	47
5.2. Future Work	47
A. Appendix	53

List of Figures

1.1.	Model checking process.	1
2.1.	An example DTMC.	7
2.2.	An example MDP with two possible actions.	9
2.3.	All reachable states of the induced MDP $M_{P_{ex1}}$	16
2.4.	An example MTBDD (a) stored in fully reduced form (b).	18
2.5.	Example MTBDDs representing a set of states and a transition matrix.	20
3.1.	MTBDDs for the variable $t \in [2, 4]$	23
3.2.	The expression $v' = v + 1$ in MTBDD representation ($v, v' \in [0, 3]$).	25
3.3.	The MTBDD M_c representing the example command c	26
3.4.	Partitioning of the probabilistic program P_{ex2}	28
3.5.	The transition MTBDD of P_{ex2}	29
4.1.	Relative time for construction and reachability.	37
4.2.	Chaining vs. BFS (iterations).	39
4.3.	Chaining vs. BFS (time per iteration).	40
4.4.	An identity MTBDD with different variable orderings.	40
4.5.	An example MTBDD with different meta-variable orderings.	41
4.6.	Ordering module meta-variables.	44

1. Introduction

1.1. Motivation

Nowadays computerised systems are spread over a huge range of applications areas, but analysing these more and more complex systems becomes very difficult. One approach to prove that a given system behaves as intended is *formal verification*. In particular, we are looking for an automated version of the verification process, e.g. *model checking* [2]. In general, model checking is a time and resource intensive task. Thus, it is mainly used for safety-critical systems, where errors have severe consequences. These systems are for example control software of aircraft or satellites [3].

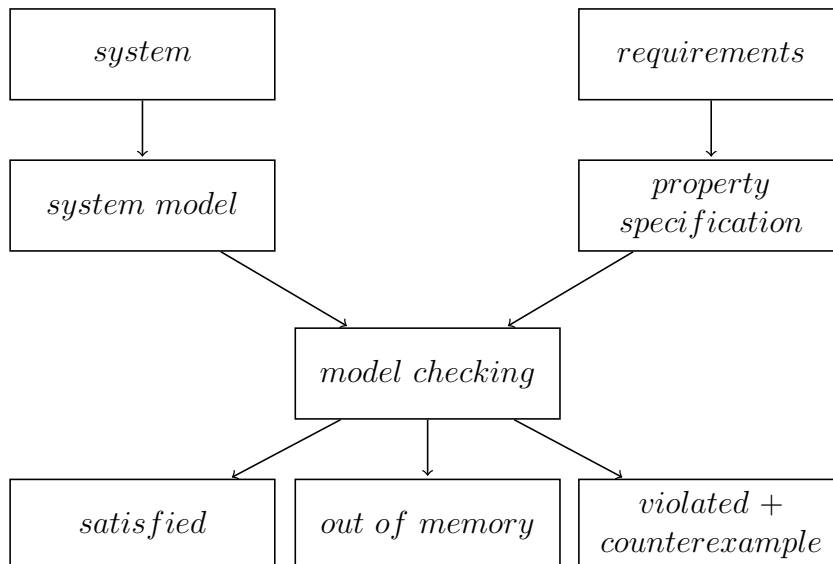


Figure 1.1.: Model checking process.

The process of model checking is illustrated in Figure 1.1. A model checker requires a formal model and formal properties as input. Usually a real-world system is represented by a model in a formal language [4]. The user specifies the requirements and formalises these requirements in a property specification language [5]. The model checking tool automatically checks whether the (formal) model satisfies the (formal) properties. In the best case, it either proves the correctness of the system or provides a counterexample. Such a counterexample is a violating execution of the system, which can support the designer to resolve the existing issue. However, in practice we are also limited by the

available resources. Thus, we might get neither satisfied nor violated as possible result because of insufficient resources (and this includes time).

Model checking is a quite expensive task, because all possible executions of the system need to be analysed. In contrast to ordinary testing (with test cases), model checking can prove the correctness of the whole system, while testing can only prove the existence of an error. Some systems exhibit probabilistic behaviour in the real-world, e.g. games [6], communication networks [7] and biological systems [8]. Probabilistic models [2] are needed to formally reflect this behaviour. Therefore, we entirely focus on this type of model.

Furthermore, we are confronted with the state space explosion problem for many real-world systems. Modelling already simple systems often leads to an enormous number of states of the corresponding model. Several techniques have been proposed to combat the state space explosion problem. One way to do so is to focus on the data structure that is used. In particular, we focus on a branch of model checking called *symbolic model checking* [9], where a given system is not represented explicitly. Instead, the representation is based on the idea to encode sets of states rather than single states. In general, there are different symbolic data structures. In this thesis we focus on the system's representation with *multi-terminal binary decision diagrams* (MTBDDs). MTBDDs are an extension of binary decision diagrams (BDDs) [10], which have already been used for verification of non-probabilistic models earlier. Parker [11] and others have proposed to use MTBDDs also for probabilistic models, because they often allow for the compact representation of the given system. Nevertheless, the symbolic approach does not necessary provide the best results compared to other approaches, like *explicit* or *hybrid* model checking [11].

1.2. Outline

The thesis is structured as follows. Chapter 2 provides the theoretical background of symbolic model checking. We introduce the different model types and multi-terminal binary decision diagrams (MTBDDs) to represent them. Furthermore, we provide a property specification language for the requirements. In the following Chapter 3 describes the conceptual view of symbolic model checking and presents details about the encoding. In the evaluation in Chapter 4 we provide experimental results for common models. In particular, we compare the performance of our implementation in the model checker STORM with the well-known model checker PRISM [1]. In addition, we test a new variant of the standard reachability algorithm and consider the influence of different variable orderings on the MTBDD size.

1.3. Related Work

The state-of-the-art model checking tool PRISM is presented in [1]. In particular, the representation based on MTBDDs and the model checking algorithms in PRISM are described in [11]. Furthermore, we exclusively use a symbolic model representation. In contrast, a hybrid model checking approach with symbolic and explicit data structures is covered in [11].

We consider probabilistic models with a discrete time space, i.e. DTMCs and MDPs. The continuous-time models (CTMCs and CTMDPs) are used to model processes with an exponential distributed residence time, but they are not covered in this thesis. Further information about model checking these models can be found in [12] for CTMCs and in [13] for CTMDPs.

There are also some minimisation techniques for the given model. One technique is called probabilistic bisimulation [14], which combines states that behave equivalently. Thus, we are often able to significantly reduce the state space of the given model, which usually leads to faster model checking times. But the bisimulation itself requires also time and resources.

For the specification of the requirements we use the probabilistic computation tree logic (PCTL), which has been introduced by Hansson and Jonsson [5]. Other non-probabilistic temporal logics are for example LTL and CTL, which are extensively described in [2] including many examples. The CTL model checking is also covered in [15].

2. Preliminaries

2.1. Markov Models

Traditional model checking allows us to verify certain properties of a transition system [2]. Basically, a transition system consists of a number of states with transitions in-between. We extend this idea to probabilistic model checking, because in real-world environments we are often confronted with stochastic behaviour. Therefore, transition systems are enriched with transition probabilities, which lead to probabilistic models. In this thesis we consider two different types of probabilistic models: discrete-time Markov Chains (DTMCs), which model transitions using only stochastic behaviour, and Markov decision processes (MDPs), which also include non-deterministic choices.

2.1.1. Discrete-time Markov Chain

Discrete-time Markov Chains behave like transition systems, where all non-deterministic choices among successor states are replaced by probabilistic ones.

Definition 2.1 (Discrete-Time Markov Chain (DTMC)).

A discrete-time Markov Chain is a tuple

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

where

- S is a countable, non-empty set of states,
- $\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability function, where $\forall s \in S : \sum_{s' \in S} P(s, s') = 1$,
- $s_{init} \in S$ is the initial state,
- AP is a set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is the labeling function.

Based on the state space S , a discrete probability distribution over S is a function $\mu : S \rightarrow [0, 1]$, where $\sum_{s \in S} \mu(s) = 1$. The set $Dist_S$ contains all probability distributions over S . In each state $s \in S$ of the DTMC the successor state s' is chosen according to the probability distribution $\mathbf{P}(s, \cdot) \in Dist_S$. Thereby, the distribution depends only on the current state and not on prior information or previous states [16]. This assumption

is known as the Markov Property.

The value $\mathbf{P}(s, s')$ of the transition probability function specifies the probability going from state s to s' in one step, i.e. by a single transition. The state s' is called a successor of s , if $\mathbf{P}(s, s') > 0$. We denote the set of all possible successors of state s by $Post(s) = \{s' \in S \mid \mathbf{P}(s, s') > 0\}$. For $T \subseteq S$, the probability moving to an arbitrary state $t \in T$ in one step is defined by $\mathbf{P}(s, T) = \sum_{t \in T} \mathbf{P}(s, t)$.

An execution of the system is represented by a path $\pi = s_0 s_1 s_2 \dots \in S^\omega$, which is an infinite sequence of states such that $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$. Instead of s_i , we also write $\pi[i]$. We further denote all possible paths starting in state $s \in S$ by $Paths(s)$ and all paths in the DTMC D by $Paths_D$. The DTMC D induces a probability measure Pr^D on sets of paths $Paths_D$ [17]. Regarding a measurable set of paths $T \subseteq Paths_D$, T gets a unique probability $Pr^D(T)$ assigned.

In order to verify different properties on the given model, we have to describe the observable behaviour of the underlying real-world system. Therefore, the labeling function L assigns each state a possibly empty set of atomic propositions AP . These atomic propositions represent the properties of interest.

Example 2.1 (DTMC). Figure 2.1 shows a DTMC, which is formally defined by

- $S = \{s_0, s_1, s_2, s_3\}$,
- $\mathbf{P} = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0.4 & 0.6 \\ 0 & 0 & 0.4 & 0.6 \\ 0 & 0 & 0 & 1 \end{pmatrix}$, where entry (m, n) is the transition probability $\mathbf{P}(s_m, s_n)$,
- $s_{init} = s_0$,
- $AP = \{a, g\}$,
- $L(s_0) = L(s_1) = \{a\}$, $L(s_2) = \emptyset$, $L(s_3) = \{g\}$.

In the example the atomic proposition g characterises a goal state. Starting in state s_0 , we can reach the only goal state s_3 in two steps by choosing either the path $\pi_1 = s_0 s_1 (s_3)^\omega$ or $\pi_2 = s_0 s_2 (s_3)^\omega$. The corresponding probabilities are given by:

$$Pr^D(\{\pi \in Paths_D \mid \pi[0] = s_0, \pi[2] = s_3\}) = Pr^D(\{\pi_1, \pi_2\}) = 0.6$$

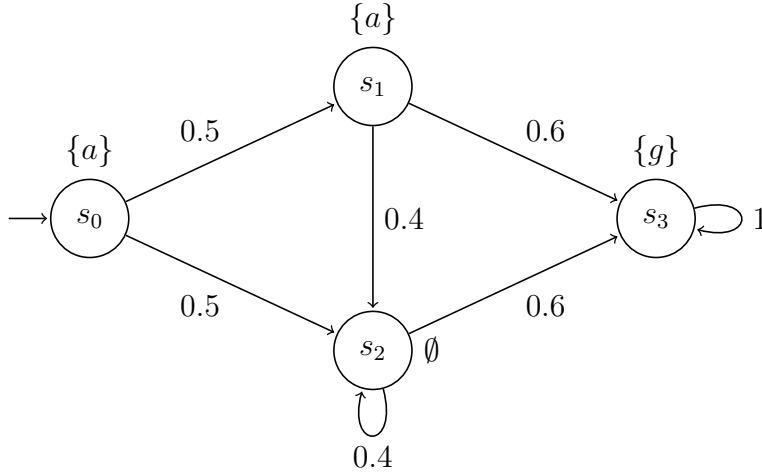


Figure 2.1.: An example DTMC.

2.1.2. Discrete-time Markov Decision Process

Discrete-time Markov Decision Processes are a generalisation of DTMCs. MDPs can model probabilistic as well as non-deterministic behaviour. Non-determinism is for example useful when modelling concurrency between multiple probabilistic systems, which operate in parallel.

Definition 2.2 (Discrete-Time Markov Decision Process (MDP)).

A discrete-time Markov Decision Process is a tuple

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

where

- S is a countable, non-empty set of states,
- A is a set of actions,
- $\mathbf{P} : S \times A \times S \rightarrow [0, 1]$ is the transition probability function, where $\forall s \in S, \forall a \in A : \sum_{s' \in S} \mathbf{P}(s, a, s') \in \{0, 1\}$,
- $s_{init} \in S$ is the initial state,
- AP is a set of atomic propositions,
- $L : S \rightarrow 2^{AP}$ is the labeling function

An action $a \in A$ is called enabled in state s if and only if $\sum_{s' \in S} \mathbf{P}(s, a, s') = 1$. Otherwise, it is called disabled. The set of all enabled actions in state s is denoted by $A(s)$. Furthermore, we require at least one enabled action in each state to prevent deadlocks, i.e. $\forall s \in S. |A(s)| > 0$.

In a state $s \in S$, there exists a non-deterministic choice between all enabled actions $A(s)$. After selecting an enabled action $a \in A(s)$, the successor state is chosen according to the probability distribution $P(s, a, \cdot) \in Dist_S$. This distribution specifies the probability of moving to any other state in S with a single step. With probability $\mathbf{P}(s, a, s')$ we move to the state $s' \in S$. For a set of states $T \subseteq S$, $\mathbf{P}(s, a, T)$ denotes the probability of moving to a successor $t \in T$ with action a and is given by:

$$\mathbf{P}(s, a, T) = \sum_{t \in T} \mathbf{P}(s, a, t)$$

Note that any MDP with $|A(s)| = 1$ for all states $s \in S$ is also a DTMC and vice versa, a DTMC is always an MDP (without non-deterministic choices).

In an MDP, a path $\pi = s_0 a_0 s_1 a_1 s_2 \dots \in (S \times A)^\omega$ consists of an infinite, alternating sequence of states and actions, such that $\mathbf{P}(s_i, a_i, s_{i+1}) > 0$ for all $i \geq 0$. Similar to DTMCs, $Paths(s)$ denotes all paths starting in state s and $Paths_M$ are all paths in the MDP M . The expression $\pi[i]$ refers again to the $(i+1)$ -th state of path π . We define an appropriate probability measure on MDPs by resolving all non-deterministic choices. Since there are no constraints imposed on the non-deterministic choices, we assume a scheduler resolving this non-determinism.

Definition 2.3 (Scheduler).

Let $M = (S, A, \mathbf{P}, s_{init}, AP, L)$ be an MDP. A scheduler σ for M is a function

$$\sigma : S^+ \rightarrow A$$

such that $\sigma(s_0 s_1 \dots s_n) \in A(s_n)$ for all $s_0 s_1 \dots s_n \in S^+$.

In particular a scheduler σ is called memoryless, if its decision depends only on the last state of the given sequence, i.e. σ can be viewed as a function $\sigma_M : S \rightarrow A$.

A scheduler resolves all non-deterministic choices in the MDP, which reduces to a purely probabilistic model, i.e. a DTMC.

Definition 2.4 (Induced DTMC by a Scheduler).

Let $M = (S, A, \mathbf{P}, s_{init}, AP, L)$ be an MDP and σ a scheduler for M . The induced DTMC M_σ is given by

$$M_\sigma = (S^+, \mathbf{P}_\sigma, s'_{init}, AP, L')$$

where for $w = s_0 s_1 \dots s_n \in S^+$:

$$\mathbf{P}_\sigma(w, w s_{n+1}) = \mathbf{P}(s_n, \sigma(w), s_{n+1})$$

and $L'(\sigma) = L(s_n)$.

2.1. Markov Models

Note that M_σ in general has an infinite state space. As M_σ is a DTMC, it induces again a unique probability measure Pr^{M_σ} over all paths $Paths_{M_\sigma}$ [17]. In the following, we denote Pr^{M_σ} also by Pr_σ^M .

Example 2.2 (MDP). In Figure 2.2 there is an example MDP M with

- $S = \{s_0, s_1, s_2, s_3\}$,
- $A = \{\alpha, \beta\}$,
- amongst others, $\mathbf{P}(s_0, \alpha, s_0) = 1$, $\mathbf{P}(s_0, \beta, s_1) = 0.5$,
- initial state s_0 ,
- $AP = \{a, b, g\}$,
- $L(s_0) = \emptyset$, $L(s_1) = \{a\}$, $L(s_2) = \{b\}$, $L(s_3) = \{g\}$.

Let σ be a memoryless scheduler for M with $\sigma(s_0) = \sigma(s_1) = \sigma(s_2) = \beta$ and $\sigma(s_3) = \alpha$. Starting in state s_0 , we get for example the path $\pi = s_0 \sigma(s_0) s_1 \sigma(s_1) (s_3 \sigma(s_3))^\omega = s_0 \beta s_1 \beta (s_3 \alpha)^\omega$. The corresponding probability is given by $Pr_\sigma^M(\{\pi'\}) = Pr^{M_\sigma}(\{\pi'\}) = 0.35$, where $\pi' = (s_0)(s_0 s_1)(s_0 s_1 s_3)(s_0 s_1 s_3 s_3) \dots$.

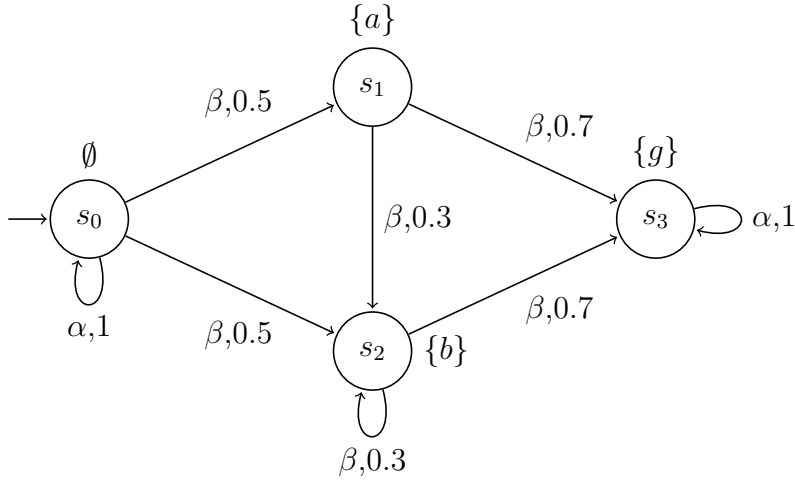


Figure 2.2.: An example MDP with two possible actions.

2.2. Probabilistic Computation Tree Logic

In the last section we have introduced two formalisms to represent certain systems. Now we describe the formalism that is used to specify properties.

A common way to express qualitative properties is the Computation Tree Logic, short CTL [18]. CTL is a branching-time temporal logic, whose syntax is defined by a combination of state and path formulas. State formulas are interpreted in states and evaluate to either true or false, whereas path formulas are evaluated over paths. There are two CTL path quantifiers \exists and \forall . A state $s \in S$ satisfies the CTL formula $\exists\varphi$, if and only if there exists a path starting in s that satisfies φ . Accordingly, for the formula $\forall\varphi$, the sub-formula φ has to hold for all paths starting in s . However, both quantifier do not consider the probability of paths.

For probabilistic models like DTMCs and MDPs, we introduce probabilistic CTL (PCTL), which is based on standard CTL. PCTL also considers the probability of paths. Therefore, $\exists\varphi$ and $\forall\varphi$ are replaced by a new operator $P_{\circ p}(\varphi)$, where $\circ \in \{\leq, \geq, <, >\}$ and $p \in [0, 1]$ is a probability bound. In general, path formulas φ may only occur as a parameter of $P_{\circ p}(\varphi)$.

Definition 2.5 (Probabilistic Computation Tree Logic (PCTL)).

The syntax of a PCTL formula over a set of atomic propositions AP is defined as follows:

- state formulas

$$\Phi ::= true \mid a \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid P_{\circ p}(\varphi)$$

where $a \in AP$, $\circ \in \{\leq, \geq, <, >\}$, $p \in [0, 1]$, φ is a path formula.

- path formulas

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathbf{U}\Phi_2 \mid \Phi_1 \mathbf{U}^{\leq k}\Phi_2$$

where $k \in \mathbb{N}$, Φ , Φ_1 and Φ_2 are state formulas.

Note that the set of atomic propositions AP is taken from the DTMC or MDP. The new operator $P_{\circ p}(\varphi)$ is interpreted as follows. For a DTMC, a state $s \in S$ satisfies $P_{\circ p}(\varphi)$ if the probability of the set of paths starting in s and satisfying φ lies in the interval specified by $\circ p$. For an MDP M , this has to hold for all schedulers σ for M .

Basically, we allow three different path formulas in PCTL. The next operator \bigcirc and until operator \mathbf{U} are the same as for CTL. The bounded until operator $\mathbf{U}^{\leq k}$ adds a step-bound to the until operator. Intuitively, the formula $\Phi_1 \mathbf{U}^{\leq k}\Phi_2$ is true if Φ_2 holds

2.2. Probabilistic Computation Tree Logic

within at most k steps, while Φ_1 holds in all states before Φ_2 is satisfied.

In the following, we define the semantics of PCTL over MDPs, where $s \models \Phi$ indicates that the PCTL formula Φ is satisfied in state s .

Definition 2.6 (PCTL Semantics (for MDPs)).

The PCTL semantics for an MDP $M = (S, A, \mathbf{P}, s_0, AP, L)$ is recursively defined by

- state formulas

For $s \in S$,

$$\begin{aligned}
 s \models true & \\
 s \models a & \quad \text{iff} \quad a \in L(s) \\
 s \models \neg\Phi & \quad \text{iff} \quad a \not\models \Phi \\
 s \models \Phi_1 \wedge \Phi_2 & \quad \text{iff} \quad (a \models \Phi_1) \text{ and } (a \models \Phi_2) \\
 s \models P_{op}(\varphi) & \quad \text{iff} \quad \text{for all schedulers } \sigma \text{ for } M : \\
 & \quad Pr_{\sigma}^M(\{\pi \in Paths(s) \mid \pi \models \varphi\}) \circ p
 \end{aligned}$$

where $a \in AP$, $\circ \in \{\leq, \geq, <, >\}$, $p \in [0, 1]$, φ is a path formula.

- path formulas

For $\pi \in Paths_M$,

$$\begin{aligned}
 \pi \models \bigcirc\Phi & \quad \text{iff} \quad \pi[1] \models \Phi \\
 \pi \models \Phi_1 \mathbf{U}^{\leq k} \Phi_2 & \quad \text{iff} \quad \exists i \in \mathbb{N}, i \leq k. (\pi[i] \models \Phi_2 \wedge \forall j < i. \pi[j] \models \Phi_1) \\
 \pi \models \Phi_1 \mathbf{U} \Phi_2 & \quad \text{iff} \quad \exists i \in \mathbb{N}. (\pi[i] \models \Phi_2 \wedge \forall j < i. \pi[j] \models \Phi_1)
 \end{aligned}$$

where $k \in \mathbb{N}$, Φ , Φ_1 and Φ_2 are state formulas.

For a DTMC $D = (S, \mathbf{P}, s_0, AP, L)$, the PCTL semantics is similar defined except for $P_{op}(\varphi)$. The satisfaction relation for a state $s \in S$ does not depend on schedulers any more:

$$s \models P_{op}(\varphi) \quad \text{iff} \quad Pr^D(\{\pi \in Paths(s) \mid \pi \models \varphi\}) \circ p$$

where $\circ \in \{\leq, \geq, <, >\}$, $p \in [0, 1]$ and φ is a path formula.

Given an MDP M and a PCTL formula Φ , we denote the set of states satisfying Φ by $Sat_M(\Phi) = \{s \in S \mid s \models \Phi\}$. We also write $M \models \Phi$, if the initial state s_{init} satisfies the formula Φ , i.e. $s_{init} \in Sat_M(\Phi)$.

We have introduced the basic PCTL operators in the Definition 2.5. We can derive some more operators in order to simplify some expressions. The disjunction of two formulas $\Phi_1 \vee \Phi_2$ is obtained by the de Morgan's rules, where $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$. Furthermore, the eventually operator (\diamond) is derived by $\diamond\Phi = true \mathbf{U} \Phi$. A path $\pi \in Paths_M$ satisfies $\diamond\Phi$ if there exists an index $i \in \mathbb{N}$ such that $\pi[i] \models \Phi$. Analogously, a path π satisfies $\square\Phi$ if and only if $\forall i \in \mathbb{N}. \pi[i] \models \Phi$. The always operator \square is derived by the

duality of eventually and always and the duality of lower and upper bounds [17].

In some scenarios we want to determine the exact probabilities with which a path formula is satisfied and not restrict to a certain interval. Let D be a DTMC. We can omit the bound of $P_{op}(\varphi)$ and introduce the abbreviation:

$$P_{=?}^s(\varphi) = Pr^D(\{\pi \in Paths(s) \mid \pi \models \varphi\}) \in [0, 1]$$

For MDPs, we also introduce this abbreviation, but we have to consider different schedulers σ for the MDP M :

$$P_{max=?}^s(\varphi) = \sup_{\sigma} \{ Pr_{\sigma}^M(\{\pi \in Paths(s) \mid \pi \models \varphi\}) \} \in [0, 1]$$

$P_{min=?}^s(\varphi)$ is defined similarly.

Example 2.3 (PCTL). Let $M = (S, A, \mathbf{P}, s_0, AP, L)$ be the example MDP introduced in Figure 2.2.

Starting in state s_0 we measure the probability that the next state satisfies $a \in AP$. We get the minimum probability, if a scheduler σ_M selects action $\alpha \in A(s_0)$:

$$Pr_{min=?}^{s_0}(\bigcirc a) = 0, \text{ where } \sigma_M(s_0) = \alpha.$$

Accordingly, the maximum probability is obtained by action β :

$$Pr_{max=?}^{s_0}(\bigcirc a) = 0.5, \text{ where } \sigma_M(s_0) = \beta.$$

A certain probability bound has to hold for all schedulers for M :

$$\begin{aligned} s_0 &\not\models \Phi_1 = P_{>0}(\bigcirc a) \\ s_0 &\models \Phi_2 = P_{\leq 0.5}(\bigcirc a) \end{aligned}$$

Because s_0 is the initial state, we also get:

$$M \not\models \Phi_1, M \models \Phi_2$$

If we focus on the goal state s_3 , which is indicated by the atomic proposition g .

We can avoid to reach the goal state by choosing action α in state s_0 :

$$Pr_{min=?}^{s_0}(\diamond g) = 0, \text{ where } \sigma_M(s_0) = \alpha.$$

In consequence:

$$M \not\models P_{>0}(\diamond g)$$

But if we choose action β , we will almost surely reach the goal state:

$$Pr_{max=?}^{s_0}(\diamond g) = 1, \text{ where } \sigma_M(s_0) = \sigma_M(s_2) = \beta \text{ and } \sigma_M(s_1) = \alpha.$$

The probability to reach the goal state within two steps is given by:

$$Pr_{max=?}^{s_0}(true \mathbf{U}^{\leq 2} g) = 0.7, \text{ where } \sigma_M(s_0) = \sigma_M(s_2) = \beta \text{ and } \sigma_M(s_1) = \alpha.$$

2.3. Modeling Language

This section presents a modeling language for the probabilistic models introduced in Section 2.1. This modeling language is based on a well-known model checker, called PRISM [1]. PRISM is a tool supporting probabilistic model checking for different types of probabilistic models, amongst others, DTMCs and MDPs. The corresponding input language is called the PRISM language, which is a high-level specification formalism to describe the models. For simplicity we introduce only definitions for MDPs, because we have seen that any DTMC is also an MDP.

For a finite set $Var = \{v_1, \dots, v_n\}$ of variables, the domain of each variable is defined by a function $VarDomain : Var \rightarrow [a, b]$, where $a, b \in \mathbb{N}$ and $a < b$. A variable valuation is a function $val : Var \rightarrow \mathbb{N}$, such that $\forall v \in Var. val(v) \in VarDomain(v)$. We further denote the set of all possible valuations by Val^{Var} .

Definition 2.7 (Probabilistic Program, Module, Command).

A probabilistic program is a tuple [19]

$$P = (Var, VarDomain, val_{init}, \mathbf{M})$$

where

- Var is a finite set of variables,
- $VarDomain : Var \rightarrow [a, b]$ specifies the domains for all variables Var ,
- $val_{init} \in Val^{Var}$ is the initial variable valuation,
- $\mathbf{M} = \{Mod_1, Mod_2, \dots, Mod_k\}$ is a finite set of modules

A module is defined by:

$$Mod_i = (Var_i, Act_i, \mathbf{C}_i)$$

- $Var_i \subseteq Var$ is a finite set of variables,
such that $\bigcup_{j=1}^k Var_j = Var$, $\forall 1 \leq i, i' \leq k, i \neq i'. (Var_i \cap Var_{i'} = \emptyset)$,
- Act_i is a finite set of synchronising actions,
- \mathbf{C}_i is a finite set of commands

A command $c \in \mathbf{C}_i$ is a tuple

$$c = (a_c, g_c, (p_1, U_1), \dots, (p_m, U_m))$$

- $a_c \in Act_i \dot{\cup} \{\tau\}$ is a synchronising or independent action,

- g_c is a boolean guard expression, which specifies a subset of valuations $V_{g_c} \subseteq Val^{Var}$,
- $1 \leq j \leq m$, $p_j \in [0, 1]$ is the probability of the update U_j , such that $\sum_{j=1}^m p_j = 1$,
- $U_j : Val^{Var} \rightarrow Val^{Var_i}$ is an update function.

For a command c , the guard g_c specifies a subset $V_{g_c} \subseteq Val^{Var}$ of valuations, which satisfy the guard expression. Let $val \in Val^{Var}$ be a valuation. We write $val \models g_c$, if the guard g_c evaluates to true using val . Note that the current valuation of the variable $v \in Var_i$ in Mod_i can be read by all modules. In contrast, we can change the valuation of v only locally by commands in \mathbf{C}_i . Thus, an update function U_j maps a valuation Val^{Var} of all variables to a valuation Val^{Var_i} of the variables in Mod_i .

The semantics of a probabilistic program $P = (Var, VarDomain, val_{init}, \mathbf{M})$ is defined as follows. The program P usually contains more than one module, i.e. $|\mathbf{M}| > 1$. For simplicity, we reduce the number of modules to one by successively building modules that represent the parallel composition $Mod_1 \parallel Mod_2$ of two modules Mod_1 and Mod_2 . Thus, we get a probabilistic program $P' = (Var, VarDomain, val_{init}, \{Mod\})$ with a single module Mod that represents the parallel composition of all modules of the original program P .

The parallel composition of two modules in \mathbf{M} results in a new module, where we intuitively combine both sets of variables and commands. When combining the command sets, we distinguish three cases depending on the action of a command. First, all commands with an independent action, i.e. a τ -action, always behave independent of the other modules. Similar to the independent action, if an action appears only in one of the two modules, the corresponding commands also behave independently. In contrast, we synchronise between two commands with the same action in both modules. The parallel composition is formally defined by the following Definition 2.8 [19].

Definition 2.8 (Parallel Composition).

Let Mod_1, Mod_2 be two modules. The parallel composition $Mod_1 \parallel Mod_2$ is defined by

$$Mod_1 \parallel Mod_2 = (Var_1 \cup Var_2, Act_1 \cup Act_2, \mathbf{C}_{ind} \cup \mathbf{C}_{sync})$$

- All commands with an independent action or a synchronising action appearing only in one module:

$$\mathbf{C}_{ind} = \{c \mid c \in \mathbf{C}_1 \cup \mathbf{C}_2 \wedge (a_c = \tau \vee a_c \in (Act_1 \cup Act_2) \setminus (Act_1 \cap Act_2))\}$$

- All commands with a synchronising action in both modules:

$$\mathbf{C}_{sync} = \{c \otimes c' \mid c \in \mathbf{C}_1 \wedge c' \in \mathbf{C}_2 \wedge a_c = a_{c'} \in (Act_1 \cap Act_2)\}$$

2.3. Modeling Language

where

$$c \otimes c' = (a_c, g_c \wedge g_{c'}, (p_1 \cdot p'_1, U_1 \oplus U'_1), \dots, (p_1 \cdot p'_{m'}, U_1 \oplus U'_{m'}), \dots, (p_m \cdot p'_{m'}, U_m \oplus U'_{m'}))$$

and for two update functions $U_1 : Val^{Var} \rightarrow Val^{Var_1}$, $U_2 : Val^{Var} \rightarrow Val^{Var_2}$, $(U_1 \oplus U_2) : Val^{Var} \rightarrow Val^{Var_1 \cup Var_2}$ is given by:

$$(U_1 \oplus U_2)(w)(v) = \begin{cases} U_1(w)(v) & , \text{if } v \in Var_1 \\ U_2(w)(v) & , \text{if } v \in Var_2 \end{cases}$$

By parallel composition we get a probabilistic program P' with only one module. In general the program P' induces a probabilistic automata (PA) [20]. A PA is a generalised version of an MDP, where an action is an identifier for possibly multiple probability distributions. Intuitively, we can translate any PA into an MDP by renaming actions such that an action is an identifier for a single probability distribution. Thus, w.l.o.g. we assume different actions for two commands $c_1, c_2 \in \mathbf{C}'$ with overlapping guards, where \mathbf{C}' is the command set of the single module Mod :

$$\exists val \in Val^{Var}. (val \models g_{c_1} \wedge val \models g_{c_2}) \Rightarrow a_{c_1} \neq a_{c_2}$$

The semantics of the probabilistic program P' in terms of an MDP is defined as follows.

Definition 2.9 (Semantics of a Probabilistic Program).

Let $P' = (Var, VarDomain, val_{init}, \{Mod\})$ be a probabilistic program with only one module $M = (Var, Act, \mathbf{C})$. The corresponding MDP $A = (S, A, \mathbf{P}, s_{init}, AP, L)$ is given by:

- $S = Val^{Var}$,
- $A = Act \dot{\cup} \{\tau\}$,
- $\mathbf{P}(s, a, s') = \sum \{p_j \mid c \in \mathbf{C}, a_c = a, s \models g_c, U_i(s) = s'\}$
- $s_{init} = val_{init}$,
- $AP = \{(v = x) \mid v \in Var, x \in VarDomain(v)\}$
- $L : S \rightarrow 2^{AP}$.

Note that each state $val \in S$ of the MDP can be seen as a vector (y_1, \dots, y_n) . The entry $y_i = val(v_i)$ is the value of the variable $v_i \in Var$.

Example 2.4 (PRISM Language). Listing 2.1 shows a probabilistic program P_{ex1} with two modules. The first module represents the example MDP illustrated in Figure 2.2. By parallel composition of both modules, the program P_{ex1} induces the MDP shown in Figure 2.3.

Listing 2.1: A probabilistic program P_{ex1} in PRISM Language.

```

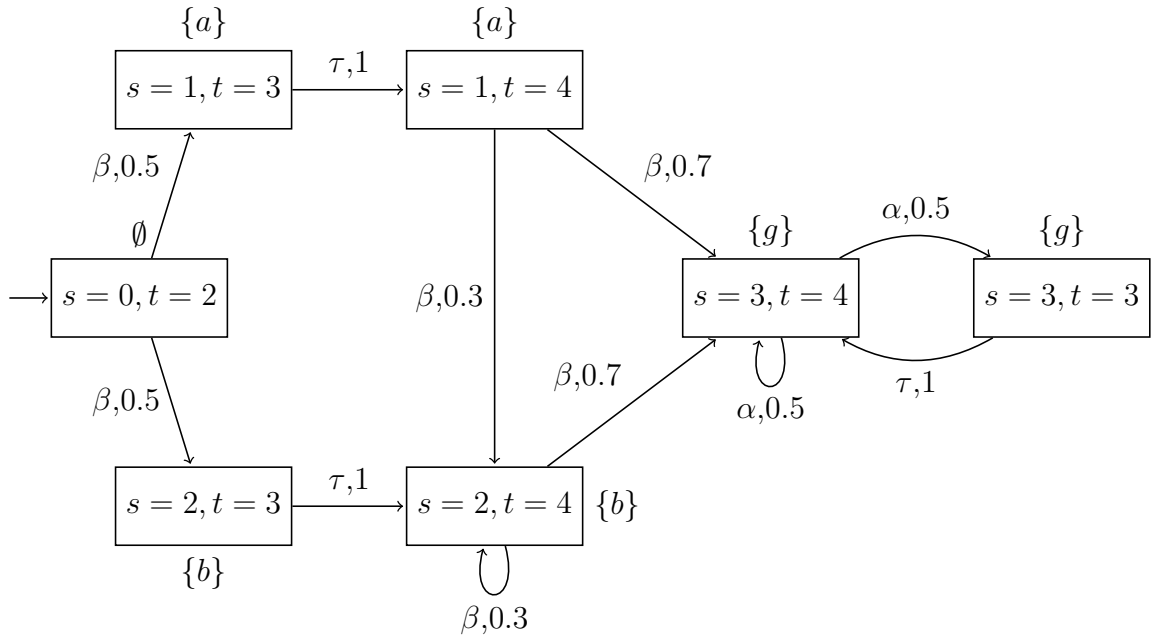
mdp
module mod1
    s: [0..3] init 0;

    [alpha] s=0 -> 1: (s'=0);
    [beta] s=0 -> 0.5: (s'=1) + 0.5: (s'=2);
    [beta] s=1 -> 0.3: (s'=2) + 0.7: (s'=3);
    [beta] s=2 -> 0.3: (s'=2) + 0.7: (s'=3);
    [alpha] s=3 -> 1: (s'=3);
endmodule
module mod2
    t: [2..4] init 2;

    [beta] t=2 -> 1: (t'=3);
    [] t=3 -> 1: (t'=4);
    [alpha] t=4 -> 0.5: (t'=3) + 0.5: (t'=4);
    [beta] t=4 -> 1: (t'=4);
endmodule

// atomic propositions
label "a" = (s=1);
label "b" = (s=2);
label "g" = (s=3);

```

Figure 2.3.: All reachable states of the induced MDP $M_{P_{ex1}}$.

2.4. MTBDD

In this section we introduce multi-terminal binary decision diagrams (MTBDDs), which are an extension of ordinary binary decision diagrams (BDDs) [10]. MTBDDs are a symbolic data structure, which can be used to represent the state space or the transition matrix of a probabilistic model. They have been first proposed in [21].

An MTBDD is a directed acyclic graph $G = (V, E)$ representing a function $f : \mathbb{B}^n \rightarrow D$, where D is usually taken to be \mathbb{R} . In particular, a BDD is a special variant of MTBDDs, where D equals $\{0, 1\}$.

Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of boolean variables, which are totally ordered by $x_1 \prec x_2 \prec \dots \prec x_n$. An MTBDD M is defined over the tuple $\underline{x} = (x_1, x_2, \dots, x_n)$ according to the total order. We distinguish between inner nodes $I \subset V$ and terminal nodes $T \subseteq V$, where $I \cap T = \emptyset$ (and $I \cup T = V$). A labeling function $lab : V \rightarrow (X \cup \mathbb{R})$ specifies a label for each node with the following constraints: An inner node $i \in I$ is labelled with a variable, i.e. $lab(i) \in X$, and a terminal node $t \in T$ by $lab(t) \in \mathbb{R}$, respectively.

The ordering of the nodes of the MTBDD M is imposed by \underline{x} . Let $i_1, i_2 \in I$ be two inner nodes. They are ordered by $i_1 \prec i_2$, if $lab(i_1) \prec lab(i_2)$. For a terminal node $t \in T$, it holds that $i \prec t$ for all inner nodes $i \in I$. Furthermore, an inner node i has exactly two successors denoted by $succ_0(i)$ and $succ_1(i)$, where $i \prec succ_0(i)$ and $i \prec succ_1(i)$. A terminal node has no outgoing edge, i.e. no successor.

An MTBDD M over $\underline{x} = (x_1, x_2, \dots, x_n)$ represents a function $f_M(x_1, x_2, \dots, x_n) : \mathbb{B}^n \rightarrow \mathbb{R}$, where the function value of $f_M(x_1, x_2, \dots, x_n)$ is determined as follows. We start in the unique root node of M . In each inner node $i \in I$, if $lab(i)$ is 0, we take the edge to $succ_0(i)$. Respectively for $lab(i) = 1$, we go to $succ_1(i)$. Finally, the terminal node t determines the resulting value $lab(t)$. If $lab(t) \neq 0$, the corresponding valuation of all boolean variables \underline{x} is called a minterm of M . In the following, we also write $f_M[\underline{x}]$ instead of $f_M(x_1, x_2, \dots, x_n)$.

Let M be an MTBDD over $\underline{x} = (x_1, x_2, \dots, x_n)$ and $b \in \{0, 1\}$. A cofactor $M|_{x_i=b}$ is an MTBDD over the variables $\underline{x}' = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$, which represents the function $f_M(x_1, \dots, x_{i-1}, b, x_{i+1}, \dots, x_n)$. Based on the definition of cofactors, we can write the function f_M for an MTBDD M [11] as follows:

$$f_M(x_1, x_2, \dots, x_n) = x_1 \cdot f_{M|_{x_1=1}} + (1 - x_1) \cdot f_{M|_{x_1=0}}$$

where the top node of M is labelled with x_1 .

Note that the cofactors $M|_{x_1=1}$ and $M|_{x_1=0}$ are also MTBDDs defined over the variables $\underline{x}' = (x_2, \dots, x_n)$. They are recursively defined in the same manner.

2.4. MTBDD

Let $M = (S, A, \mathbf{P}, s_{init}, AP, L)$ be an MDP. We can represent a set of states $T \subseteq S$ by an MTBDD, where each state is encoded binary. In order to guarantee a unique state encoding, the function e has to be injective:

$$e : S \rightarrow \mathbb{B}^n$$

where $n \geq \lceil \log_2(|S|) \rceil$. Note that we require at least $\lceil \log_2(|S|) \rceil$ boolean variables, but in practice we use a more structural encoding with even more variables. This encoding scheme is presented in Chapter 3.

We represent a set of states $T \subseteq S$ by an MTBDD MT . In particular, this MTBDD is also a BDD, because there are only two terminal nodes t_0, t_1 with $lab(t_0) = 0$ and $lab(t_1) = 1$. If we encode the state space S with n boolean variables, the MTBDD function takes also n variables as input. Intuitively, the function $f_{MT} : \mathbb{B}^n \rightarrow \{0, 1\}$ maps a state encoding $e(s)$ to 1, if and only if the state s is in T :

$$f_{MT}[e(s)] = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{otherwise} \end{cases}$$

Based on the state encoding, we can also represent the transition matrix of the MDP M with an MTBDD. The probability function $\mathbf{P}(s, a, s')$ has three parameters, which we all encode binary. Thus, beside the state encoding e , we also specify an injective action encoding $e_A : A \rightarrow \mathbb{B}^m$, where $m = \lceil \log_2(|A|) \rceil$.

The MTBDD P , representing the transition matrix \mathbf{P} , is defined by the function $f_P : \mathbb{B}^{n+m+n} \rightarrow \mathbb{R}$ with

$$f_P[e(s), e_A(a), e(s')] = \mathbf{P}(s, a, s')$$

where $s, s' \in S$ and $a \in A$.

For a DTMC $D = (S, \mathbf{P}, s_0, AP, L)$, we omit the action encoding, which leads to

$$f_P[e(s), e(s')] = \mathbf{P}(s, s')$$

where $s, s' \in S$.

Example 2.6 (MTBDDs representing States and Transitions). The Figure 2.5 shows two exemplary MTBDDs based on the DTMC illustrated in Figure 2.1. The first MTBDD T in Figure 2.5(a) is defined over the boolean variables (x_1, x_0) . We encode each state binary with these boolean variables, i.e. $e(s_0) = 00$, $e(s_1) = 01$, $e(s_2) = 10$, $e(s_3) = 11$. The MTBDD T represents the set of states $S_T = \{s_1, s_2\}$, e.g. $f_T[e(s_1)] = f_T(0, 1) = 1$.

Furthermore Figure 2.5(b) shows the transition matrix of the DTMC (Figure 2.1) encoded as an MTBDD P over (x_1, x_0, x'_1, x'_0) . We encode the source state of the transition binary with the variables x_1, x_0 . Analogously, the destination state is encoded with

primed variables x'_1, x'_0 . Thus, the MTBDD P represents the function $f_P : \mathbb{B}^4 \rightarrow [0, 1]$, where for example $f_P[e(s_2), e(s_3)] = f_P(1, 0, 1, 1) = 0.6$.

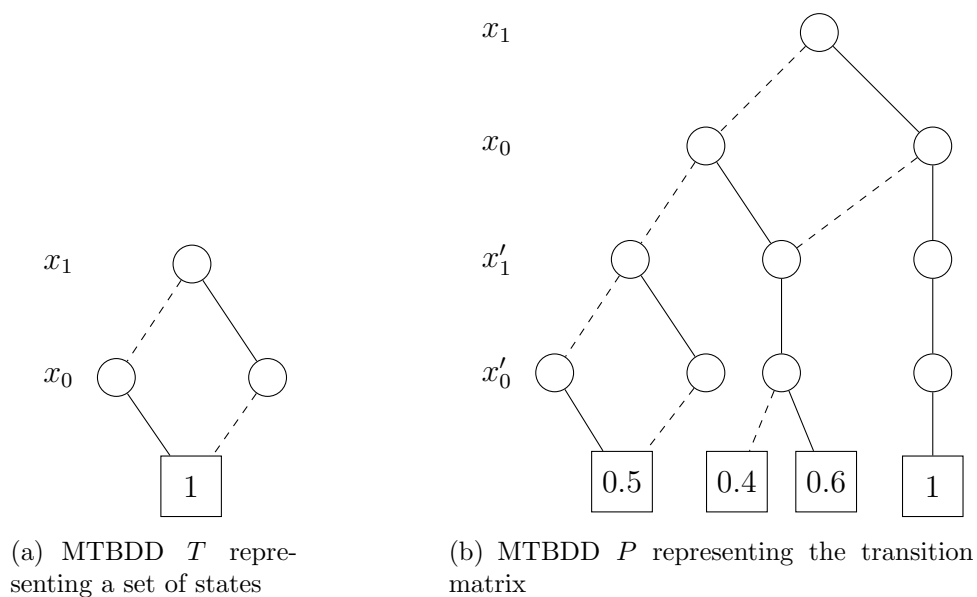


Figure 2.5.: Example MTBDDs representing a set of states and a transition matrix.

Note that the commonly used variable ordering is an interleaved sequence of source and destination variables. In the example this leads to $x_1 \prec x'_1 \prec x_0 \prec x'_0$. In general, the variable ordering significantly influences the size (i.e. the number of nodes $|V|$) of an MTBDD and is therefore covered in more detail in Section 4.3.

3. From Probabilistic Programs to MTBDDs

In this chapter we describe the conceptual view of symbolic model checking with MTBDDs. In particular, the description is based on the implementation in the model checker STORM. The accepted model types are DTMCs and MPDs given by a probabilistic program in the PRISM language. We first establish a symbolic representation based on MTBDDs for the given models (Section 3.1). In particular, we focus on the definition for MDPs, because the MDP representation is also applicable to DTMCs. Note that any MTBDD M represents a function f_M , thus we describe the whole construction process by operations over functions. Afterwards in Section 3.2, we focus on the model checking algorithms for MDPs (and DTMCs). Overall, given a probabilistic program and some quantitative properties in PCTL, the MTBDD representation and the corresponding algorithms enable automated analysis. Finally, Chapter 4 provides some case studies and performance benchmarks.

3.1. Model Representation

In this section we describe the construction of the MTBDDs representing an MDP. First, we present the required encoding schemes and consider the representation of states (Section 3.1.1). In Section 3.1.2 we introduce expressions, which are used to represent the update functions and atomic propositions. Based on the expressions, we represent a command in the probabilistic program by an MTBDD (Section 3.1.3). When combining these commands in Section 3.1.4, we have to consider the synchronising actions and different kinds of non-determinism. In general, we mainly focus on the transition matrix of the probabilistic model, which requires the most complex operations w.r.t. time and resources. The representation of the transition matrix is based on the one in PRISM described in [11].

3.1.1. Variables

Let $P = (Var, VarDomain, val_{init}, \mathbf{M})$ be a probabilistic program, which symbolically represents an MDP $M = (S, A, \mathbf{P}, s_{init}, AP, L)$. In Section 2.4 we have introduced a state encoding function $e : S \rightarrow \mathbb{B}^n$ for the MTBDD representation, which intuitively enumerates all states. In practice we use a more structural approach based on variable encodings. As seen in the Definition 2.9, a state $s \in S$ is a valuation over all variables Var , i.e. $s(v)$ is the value of variable v in state s . We encode each variable

$v \in Var = \{v_1, \dots, v_n\}$ and combine all variable encodings to the whole state encoding. Let $[a, b] = VarDomain(v)$ be the domain of variable v . We represent all possible values of v by a tuple $\underline{v} = (a, a + 1, \dots, b - 1, b)$, where $\underline{v}(i)$ denotes the $(i + 1)$ -th entry of \underline{v} . If $a = 0$, we encode the variable v , i.e. all possible values in $[0, b]$, by

$$e_v : \{0, \dots, 2^m - 1\} \rightarrow \mathbb{B}^m, x \mapsto (x)_2$$

where m is the lowest possible index such that $2^m \geq b$. Furthermore, $(\cdot)_2$ denotes the standard binary representation of an integer value.

Note that the encoding functions e_v is specified for domains of the form $[0, b]$. If the lower bound does not equal zero, i.e. $[a, b]$ with $a > 0$, we shift the whole domain to $[0, b - a]$. Based on the variable encodings, we define the state encoding as follows:

$$e : S \rightarrow \mathbb{B}^{m_1 + \dots + m_n}, s \mapsto (e_{v_1}(y_1), \dots, e_{v_n}(y_n))$$

where a state s is represented as a tuple (y_1, \dots, y_n) of variable values and m_j is the number of binary variables to encode the variable v_j .

In addition, each variable $v \in Var$ is itself represented by an MTBDD M_v , which is based on the encoding function e_v . The variable MTBDD M_v is defined by the function $f_v : \mathbb{B}^m \rightarrow \mathbb{R}$, such that

$$f_v[e_v(i)] = \underline{v}(i)$$

for all $0 \leq i \leq 2^m - 1$.

If the vector \underline{v} does not contain exactly 2^m entries, we add 0-entries beyond the upper bound of the domain, such that $\underline{v} = (a, a + 1, \dots, b - 1, b, 0, \dots, 0)$ and \underline{v} contains 2^m entries. In order to guarantee only valid states with respect to the variable domain, we define an MTBDD $M_{r(v)}$. We refer to $r(v)$ as the range of variable v . The MTBDD $M_{r(v)}$ is defined by the function $f_{r(v)}$, which maps the encoding of index i to 1, if and only if $\underline{v}(i)$ is not an additional 0-entry:

$$f_{r(v)}[e_v(i)] = \begin{cases} 0 & \text{if } \underline{v}(i) \text{ is an additional 0-entry} \\ 1 & \text{otherwise} \end{cases}$$

If we consider a transition between two states s and s' , where the variable v does not change, an identity MTBDD $M_{id(v)}$ represents this behaviour. The MTBDD $M_{id(v)}$ takes two variable encodings as input. The first encoding represents the old value for v , respectively the second encoding with primed variables the new value after the transition. The MTBDD $M_{id(v)}$ is formally defined by the function:

$$f_{id(v)}[e_v(i), e_v(i')] = \begin{cases} 1 & \text{if } \underline{v}(i) = \underline{v}(i') \\ 0 & \text{otherwise} \end{cases}$$

3.1. Model Representation

For a set of variables $Var' = \{v_1, v_2, \dots, v_m\} \subseteq Var$, the MTBDD $M_{id(Var')}$ represents the identity function of all variables in Var' , i.e.

$$f_{id(Var')} = f_{id(v_1)} \cdot f_{id(v_2)} \cdot \dots \cdot f_{id(v_m)}$$

Example 3.1 (Variable MTBDDs). Figure 3.1 illustrates the MTBDDs of the variable $t \in [2, 4]$, which is taken from the second module of P_{ex1} (2.1). In particular, Figure 3.1(a) illustrates the variable MTBDD M_t over (x_1, x_0) . The corresponding encoding function $e_t : \{0, 1, 2, 3\} \rightarrow \mathbb{B}^2$ requires a shift of the whole domain to $[0, 2]$, such that

$$f_t[e_t(0)] = f_t(0, 0) = 2, \quad f_t[e_t(1)] = f_t(0, 1) = 3, \quad f_t[e_t(2)] = f_t(1, 0) = 4$$

The MTBDD $M_{r(t)}$ (Figure 3.1(b)) covers the range of t and the Figure 3.1(c) the identity function.

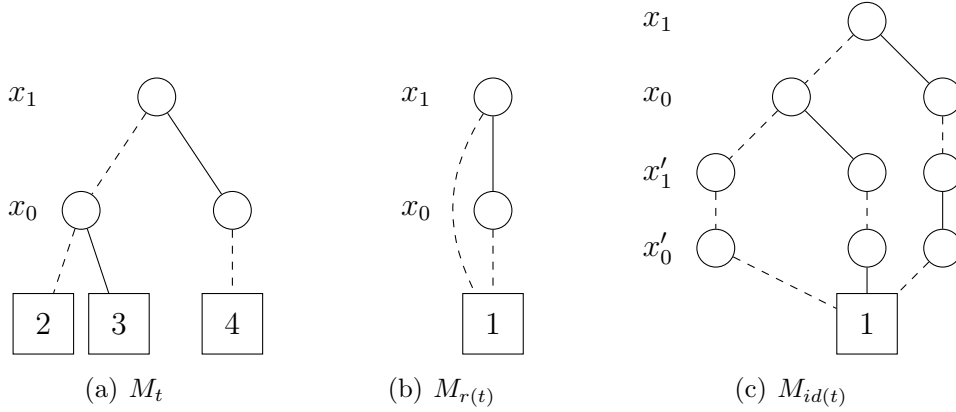


Figure 3.1.: MTBDDs for the variable $t \in [2, 4]$.

3.1.2. Expressions

Expressions are mainly used to represent update functions and atomic propositions. In the implementation we allow the expressions over a set of variables Var shown in following Definition 3.1, where we distinguish between general expressions and boolean ones.

Definition 3.1 (Expressions).

All expressions $Expr^{Var}$ over the set of variables Var are defined by the grammar:

- expressions

$$\Psi ::= \Phi \mid c_{\mathbb{R}} \mid v \mid -\Psi \mid (\Psi_1 \circ \Psi_2) \mid \text{IF } \Phi \text{ THEN } \Psi_1 \text{ ELSE } \Psi_2$$

where $\circ \in \{+, -, \cdot, /\}$, $c_{\mathbb{R}} \in \mathbb{R}$ and $v \in Var$. Φ is a boolean expression.

- boolean expressions

$$\Phi ::= c_{0,1} \mid v_{0,1} \mid \neg\Phi \mid (\Phi_1 \wedge \Phi_2) \mid (\Psi_1 \circ \Psi_2) \mid IF \Phi THEN \Phi_1 ELSE \Phi_2$$

where $\circ \in \{=, \neq, <, >, \leq, \geq\}$, $c_{0,1} \in \{0, 1\}$ and $v_{0,1} \in Var$ is a boolean variable (i.e. $VarDomain(v_{0,1}) = [0, 1]$). Ψ_1 and Ψ_2 are expressions.

The Definition 3.2 provides some corresponding MTBDD operations defined by functions. Note that for a variable expression $\Psi = v$, we use the variable MTBDD M_v defined in the previous Section 3.1.1. Given a variable valuation $val \in Val^{Var}$, we evaluate an expression $e \in Expr^{Var}$ to a certain value in \mathbb{R} . In particular, boolean expressions evaluate only to boolean values true ($= 1$) and false ($= 0$). Let $b \in Expr^{Var}$ be a boolean expression. We write $val \models b$, if and only if val satisfies b . Each atomic proposition $a \in AP$ is connected with a set of states labelled with a . In practice this set of states is characterised by a boolean expression $b_a \in Expr^{Var}$, where a state $s \in Val^{Var}$ is labelled with a , if $s \models b_a$.

Definition 3.2 (MTBDD Operations).

- $\Psi = IF \Phi_1 THEN \Psi_1 ELSE \Psi_2$

$$f_\Psi[\underline{x}] = \begin{cases} f_{\Psi_1}[\underline{x}] & \text{if } f_{\Phi_1}[\underline{x}] = 1 \\ f_{\Psi_2}[\underline{x}] & \text{otherwise} \end{cases}$$

- $\Psi = c$

$$f_\Psi[\underline{x}] = c, \text{ where } c \in \mathbb{R}.$$

- $\Psi = v$

$$f_\Psi[\underline{x}] = f_v[\underline{x}]$$

- $\Phi = \Phi_1 \wedge \Phi_2$

$$f_\Phi[\underline{x}] = \begin{cases} 1 & \text{if } f_{\Phi_1}[\underline{x}] = 1 \wedge f_{\Phi_2}[\underline{x}] = 1 \\ 0 & \text{otherwise} \end{cases}$$

- $\Phi = (\Psi_1 = \Psi_2)$

$$f_\Phi[\underline{x}] = \begin{cases} 1 & \text{if } f_{\Psi_1}[\underline{x}] = f_{\Psi_2}[\underline{x}] \\ 0 & \text{otherwise} \end{cases}$$

Example 3.2 (Expression MTBDD). Figure 3.2 illustrates the MTBDD representing the (boolean) expression $\Phi = (v' = v + 1)$ over the variables $Var = \{v, v'\}$. This MTBDD is build recursively by combination of all sub-formulas, where the variable v is encoded with $\underline{x} = (x_1, x_0)$ and v' with $\underline{x}' = (x'_1, x'_0)$. The corresponding boolean function is given by:

$$f_\Phi[\underline{x}, \underline{x}'] = \begin{cases} 1 & \text{if } (f_{v'}[\underline{x}'] = f_v[\underline{x}] + f_1) \\ 0 & \text{otherwise} \end{cases}$$

3.1. Model Representation

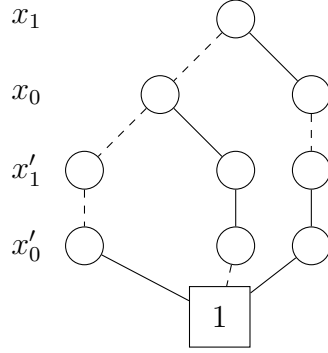


Figure 3.2.: The expression $v' = v + 1$ in MTBDD representation ($v, v' \in [0, 3]$).

3.1.3. Commands

Now we consider the transition probability matrix \mathbf{P} of the given model. As described in Section 2.4, the matrix is represented by an MTBDD P . For the state encoding we use the structural encoding function $e : S \rightarrow \mathbb{B}^n$, introduced in Section 3.1.1. For $Mod_i = (Var_i, Act_i, \mathbf{C}_i)$, we translate a command $c = (a, g, (p_1, U_1), \dots, (p_m, U_m)) \in \mathbf{C}_i$ into an MTBDD M_c . In the theoretical background we have defined an update function $U_j : Val^{Var} \rightarrow Val^{Var_i}$, which specifies the destination state $s' \in S$ based on the source one s . In practice the function U_j is represented by an expression of the form $v'_1 = e_1 \wedge \dots \wedge v'_k = e_k$, where $e_l \in Expr^{Var}$ and $Var_i = \{v_1, \dots, v_k\}$. Intuitively, the term $v'_l = e_l$ denotes the update of the variable v_l , which gets a new value based on the evaluation of the expression e_l assigned. Note that if a variable v_l remains unchanged during an update, we omit the term of the form $v'_l = v_l$ in the syntactical representation. Implicitly, all variables $v \in Var \setminus Var_i$ do not change, because only variables in Var_i can be written by commands in \mathbf{C}_i . Hence, we represent any update expression in c by an MTBDD with the function $f_{U_j} : \mathbb{B}^{n+n} \rightarrow \mathbb{R}$ (see Section 3.2). In practice, we first represent the expression e_l by an MTBDD over the non-primed variables. Afterwards, we take the MTBDD M_{v_l} over the primed variables and apply the equality-operator on both MTBDDs. The probability $p_j \in [0, 1]$ of the update U_j is interpreted as constant MTBDD with function f_{p_j} . Overall, the MTBDD M_c representing the command c is defined by the function $f_c : \mathbb{B}^{n+n} \rightarrow \mathbb{R}$:

$$f_c [\underline{x} = e(s), \underline{x}' = e(s')] = f_{g_c} [\underline{x}] \cdot \left(\sum_{j=1}^m f_{p_j} \cdot f_{U_j} [\underline{x}, \underline{x}'] \right)$$

Note that the guard g_c is a boolean expression over all variables Var , which defines a subset of states $S_{g_c} \subseteq S$ satisfying this guard. An MTBDD with boolean function f_{g_c} represents this set S_{g_c} , such that $f_{g_c}[e(s)] = 1$ if and only if $s \in S_{g_c}$, otherwise $f_{g_c}[e(s)] = 0$.

Example 3.3 (Command MTBDD). The Figure 3.3 shows the MTBDD M_c representing an example command c , which is given by $(s \in [0, 3])$:

$$[] \quad s=1 \rightarrow 0.3 : (s'=2) + 0.7 : (s'=3);$$

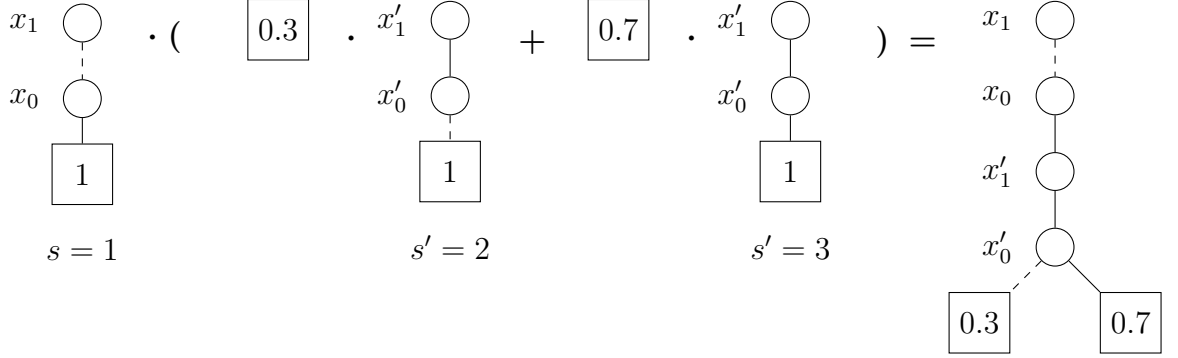


Figure 3.3.: The MTBDD M_c representing the example command c .

3.1.4. Synchronising Actions and Non-determinism

Now we consider the synchronising actions Act in the probabilistic program P and the non-deterministic choices, which arise during the construction. The steps of the construction are described in the following.

First, we are confronted with local non-determinism, i.e. two commands with the same action and overlapping guards within a module. Let n_l be the maximum number of non-deterministic choices in a state $s \in S$ of the MDP, where n_l is bounded by the number of commands in P . Let $Mod_i = (Var_i, Act_i, \mathbf{C}_i)$ be a module. To resolve the local non-determinism in \mathbf{C}_i , we partition the set of commands \mathbf{C}_i into subsets $\mathbf{C}_{i,1}, \dots, \mathbf{C}_{i,n_l}$, such that for two commands $c_1, c_2 \in \mathbf{C}_{i,l}$ with $a_{c_1} = a_{c_2}$:

$$\forall val \in Val^{Var_i}. (val \not\models g_{c_1} \vee val \not\models g_{c_2})$$

Intuitively, each set $\mathbf{C}_{i,l}$ contains only commands without overlapping guards for the same action. The index l denotes the l -th non-deterministic choice, where $0 \leq l \leq n_l$. Note that if a module has fewer non-deterministic choices than n_l , we get some empty subsets $\mathbf{C}_{i,l} = \emptyset$. In the following, we combine all sets of commands to the whole transition matrix. We distinguish between commands with the non-synchronising τ -action and commands with a synchronising action $a \in Act$.

First, we combine all commands for a specific local non-deterministic choice l . In particular, the commands with a τ -action for a certain module Mod_i represent always a non-synchronising behaviour towards the other modules. The variables $Var \setminus Var_i$ remain unchanged during these τ -transitions, thus we multiply the identity matrix of

3.1. Model Representation

the other modules:

$$f_{\tau_i, l}[\underline{x} = e(s), \underline{x}' = e(s')] = \begin{cases} \left(\sum_{\{c \in \mathbf{C}_{i, l}, a_c = \tau\}} f_c[\underline{x}, \underline{x}'] \right) \cdot \prod_{j \neq i} f_{id(Var_j)}[\underline{x}, \underline{x}'] & \text{if } \mathbf{C}_{i, l} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

In contrast, for each synchronising action $a \in Act_i$ in module Mod_i we synchronise between other modules $Mod_j \in \mathbf{M}$, which also contain this action a . The commands in the remaining modules without action a move in a non-synchronising way just like the τ -labeled commands:

$$f_{a, l}[\underline{x} = e(s), \underline{x}' = e(s')] = \prod_{\{i | a \in Act_i\}} \left(\sum_{\{c \in \mathbf{C}_{i, l}, a_c = a\}} f_c[\underline{x}, \underline{x}'] \right) \cdot \prod_{\{i | a \notin Act_i\}} f_{id(Var_i)}[\underline{x}, \underline{x}']$$

So far we have combined the commands for each local non-deterministic choice l , for each module in \mathbf{M} and for each action $Act \cup \{\tau\}$. We define a tuple \underline{z} of boolean variables to encode these non-deterministic choices, i.e. actions, scheduling between modules and local non-determinism. The corresponding encoding function e_A is defined by:

$$e_A : Act \times \mathbf{M} \times \{1, \dots, n_l\} \rightarrow \mathbb{B}^m, (a, i, l) \mapsto \underline{z} = (z_m, \dots, z_1)$$

where $m = |Act| + |\mathbf{M}| + n_l$.

Finally, we combine all subsets of commands and add the specific non-deterministic variables \underline{z} for each subset:

$$f_P[\underline{z} = e_A(a, i, l), \underline{x} = e(s), \underline{x}' = e(s')] = \begin{cases} f_{\tau_i, l}[\underline{x}, \underline{x}'] & \text{if } a = \tau \\ f_{a, l}[\underline{x}, \underline{x}'] & \text{if } a \in Act \\ 0 & \text{otherwise} \end{cases}$$

The MTBDD M_P representing the transition matrix \mathbf{P} of the given model is defined by this boolean function f_P [11].

Example 3.4 (Partitioning and Transition Matrix). Figure 3.4 illustrates the partitioning of all commands in the probabilistic program P_{ex2} (Listing 3.1). Each partitioning contains only non-overlapping guards, i.e. no local non-determinism. Based on the partitioning, we construct the whole MTBDD representing the transition matrix, which is shown in Figure 3.5. Overall we use four variables $\underline{z} = (z_3, z_2, z_1, z_0)$ to encode the non-deterministic choices. In particular, the variables z_3, z_2 encode the action, where $\tau = 00$, $\alpha = 10$ and $\beta = 01$. The variable z_1 encodes the choice between modules. Note that z_1 always equals 0, because P_{ex2} contains only one module. The last variable z_0 is used to encode the local non-determinism. There are at most two non-deterministic choices in a state $s \in S$. Thus, $z_0 = 0$ and $z_0 = 1$ represent the first and the second choice, respectively.

Listing 3.1: A probabilistic program P_{ex2} .

```

mdp
module mod1
  t: [0..2] init 0;

  [beta] t=0 -> 1: (t'=1);
  [] t=1 -> 1: (t'=2);
  [alpha] t=2 -> 0.5: (t'=1) + 0.5: (t'=2);
  [beta] t=2 -> 0.5: (t'=0) + 0.5: (t'=1);
  [beta] t=2 -> 1: (t'=2);
endmodule

```

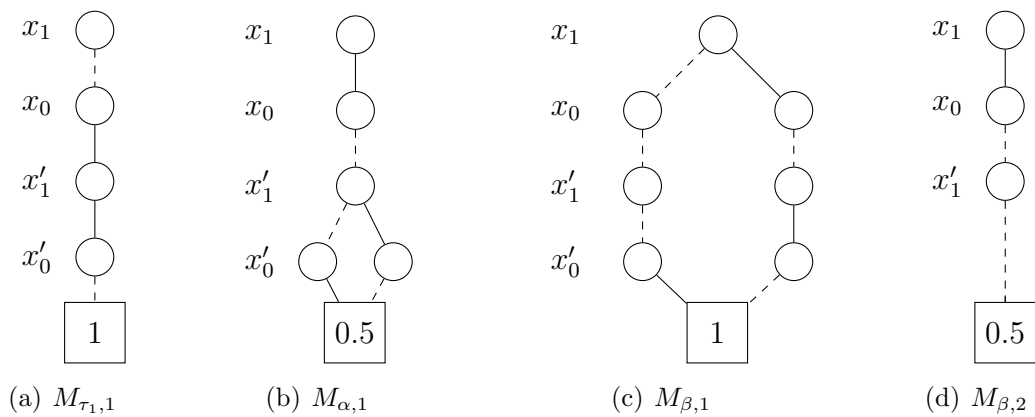


Figure 3.4.: Partitioning of the probabilistic program P_{ex2} .

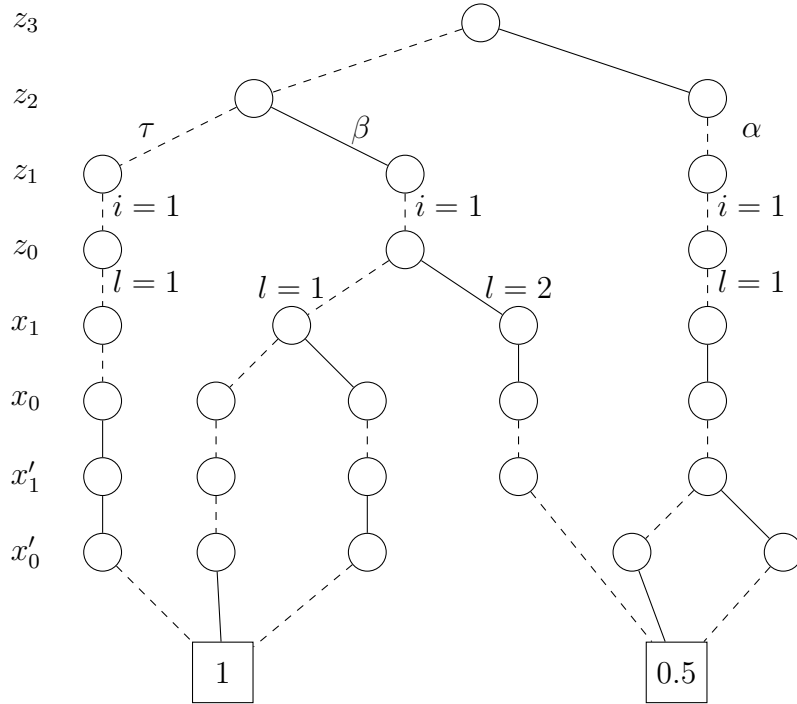


Figure 3.5.: The transition MTBDD of P_{ex2} .

3.2. Model Checking Algorithms

In the last Section 3.1 we have covered the construction and succinct storage of probabilistic models with MTBDDs. Now we present the model checking algorithms for these models. Thereby, we mainly focus on MDP model checking, because the MDP algorithms are also applicable to DTMCs. First, we consider PCTL state formulas as introduced in Section 2.2. In the next Section 3.2.2 we extend the definitions to PCTL path formulas.

3.2.1. State Formulas

Given a state formula Φ , we determine the set of states satisfying this formula. To be consistent with the model representation, each satisfaction set is represented by an MTBDD (in fact a BDD). Thus, similar to the recursive definition of a PCTL formula, we define boolean functions $f : \mathbb{B}^n \rightarrow \{0, 1\}$ for the operators *true*, *a*, \neg , \wedge in a state formula:

- Case '*true*':

The satisfaction set of *true* includes all states of the given model, i.e.

$$f_{Sat(true)}[\underline{x} = e(s)] = 1$$

- Case 'a':

An atomic proposition $a \in AP$ is encoded by the boolean function $f_a : \mathbb{B}^n \rightarrow \{0, 1\}$ in the following manner. If a state $s \in S$ is labelled with a , i.e. s satisfies the boolean expression connected with a , then $f_a[e(s)] = 1$. Otherwise, $e(s)$ is mapped to 0. Based on the function f_a , we define the satisfaction set:

$$f_{Sat(a)}[\underline{x} = e(s)] = f_a[\underline{x}]$$

- Case '¬':

For the PCTL formula $\neg\Phi$, the operator \neg inverts the satisfaction set of the sub-formula Φ , i.e. $Sat(\neg\Phi) = S_{reach} \setminus Sat(\Phi)$. The set $S_{reach} \subseteq S$ denotes all reachable states.

$$f_{Sat(\neg\Phi)}[\underline{x} = e(s)] = \begin{cases} 1 - f_{Sat(\Phi)}[\underline{x}] & \text{if } s \in S_{Reach} \\ 0 & \text{otherwise} \end{cases}$$

- Case '∧':

The satisfaction set for the formula $\Phi_1 \wedge \Phi_2$ is developed by recursive combination of the satisfaction sets for the sub-formulas Φ_1 and Φ_2 :

$$f_{Sat(\Phi_1 \wedge \Phi_2)}[\underline{x} = e(s)] = f_{Sat(\Phi_1)}[\underline{x}] \cdot f_{Sat(\Phi_2)}[\underline{x}]$$

Note that for PCTL state formulas, there is no difference between MDP and DTMC model checking.

3.2.2. Path Formulas

Based on Section 3.2.1, we provide algorithms to compute the probability of a given PCTL path formula with the operators $\bigcirc, \mathbf{U}^{\leq k}, \mathbf{U}$. First, we focus on the most complex calculations for the until operator (\mathbf{U}). For an until formula $\varphi = \Phi_1 \mathbf{U} \Phi_2$, we apply some preprocessing algorithms via graph analysis. These preprocessing algorithms determine all states, which satisfy the formula φ either with probability 0 (S_0) or 1 (S_1). Thus, we can reduce the effort in the following calculations, because we consider only states with a probability $0 < p < 1$. We denote this set of states by $S_? = S \setminus (S_0 \cup S_1)$. Furthermore, preprocessing avoids some round-off errors regarding the extremal values (0 and 1). For MDPs, there are four different preprocessing algorithms depending of determining minimal or maximal probabilities, respectively probabilities equal to zero or one. One algorithm, called *Prob0A*, is exemplary shown in the following Algorithm 3.1. The *Prob0A* algorithm is quite simple compared to the other algorithms, which are shown in [11].

Algorithm 3.1 Prob0A($f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}]$)

```

 $f_{res}[\underline{x}] := f_{\Phi_2}[\underline{x}]$ 
 $done := false$ 
while  $done = false$  do
     $f_{tmp}[\underline{x}'] = SwapVariables(f_{res}[\underline{x}], \underline{x}, \underline{x}')$ 
     $f_{tmp}[\underline{z}, \underline{x}] = ExistsAbstract(f_{tmp}[\underline{x}'] \cdot f_{P'}[\underline{x}, \underline{x}'], \underline{x}')$ 
     $f_{tmp}[\underline{x}] = (f_{tmp}[\underline{x}] \wedge f_{\Phi_1}[\underline{x}]) \vee f_{\Phi_2}(\underline{x})$ 

    if  $f_{tmp}[\underline{x}] = f_{res}[\underline{x}]$  then
         $done := true$ 
    end if
     $f_{res}[\underline{x}] = f_{tmp}[\underline{x}]$ 
end while
return  $\neg f_{res}[\underline{x}]$ 
    
```

Intuitively, the algorithm *Prob0A* determines all states, which independent of the non-deterministic choice surely reach a state not satisfying Φ_1 before a Φ_2 -state or never reach a state satisfying Φ_2 . The states are determined via graph analysis, thus the algorithm uses a simplified transition matrix \mathbf{P}' . First, we initialise the set of states *res* represented by the function f_{res} with all states satisfying Φ_2 . In the while loop we add further states to this set, if a state satisfies Φ_1 and there exists an action potentially leading to a state currently in *res*. We exit the while loop, if there are not any new states during one iteration. Overall, the number of iterations is limited by the number of states in the model, i.e. $|S|$. In the end, we take the complementary set $S \setminus res$, such that for all states in this set, there does not exist a possible resolution of non-determinism with a reachability probability greater than zero.

Note that we determine the set *res* via graph analysis, because we can abstract from the actual probabilities in the system and map all non-zero probabilities to one to indicate that there is some transition. Thus, the *Prob0A* algorithm uses an MTBDD P' representing a modified transition matrix, which is defined by:

$$f_{P'}[\underline{z}, \underline{x}, \underline{x}'] = \begin{cases} 1 & \text{if } \exists \underline{z}' \in \mathbb{B}^m. f_P[\underline{z}', \underline{x}, \underline{x}'] > 0 \\ 0 & \text{otherwise} \end{cases}$$

The operation $SwapVariables(f_{res}[\underline{x}], \underline{x}, \underline{x}')$ renames all variables in $f_{res}[\underline{x}]$ from \underline{x} into \underline{x}' and vice versa.

The algorithm *Prob0A* requires another operation *ExistsAbstract* on MTBDDs. This operation takes an MTBDD and a tuple of variables, from which we want to abstract, as input. Formally it is defined as follows:

$$ExistsAbstract(f[\underline{z}, \underline{x}], \underline{z}) = \begin{cases} 1 & \exists \underline{z}' \in \mathbb{B}^m. f[\underline{z}', \underline{x}] > 0 \\ 0 & \text{otherwise} \end{cases}$$

For *Prob0E*, we also require *UniversalAbstract*, which is similar defined by

$$UniversalAbstract(f[\underline{z}, \underline{x}], \underline{z}) = \begin{cases} 1 & \forall \underline{z}' \in \mathbb{B}^m. f[\underline{z}', \underline{x}] > 0 \\ 0 & otherwise \end{cases}$$

After the preprocessing step, we compute the exact probabilities of the remaining states $S_?$. For MDP model checking, we use an algorithm called value iteration [22]. Algorithm 3.2 shows this algorithm for either minimal or maximal probabilities depending on the input parameter $Y \in \{min, max\}$. Let $\Phi_1 \mathbf{U} \Phi_2$ be the PCTL formula. The algorithm also takes the two satisfaction sets of the sub-formulas Φ_1 and Φ_2 represented by an MTBDD (f_{Φ_1} and f_{Φ_2}) as input.

Algorithm 3.2 *MDPUntil*($Y, f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}]$)

```

if  $Y = min$  then
   $f_0[\underline{x}] := Prob0E(f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}])$ 
   $f_1[\underline{x}] := Prob1A(f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}])$ 
end if
if  $Y = max$  then
   $f_0[\underline{x}] := Prob0A(f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}])$ 
   $f_1[\underline{x}] := Prob1E(f_{\Phi_1}[\underline{x}], f_{\Phi_2}[\underline{x}])$ 
end if
 $f_?[\underline{x}] := \neg(f_1[\underline{x}] \vee f_0[\underline{x}])$ 
 $f_{P'}[z, \underline{x}, \underline{x}'] := f_?[\underline{x}] \cdot f_{P'}[z, \underline{x}, \underline{x}']$ 
 $f_{res}[\underline{x}] := f_1[\underline{x}]$ 
 $done := false$ 
while  $done = false$  do
   $f_{tmp}[\underline{x}'] = SwapVariables(f_{res}[\underline{x}], \underline{x}, \underline{x}')$ 
   $f_{tmp}[z, \underline{x}] = ExistsAbstract(f_{tmp}[\underline{x}'] \cdot f_{P'}[z, \underline{x}, \underline{x}'], \underline{x}')$ 
   $f_{tmp}[\underline{x}] = Abstract_Y(f_{tmp}[z, \underline{x}], \underline{z})$ 
   $f_{tmp}[\underline{x}] = f_{tmp}[\underline{x}] + f_1[\underline{x}]$ 

  if  $Precision(f_{tmp}[\underline{x}], f_{res}[\underline{x}]) < \epsilon$  then
     $done := true$ 
  end if
   $f_{res}[\underline{x}] = f_{tmp}[\underline{x}]$ 
end while
return  $f_{res}[\underline{x}]$ 

```

The function *Abstract_Y* intuitively abstracts from the given variables by picking either the minimal or maximal function value depending of Y :

$$Abstract_M(f[\underline{z}, \underline{x}], \underline{z}) = \begin{cases} \min_{\underline{z}' \in \mathbb{B}^m} \{f[\underline{z}', \underline{x}]\} & if \ Y = min \\ \max_{\underline{z}' \in \mathbb{B}^m} \{f[\underline{z}', \underline{x}]\} & if \ Y = max \end{cases}$$

3.2. Model Checking Algorithms

The function $Precision(f_{tmp}[\underline{x}], f_{res}[\underline{x}])$ calculates the maximal difference between the function values, i.e. $\max_{\underline{x}' \in \mathbb{B}^n} |f_{tmp}[\underline{x}'] - f_{res}[\underline{x}']|$. If this difference is lower than a specified threshold ϵ , the algorithm terminates. If we use the introduced preprocessing algorithms, it can be proven that this fix-point iteration always terminates for a non-zero ϵ [23].

Based on the algorithm for the until operator, we derive the other algorithms for bounded until ($\mathbf{U}^{\leq k}$) and next (\mathbf{O}). The main difference between these algorithms is the number of times the matrix-vector product needs to be computed. In the value iteration algorithm for until, we loop until we get a convergence determined by the threshold ϵ . For the bounded until formula $\Phi_1 \mathbf{U}^{\leq k} \Phi_2$, we make all target states Φ_2 absorbing and loop exactly k times. Finally, for the next operator \mathbf{O} , we loop only once.

4. Evaluation

The implementation described in the previous Chapter 3 is realised in about 4000 lines of C++ code in the model checker STORM. In this Chapter we first provide some performance benchmarks of some common models (Section 4.1). In particular, we compare the results to the state-of-the-art model checker PRISM [1]. In the following, in Section 4.2 we test a method, called Chaining, to calculate all reachable states of a given model. In Section 4.3 we consider the meta-variable ordering in a given probabilistic program. In particular, we show the significant influence on the size of the MTBDD representing the transition matrix. We also provide a first heuristic to order all meta-variables.

In general, we consider the following probabilistic programs:

- synchronous leader election protocol (DTMC) [24]
Given a number of processors in a synchronous ring structure, these processors are able to elect a leader based on the protocol.
- crowds protocol (DTMC) [25]
The crowds protocol provides an opportunity for anonymous web browsing. All packets of a community member are routed in a randomised fashion, such that the model is represented by a DTMC. Note that we use a slightly modified version of the probabilist program (see Appendix A).
- asynchronous leader election protocol (MDP) [24]
This protocol is a variant of the synchronous leader election protocol, where all processors are not synchronised any more. Thus, we also get non-deterministic choices represented by an MDP.
- IEEE 802.11 Wireless LAN (MDP) [26]
This program models the carrier sense multiple access with collision avoidance (short CSMA/CA) mechanism of the WLAN standard 802.11. All stations in the network non-deterministically try to send a packet, while they use a randomised backoff mechanism to avoid collisions.
- randomised consensus shared coin protocol (MDP) [27]
The shared coin protocol models a collective random walk of processes. It requires the number of processes N and a constant $K > 1$ as input.

- firewire root contention protocol (MDP) [28]
This protocol models the leader election protocol of the IEEE standard 1394. The IEEE 1394 High Performance Serial Bus (called FireWire) is used to transport video and audio data within a network.
- bounded retransmission protocol (MDP) [29]
The bounded retransmission protocol, short brp, is a variant of the alternating bit protocol. This protocol sends a file divided into chunks, while it allows only a limited number of tries for each chunk.

All benchmarks have been run on a machine with an Intel Core i7 processor and 4GB main memory, running Apple Mac OS X 10.9 Mavericks.

4.1. Performance

In this section we measure the performance of our implementation in STORM compared to the one in PRISM. We show that the performance of the STORM implementation is competitive to the state-of-the-art model checker PRISM. First, we consider the time for the construction of the transition matrix, whereby the algorithms in PRISM and STORM are quite similar. The following Table 4.1 shows the results, where "MTBDD" denotes the size (i.e. the number of nodes) of the MTBDD representing the transition matrix.

model	N	K	states	MTBDD	construction (s)	
					PRISM	STORM
synchronous	5	8	131522	1540582	0.171	0.111
leader	5	9	236746	3345130	0.297	0.203
chrowds	15	5	586242	183680	2.836	2.450
	20	5	2036647	831299	213.638	185.310
asynchronous	8		18674484	392068	0.332	0.205
leader	9		167748115	868229	0.434	0.281
	10		1516496449	1726990	0.507	0.339
csma	4	4	133301572	553488	0.285	0.140
	4	6	39051159469	3589198	0.483	0.590
coin	4	4	43136	2336	0.111	0.056

Table 4.1.: PRISM vs. STORM (construction).

Beside the construction of the transition matrix, we also perform a reachability analysis on the state space. Thus, we reduce the transition matrix to only reachable states, because it significantly reduces the effort for the model checking algorithm later. Then, for any PCTL formula, we calculate the corresponding probabilities only for the reachable states. The following Table 4.2 shows the performance of the reachability analysis in STORM and PRISM. The "factor" is defined by the time in PRISM divided by the

4.1. Performance

time in STORM. While the algorithms are nearly identical in both implementations, the implementation in STORM is slightly faster than the one in PRISM, i.e. the construction requires up to 41 percent less time. One reason might be some low-level details leading to a faster execution.

model	N	K	states	MTBDD	reachability (s)		factor
					PRISM	STORM	
synchronous	5	8	131522	1540582	22.74	16.05	1.41
leader	5	9	236746	3345130	56.14	41.32	1.35
crowds	15	5	586242	183680	0.91	0.88	1.03
	20	5	2036647	831299	8.01	5.97	1.34
asynchronous	8		18674484	392068	46.38	36.17	1.28
leader	9		167748115	868229	204.54	141.11	1.44
	10		1516496449	1726990	611.81	401.03	1.52
csma	4	4	133301572	553488	65.55	38.37	1.71
	4	6	39051159469	3589198	626.62	411.83	1.52
coin	4	4	43136	2336	0.09	0.07	1.28

Table 4.2.: PRISM vs. STORM (reachability).

For further improvements, the relation between construction and reachability might be interesting. The comparison is illustrated in Figure 4.1. Usually the time spent on constructing the model is a lot less than computing the reachable state space. An exception is the crowds protocol, where around 85 percent of the time is spent on the construction of the transition matrix. In Section 4.3 we see that the percentage for the crowds protocol heavily depends on the variable ordering.

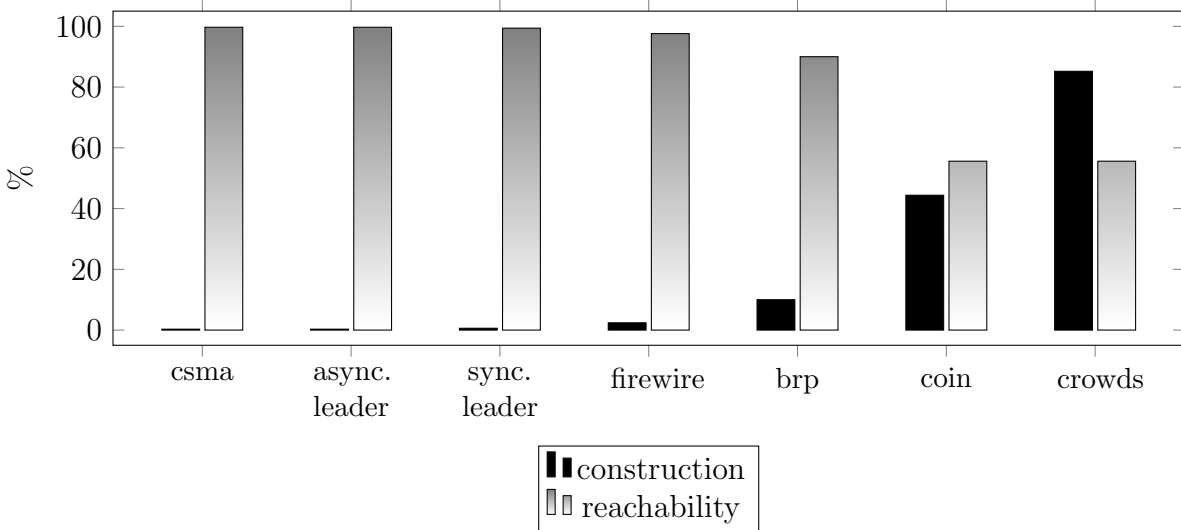


Figure 4.1.: Relative time for construction and reachability.

4.2. Chaining

In the last Section 4.1 we have seen that the reachability analysis requires a significant amount of time, but it is inevitable to reduce the effort for verification. Now we take a closer look at the algorithm to calculate all reachable states. The standard algorithm is a breadth first search (BFS) illustrated in Algorithm 4.3.

Algorithm 4.3 BFS(s_{init}, A, \mathbf{P})

```

All = { $s_{init}$ }
repeat
   $changed = false$ 
   $New = All \cup \{s' \mid \mathbf{P}(s, a, s') > 0, s \in All, a \in A\}$ 
  if  $New \neq All$  then
     $changed := true$ 
  end if
   $All = New$ 
until  $changed = false$ 
return All

```

In the STORM implementation we have also tested another approach called Chaining. This approach has been successfully applied to petri-nets. In the Chaining algorithm (Algorithm 4.4) we divide the MTBDD M_P representing the whole transition matrix into sub-MTBDDs $M_{P,a_1}, \dots, M_{P,a_n}$ representing the matrix for a certain action. We partially update the reachable states in each iteration by multiplying these sub-MTBDDs for any action. Note that for models with just one action, i.e. $|Act| = 1$, the Chaining algorithm and the BFS are identical.

Algorithm 4.4 Chaining(s_{init}, A, \mathbf{P})

```

All = { $s_{init}$ }
repeat
   $changed = false$ 
   $New = All$ 
  for all  $\alpha \in A$  do
     $New = New \cup \{s' \mid \mathbf{P}(s, \alpha, s') > 0, s \in New\}$ 
  end for
  if  $New \neq All$  then
     $changed := true$ 
  end if
   $All = New$ 
until  $changed = false$ 
return All

```

4.2. Chaining

model	N	K	iterations		memory peak (MB)		time (s)	
			Chaining	BFS	Chaining	BFS	Chaining	BFS
synchronous	5	8	6	7	319	317	15.55	15.38
leader	5	9	6	7	660	662	40.31	39.98
	6	8	7	8	1495	1508	177.87	177.10
asynchronous	8		29	94	118	187	45.70	36.17
leader	9		35	112	325	335	169.30	141.11
	10		41	131	723	629	555.03	401.03
csma	4	4	152	179	175	174	61.59	38.37
	4	6	162	195	844	725	1447.34	411.83

Table 4.3.: Chaining vs. BFS (reachability).

The experimental results with Chaining compared to the ordinary BFS are shown in Table 4.3. Overall, the Chaining algorithm does not improve the reachability analysis, i.e. it does not lead to a reduced time or less memory requirements. More specifically, the Chaining algorithm reduces the number of iterations in the while loop compared to the BFS (see Figure 4.2). But the Figure 4.3 illustrates that it also requires more time per iteration leading to an overall worse performance. One reason are the more complex computations required for the partial updates.

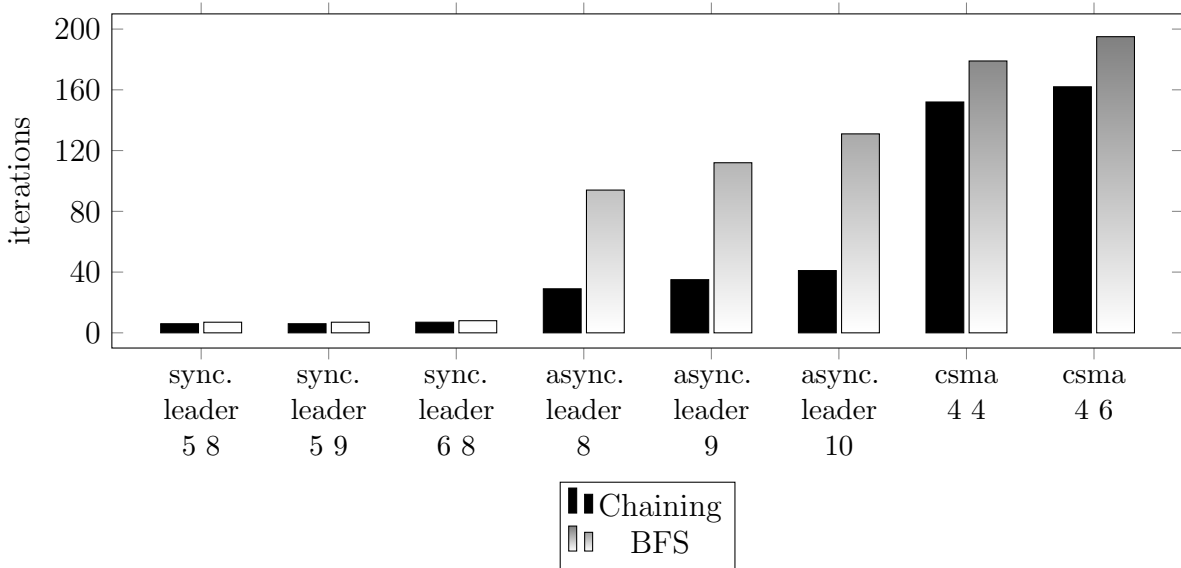


Figure 4.2.: Chaining vs. BFS (iterations).

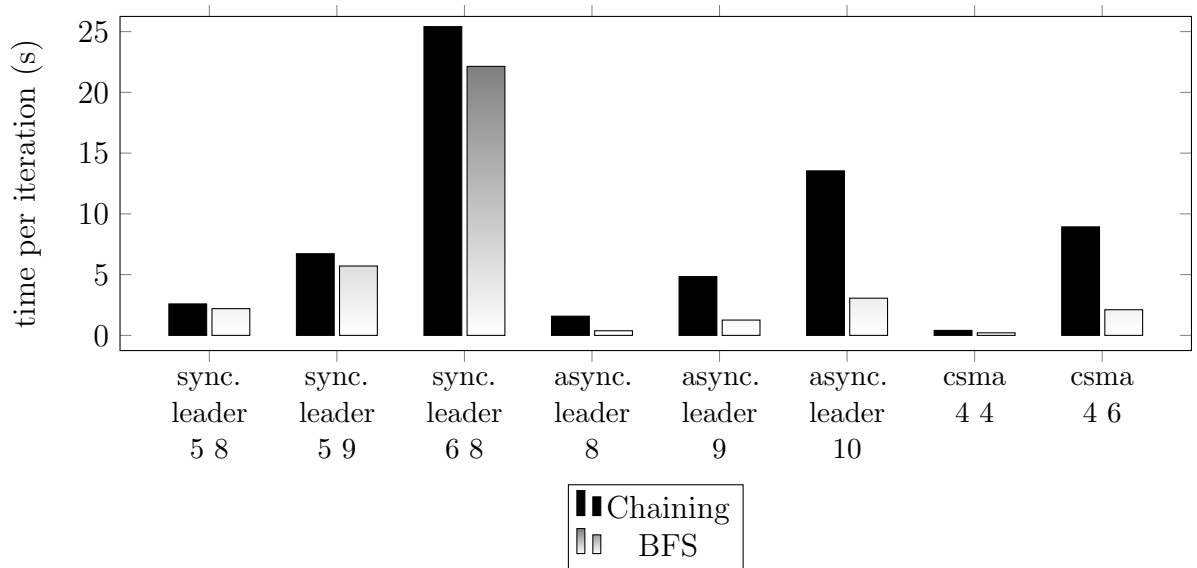


Figure 4.3.: Chaining vs. BFS (time per iteration).

4.3. Variable Ordering

In this section we consider different variable orderings for the MTBDD representing the transition matrix. The size, i.e. the number of nodes, of the MTBDD significantly influences the performance of all MTBDD operations. Figure 4.4 illustrates two orderings of an identity MTBDD. Thereby, we change the ordering of the binary variables, which are used to encode the variables Var in the probabilistic program. In [30] the common interleaved ordering (Figure 4.4(b)) is extensively tested and it is shown that this heuristic ordering usually performs rather well. In such an interleaved ordering, a binary variable x is followed by its primed version x' , i.e. $x_0 \prec x'_0 \prec \dots \prec x_m \prec x'_m$, where the variables Var are encoded with m binary MTBDD variables.

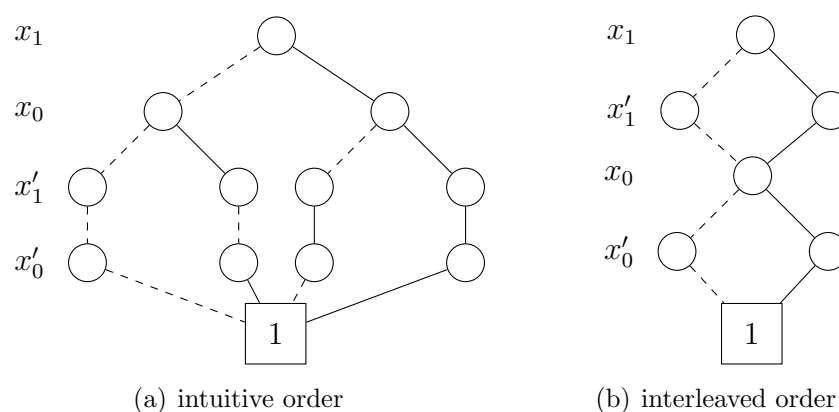


Figure 4.4.: An identity MTBDD with different variable orderings.

4.3. Variable Ordering

While the ordering of binary variables in an MTBDD has already been considered in many other papers, we take a closer look at the ordering of the variables in Var , i.e. the order of definition in the probabilistic program. In the following, we denote a variable $v \in Var$ by the term meta-variable to distinguish between the binary encoding variables x_1, \dots, x_m and original variable v .

Example 4.1 (Meta-Variable Ordering). Figure 4.5 illustrates two orderings of the example meta-variables $v_1 \in [0, 3]$ and $v_2 \in [0, 1]$, where v_1 is encoded by v_1-x_0, v_1-x_1 and v_2 by v_2-x_0 . Accordingly, the primed variables are $v_1-x'_0, v_1-x'_1$ and $v_2-x'_0$. Furthermore, both MTBDDs represent the following commands:

[] $v_1=0 \ \& \ v_2=0 \ \rightarrow \ 1: \ (v_1'=0) \ \& \ (v_2'=0);$
 [] $v_1=0 \ \& \ v_2=1 \ \rightarrow \ 1: \ (v_1'=1) \ \& \ (v_2'=1);$

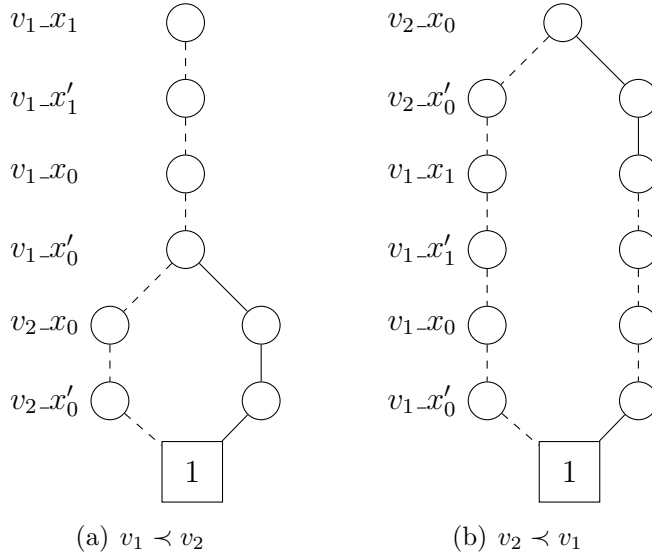


Figure 4.5.: An example MTBDD with different meta-variable orderings.

Table 4.4 provides the MTBDD size and construction time for different meta-variable orderings in the crowds protocol. The worst-case size of the MTBDD is about factor 57 greater than the optimal size. The size of the MTBDD correlates with the required memory, which ranges between $25MB$ and $1832MB$. In addition, the MTBDD size corresponds to the overall model construction time (= construction of the transition matrix + reachability analysis). For a system given in terms of a probabilistic program, we have figured out that the meta-variable ordering significantly influences the size of the MTBDD and the time for model checking, respectively. Thus, we take a closer look at a useful meta-variable ordering in the following.

meta-variable ordering	MTBDD	construction (s)	memory peak (MB)
phase,good,lastSeen,run/observe	22797	0.31	25
⋮	⋮	⋮	⋮
lastSeen,run/observe,phase,good	28119	0.69	29
⋮	⋮	⋮	⋮
phase,run/observe,lastSeen,good	852686	150.05	885
⋮	⋮	⋮	⋮
run/observe,phase,good,lastSeen	1303783	739.95	1832

Table 4.4.: Influence of the meta-variable ordering (Crowds 20_5).

We first consider programs with only one module or without a synchronising action. We have tested different meta-variable orderings within a certain module. The following Table 4.5 shows the detailed experimental results of the crowds protocol.

meta-variable ordering	MTBDD	construction(s)
phase[5], good[2], lastSeen[20], run/observe[6]	22797	0.31
good[2], phase[5], lastSeen[20], run/observe[6]	22808	0.31
phase[5], lastSeen[20], good[2], run/observe[6]	22872	0.28
good[2], lastSeen[20], run/observe[6], phase[5]	23003	0.28
lastSeen[20], good[2], phase[5], run/observe[6]	23021	0.29
lastSeen[20], phase[5], good[2], run/observe[6]	23073	0.30
phase[5], lastSeen[20], run/observe[6], good[2]	23317	0.41
lastSeen[20], phase[5], run/observe[6], good[2]	23518	0.38
lastSeen[20], run/observe[6], good[2], phase[5]	28104	0.70
lastSeen[20], run/observe[6], phase[5], good[2]	28119	0.69
good[2], lastSeen[20], run/observe[6], phase[5]	28454	0.55
lastSeen[20], good[2], run/observe[6], phase[5]	28472	0.69
phase[5], good[2], run/observe[6], lastSeen[20]	831299	81.21
good[2], phase[5], run/observe[6], lastSeen[20]	831310	180.09
phase[5], run/observe[6], lastSeen[20], good[2]	852686	150.05
phase[5], run/observe[6], good[2], lastSeen[20]	901196	106.13
good[2], run/observe[6], lastSeen[20], phase[5]	1051735	379.18
run/observe[6], lastSeen[20], good[2], phase[5]	1053524	443.02
run/observe[6], lastSeen[20], phase[5], good[2]	1053539	456.69
run/observe[6], good[2], lastSeen[20], phase[5]	1134411	801.25
good[2], run/observe[6], phase[5], lastSeen[20]	1203365	653.33
run/observe[6], phase[5], lastSeen[20], good[2]	1255273	715.90
run/observe[6], good[2], phase[5], lastSeen[20]	1286041	749.49
run/observe[6], phase[5], good[2], lastSeen[20]	1303783	739.95

Table 4.5.: All meta-variable orderings for Crowds 20_5.

4.3. Variable Ordering

The table above (Figure 4.5) illustrates the influence of the meta-variable ordering on the MTBDD size depending on the meta-variable ranges. Overall the crowds protocol with parameter 20_5 contains 24 meta-variables, which are divided into four classes with different ranges. First, there are the meta-variables `lastSeen` (with domain $[0..19]$) and `phase` (with domain $[0..4]$), followed by the boolean meta-variable `good`. The expression "run/observe" represents a group of the remaining 21 meta-variables, all with domain $[0..5]$. Because we only sort according to the meta-variable range, this group of meta-variables is treated as one static part.

As a main result, the size of the MTBDD heavily depends on the ordering of the meta-variables "lastSeen" and "run/observe". Thus, we assume that the meta-variable with the largest range should precede the others in the variable ordering. Furthermore, the ordering of the boolean meta-variables is not as important as the ones with larger ranges, e.g. in the first two entries the boolean meta-variable "good" swaps the position without significant influence on the MTBDD size. Based on these findings and further tests, we recommend an ordering of the meta-variables according to their ranges in descending order.

Until now, we have focused only the meta-variable ordering within a single module. If the given program contains multiple modules, the metavariable ordering between these modules is also relevant. While it could be that an interleaved ordering between meta-variables of different modules reduces the size of the resulting MTBDD, the experimental results have proven that in general we should try to keep meta-variables that are simultaneously modified together. Because commands in Mod_i are only allowed to modify the meta-variables in Var_i , ordering the meta-variables module-wise is usually the best option. The following Figure 4.6 gives an intuition, why it makes sense to keep module variables close together. It has a tendency to allow better reduction of the identity MTBDDs of the other modules. Note that we unrealistically assume no dependencies between meta-variables of different modules for the sake of illustration.

Example 4.2 (Ordering Module Variables). Listing 4.1 shows an example probabilistic program with two modules Mod_1, Mod_2 . The first module Mod_1 contains the meta-variables $Var_1 = \{v_1, v_2\}$, whereas the second module Mod_2 contains $Var_2 = \{w_1, \dots, w_n\}$. For simplicity all meta-variables are boolean ones, i.e. the domain equals $[0, 1]$. Furthermore we assume only two commands in the module Mod_1 .

The Figure 4.6(a) shows the common module-wise ordering. This ordering allows the reduction of the identity of the other module Mod_2 . In contrast, Figure 4.6(b) shows an interleaved ordering, where v_1 is the first meta-variable and v_2 the last one. The other meta-variables w_1, \dots, w_n are in-between v_1 and v_2 . These meta-variables w_1, \dots, w_n remain unchanged in both commands of Mod_1 , but the identity can not be reduced.

Listing 4.1: A probabilistic program P_{ex3} .

```

mdp
module mod1
  v1: [0..1] init 0;
  v2: [0..1] init 0;

  [] true -> 1: (v1'=0) & (v2'=0);
  [] true -> 1: (v1'=1) & (v2'=1);
endmodule
module mod2
  w1: [0..1] init 0;
  ...
  wn: [0..1] init 0;
endmodule

```

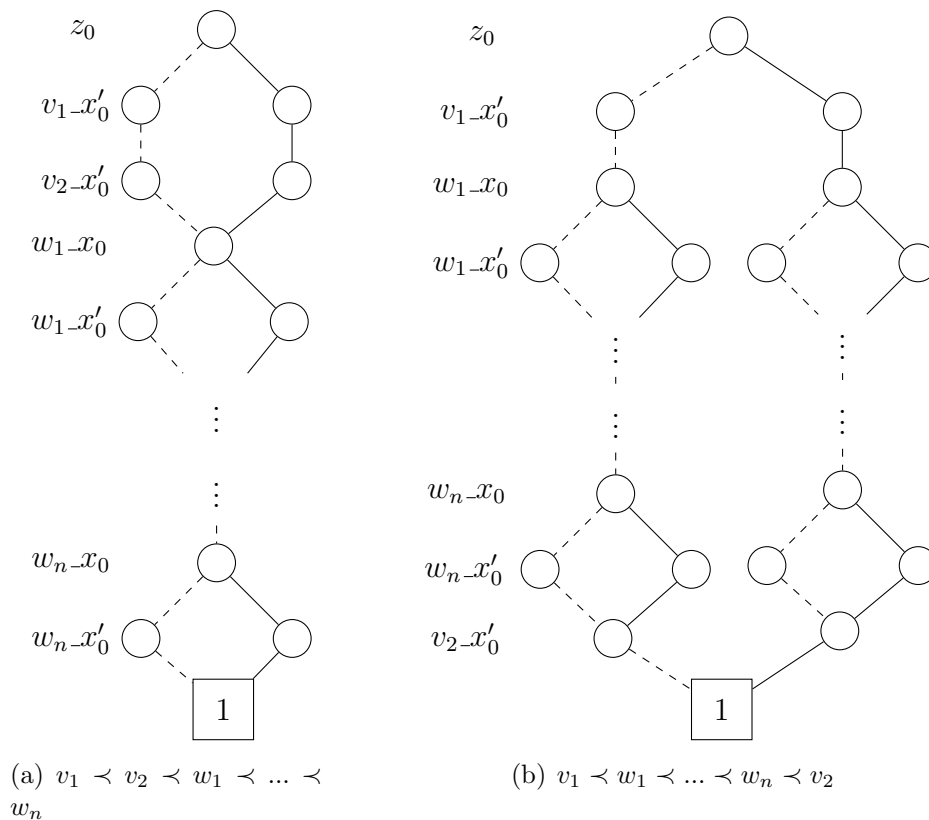


Figure 4.6.: Ordering module meta-variables.

Based on the previous tests, ordering the meta-variables according to their modules and domains seems to be a useful and simple approach. In particular, starting with meta-variables with a large range up to the ones with small range provides often an

4.3. Variable Ordering

adequate MTBDD size. The following Table 4.6 shows the experimental results of the new meta-variable ordering compared to default ordering for multiple modules. The default ordering is specified by the common ordering in the downloaded files. For some models, we also provide a worst-case size.

model	N	K	MTBDD			construction (s)	
			new	default	worst-case	new	default
synchronous	5	8	1526359	1540582	1673255	30.05	28.49
leader	5	9	3289838	3345130	–	76.01	72.03
crowds	15	5	17355	183680	280439	0.42	2.90
	20	5	28119	831299	1303783	0.72	81.52
asynchronous	8		387283	392068	520337	32.20	39.09
leader	9		854041	868229	–	126.31	146.95
brp	2500		2318	3952	–	11.94	19.53
firewire	100		329701	495062	–	38.99	43.65
csma	4	4	635859	553488	–	63.93	41.51
	4	6	3852012	3589198	–	737.22	457.47

Table 4.6.: Default variable order vs. new ordering.

Overall, the new ordering leads to a good MTBDD size. Furthermore, it sometimes improves the default ordering significantly. Nevertheless, a slightly reduced size of the MTBDD does not necessarily lead to a faster construction time (e.g. synchronous leader 5_8). For the CSMA models, the ordering leads to a minor increase in the MTBDD size, while the construction time increases drastically.

As discussed in the previous paragraphs, we should keep simultaneously modified meta-variables close-together. Usually only the meta-variables within a single module are modified by a command. But the synchronisation between different commands (via synchronising actions) also leads to a simultaneous modification of meta-variables of different modules. For example, the CSMA models contain a module "bus" that only contains commands with synchronising actions, which are used to synchronise between the other modules. Furthermore, a command might read the current value of all meta-variables. Thus, the guard or update expression potentially relies on a meta-variable of another module, so that we get further dependencies between all meta-variables. In general, we should try to keep the meta-variables with any dependency as close together as possible. But the presented new ordering of the meta-variables does not consider all these dependencies, which would require a more complex analysis.

We have proposed a new heuristic ordering, which sorts the given meta-variables according to their modules and ranges and typically achieves a very good MTBDD size. However, there are some models where this ordering performs significantly worse than the default one. The significant influence of the meta-variable ordering might justify further development of the presented heuristic. In the outlook we give first ideas about more precise analysis of the meta-variable ordering prior to the construction of the transition matrix and model checking process.

5. Conclusion

5.1. Summary

In this thesis we have provided the theoretical background for symbolic model checking (Chapter 2), where the given probabilistic program in PRISM language is represented by MTBDDs. In Chapter 3 we have described the construction of an MTBDD representing the transition matrix in detail. Furthermore, we have introduced the standard model checking algorithms for MDPs (and DTMCs), which are also implemented in the model checking tool STORM. Overall the implementation allows the automated verification of the specified requirements on the given system. In the evaluation (Chapter 4) we have first proven the competitiveness of our implementation compared to the well-known model checker PRISM. In the following we have tried to improve the reachability analysis by an algorithm called Chaining. But the experimental results have proven that the Chaining algorithm usually does not outperform the common BFS search. In the last section we have taken a closer look at the meta-variable ordering in the probabilistic program. Thereby, we have shown the significant influence of this ordering and have provided a first heuristic for ordering the meta-variables. This heuristic sorts all meta-variables module-wise according to their ranges in descending order. Different experimental results have proven the efficiency of this heuristic.

5.2. Future Work

As future work, we might consider the development of an improved heuristic or algorithm for the meta-variable ordering. This could include an extensive semantic analysis of all commands in the program. As possible representation, we could think of an undirected graph, where each node represents a certain meta-variable. An edge between two nodes represents a dependency, i.e. both meta-variables are modified simultaneously. In consequence, we could apply different graph algorithms to find a meta-variable ordering leading to a relatively small MTBDD size.

Bibliography

- [1] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Computer aided verification*, pages 585–591. Springer, 2011.
- [2] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press Cambridge, 2008.
- [3] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS approach: Correctness, modelling and performability of aerospace systems. In *Computer Safety, Reliability, and Security*, pages 173–186. Springer, 2009.
- [4] T. Sato and Y. Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the Fifteenth international joint conference on Artificial intelligence*, pages 1330–1335, 1997.
- [5] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.
- [6] D. E. Knuth and A. C. Yao. The complexity of nonuniform random number generation. *Algorithms and complexity: new directions and recent results*, pages 357–428, 1976.
- [7] E. Altman. Applications of markov decision processes in communication networks. In *Handbook of Markov decision processes*, pages 489–536. Springer, 2002.
- [8] B. K. Williams. Adaptive optimization and the harvest of biological populations. *Mathematical Biosciences*, 136(1):1–20, 1996.
- [9] K. L. McMillan. *Symbolic model checking*. Springer, 1993.
- [10] S. B. Akers. Binary decision diagrams. *Computers, IEEE Transactions on*, 100(6): 509–516, 1978.
- [11] D. Parker. *Implementation of symbolic model checking for probabilistic systems*. PhD thesis, University of Birmingham, 2002.
- [12] J.-P. Katoen, M. Z. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. Springer, 2001.

- [13] P. Buchholz, E. M. Hahn, H. Hermanns, and L. Zhang. Model checking algorithms for CTMDPs. In *Computer Aided Verification*, pages 225–242. Springer, 2011.
- [14] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing (preliminary report). In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 344–352. ACM, 1989.
- [15] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on*, pages 428–439. IEEE, 1990.
- [16] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains - modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.
- [17] M. Z. Kwiatkowska. Model checking for probability and time: from theory to practice. In *Logic in Computer Science, 2003. Proceedings. 18th Annual IEEE Symposium on*, pages 351–360. IEEE, 2003.
- [18] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.
- [19] C. Dehnert, J. Nils, W. Ralf, Á. Erika, and J.-P. Katoen. Fast debugging of PRISM models. In *Int. Symp. on Automated Technology for Verification and Analysis*. Springer, 2014.
- [20] M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.
- [21] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In *Design Automation, 1993. 30th Conference on*, pages 54–60. IEEE, 1993.
- [22] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.
- [23] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 1994.
- [24] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.
- [25] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.
- [26] M. Dufлот, L. Fribourg, T. Herault, R. Lassaigne, F. Magniette, S. Messika, S. Peyronnet, and C. Picaronny. Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC. *Electronic Notes in Theoretical Computer Science*, 128(6):195–214, 2005.

- [27] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461, 1990.
- [28] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 firewire root contention protocol. *Formal Aspects of Computing*, 14(3):295–318, 2003.
- [29] P. R. DArgenio, B. Jeannet, H. E. Jensen, and K. G. Larsen. Reachability analysis of probabilistic systems by successive refinements. In *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, pages 39–56. Springer, 2001.
- [30] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 38–41. IEEE Computer Society Press, 1993.

A. Appendix

Crowds Protocol

Listing A.1: A modified version of crowds20_5 in PRISM language.

```
dtmc

// probability of forwarding
const double PF = 0.8;
const double notPF = .2; // must be 1-PF
// probability that a crowd member is bad
const double badC = .167;
// probability that a crowd member is good
const double goodC = 0.833
// Total number of protocol runs to analyze
const int TotalRuns = 5;
// size of the crowd
const int CrowdSize = 20;

module crowds
  // protocol phase
  phase: [0..4] init 0;
  // crowd member good (or bad)
  good: bool init false;
  // the last seen crowd member
  lastSeen: [0..CrowdSize - 1] init 0;
  // number of protocol runs
  runCount: [0..TotalRuns] init 0;

  // observe_i is the number of times the attacker observed crowd member i
  observe0: [0..TotalRuns] init 0;
  observe1: [0..TotalRuns] init 0;
  observe2: [0..TotalRuns] init 0;
  observe3: [0..TotalRuns] init 0;
  observe4: [0..TotalRuns] init 0;
  observe5: [0..TotalRuns] init 0;
  observe6: [0..TotalRuns] init 0;
  observe7: [0..TotalRuns] init 0;
  observe8: [0..TotalRuns] init 0;
  observe9: [0..TotalRuns] init 0;
  observe10: [0..TotalRuns] init 0;
  observe11: [0..TotalRuns] init 0;
  observe12: [0..TotalRuns] init 0;
  observe13: [0..TotalRuns] init 0;
  observe14: [0..TotalRuns] init 0;
  observe15: [0..TotalRuns] init 0;
  observe16: [0..TotalRuns] init 0;
  observe17: [0..TotalRuns] init 0;
  observe18: [0..TotalRuns] init 0;
  observe19: [0..TotalRuns] init 0;

  // get the protocol started
  [] phase=0 & runCount<TotalRuns -> 1:(phase'=1) &
    (runCount'=runCount+1) & (lastSeen'=0);
```

```

// decide whether crowd member is good or bad according to given probabilities
[] phase=1 -> goodC : (phase'=2) & (good'=true) +
    badC : (phase'=2) & (good'=false);

// if the current member is a good member, update the last seen index
[] phase=2 & good -> 1/20 : (lastSeen'=0) & (phase'=3) +
    1/20 : (lastSeen'=1) & (phase'=3) + 1/20 : (lastSeen'=2) & (phase'=3) +
    1/20 : (lastSeen'=3) & (phase'=3) + 1/20 : (lastSeen'=4) & (phase'=3) +
    1/20 : (lastSeen'=5) & (phase'=3) + 1/20 : (lastSeen'=6) & (phase'=3) +
    1/20 : (lastSeen'=7) & (phase'=3) + 1/20 : (lastSeen'=8) & (phase'=3) +
    1/20 : (lastSeen'=9) & (phase'=3) + 1/20 : (lastSeen'=10) & (phase'=3) +
    1/20 : (lastSeen'=11) & (phase'=3) + 1/20 : (lastSeen'=12) & (phase'=3) +
    1/20 : (lastSeen'=13) & (phase'=3) + 1/20 : (lastSeen'=14) & (phase'=3) +
    1/20 : (lastSeen'=15) & (phase'=3) + 1/20 : (lastSeen'=16) & (phase'=3) +
    1/20 : (lastSeen'=17) & (phase'=3) + 1/20 : (lastSeen'=18) & (phase'=3) +
    1/20 : (lastSeen'=19) & (phase'=3);

// if the current member is a bad member, record the most recently seen index
[] phase=2 & !good & lastSeen=0 & observe0 < TotalRuns ->
    1:(observe0'=observe0+1) & (phase'=4);
[] phase=2 & !good & lastSeen=1 & observe1 < TotalRuns ->
    1:(observe1'=observe1+1) & (phase'=4);
[] phase=2 & !good & lastSeen=2 & observe2 < TotalRuns ->
    1:(observe2'=observe2+1) & (phase'=4);
[] phase=2 & !good & lastSeen=3 & observe3 < TotalRuns ->
    1:(observe3'=observe3+1) & (phase'=4);
[] phase=2 & !good & lastSeen=4 & observe4 < TotalRuns ->
    1:(observe4'=observe4+1) & (phase'=4);
[] phase=2 & !good & lastSeen=5 & observe5 < TotalRuns ->
    1:(observe5'=observe5+1) & (phase'=4);
[] phase=2 & !good & lastSeen=6 & observe6 < TotalRuns ->
    1:(observe6'=observe6+1) & (phase'=4);
[] phase=2 & !good & lastSeen=7 & observe7 < TotalRuns ->
    1:(observe7'=observe7+1) & (phase'=4);
[] phase=2 & !good & lastSeen=8 & observe8 < TotalRuns ->
    1:(observe8'=observe8+1) & (phase'=4);
[] phase=2 & !good & lastSeen=9 & observe9 < TotalRuns ->
    1:(observe9'=observe9+1) & (phase'=4);
[] phase=2 & !good & lastSeen=10 & observe10 < TotalRuns ->
    1:(observe10'=observe10+1) & (phase'=4);
[] phase=2 & !good & lastSeen=11 & observe11 < TotalRuns ->
    1:(observe11'=observe11+1) & (phase'=4);
[] phase=2 & !good & lastSeen=12 & observe12 < TotalRuns ->
    1:(observe12'=observe12+1) & (phase'=4);
[] phase=2 & !good & lastSeen=13 & observe13 < TotalRuns ->
    1:(observe13'=observe13+1) & (phase'=4);
[] phase=2 & !good & lastSeen=14 & observe14 < TotalRuns ->
    1:(observe14'=observe14+1) & (phase'=4);
[] phase=2 & !good & lastSeen=15 & observe15 < TotalRuns ->
    1:(observe15'=observe15+1) & (phase'=4);
[] phase=2 & !good & lastSeen=16 & observe16 < TotalRuns ->
    1:(observe16'=observe16+1) & (phase'=4);
[] phase=2 & !good & lastSeen=17 & observe17 < TotalRuns ->
    1:(observe17'=observe17+1) & (phase'=4);
[] phase=2 & !good & lastSeen=18 & observe18 < TotalRuns ->
    1:(observe18'=observe18+1) & (phase'=4);
[] phase=2 & !good & lastSeen=19 & observe19 < TotalRuns ->
    1:(observe19'=observe19+1) & (phase'=4);

// good crowd members forward with probability PF and deliver otherwise
[] phase=3 -> PF : (phase'=1) + notPF : (phase'=4);

// deliver the message and start over
[] phase=4 -> 1:(phase'=0);

endmodule

```