The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

EFFICIENT IMPLEMENTATION OF HYPERPCTL MODEL CHECKING

Hang Khuat

Examiners:
Univ.-Prof. Dr. Erika Ábrahám

Second Assessor:
apl. Prof. Dr. rer. nat. Thomas Noll

Aachen, July 29, 2020

Abstract

Security systems can be insecure and may expose sensitive information. To prevent insecurities such as the leakage of sensitive information, it is possible to analyse a security system. In the analysis, the system is abstracted to a model and a specification language is used to describe a property to be analysed. We use probabilistic system models to represent a system. The specification language to describe a property is a temporal logic called Probabilistic Computation Tree Logic (PCTL). With PCTL, we can only describe and observe a probability event of one execution in the system. However, in security systems, we are interested in the probabilistic relation between individual executions to observe similarities and differences. The property which allows us to draw probabilistic relations between multiple executions to observe similarities and differences is called *hyperproperty*. The term hyperproperty was introduced by Clarkson and Schneider [CS10]. Hyperproperty extends PCTL which results in *HyperPCTL*. In the paper from Ábrahám and Bonakdarpour [ÁB18], the temporal logic HyperPCTL was first introduced and a HyperPCTL model checking algorithm implemented. In this Master thesis, we improve the efficiency of the HyperPCTL model checking algorithm from [ÁB18] for a fragment of the HyperPCTL logic.

Contents

Chapter 1

Introduction

In this Master thesis, we increase the efficiency of a model checking algorithm for a fragment of the logic HyperPCTL. Ábrahám and Bonakdarpour [ÁB18] first introduced the temporal logic HyperPCTL and the original HyperPCTL model checking algorithm.

The original temporal logic behind HyperPCTL is the Computation Tree Logic (CTL) introduced by Clarke, Emerson and Sistla [CES86]. CTL is a branching time logic where the existential path quantifier (denoted as $\exists$) and the universal path quantifier (denoted as $\forall$) are allowed. The logic is evaluated over branching time structures such as trees. The temporal logic Probabilistic Computation Tree Logic (PCTL) is an extension of CTL introduced by Hansson and Johnsson [HJ94] where probability path quantification is allowed as well.

Those mentioned logics are deployed successfully to specify the (probabilistic) reachability of a system. With these logics, we can only describe a single individual execution at a time. However, our focus is different. In security systems, we are interested in the probabilistic relation between individual executions to observe similarities and differences. The temporal logics mentioned, CTL and PCTL, are not sufficient to observe multiple individual executions. A property which allows us to observe, compare and draw connections between independent executions is called *hyperproperty*. The term was introduced by Clarke and Schneider [CS10] and extended various temporal logics like Linear Temporal Logic (LTL), CTL, Computation Tree Logic* (CTL*) or PCTL which results in HyperLTL [FRS15], HyperCTL, HyperCTL* [FRS15] and HyperPCTL [ÁB18].

The HyperPCTL model checking algorithm takes a discrete-time Markov Chain (DTMC) $\mathcal{M}$ and a HyperPCTL formula ψ as input and checks if $\mathcal{M} \models \psi$ holds. A HyperPCTL formula allows us to start in different initial states. This is called multiple state quantification. As a result, we can observe different executions at a time and can draw connections and comparisons between them. Also, a HyperPCTL formula allows an arbitrary usage of the probabilities in arithmetic constraints. A HyperPCTL formula can be interpreted by a DTMC $\mathcal{M}$. A DTMC is a stochastic model where the next transition is selected according to a certain probability. Each

execution runs in the same DTMC. However, to observe and compare multiple executions we need copies of the model. The number of copies is related to the number of quantifiers n in the HyperPCTL formula. Building and combining them results in one model, which is called n-ary self-composition model $\mathcal{M}^n$. Using that model, the formula ψ is checked whether $\mathcal{M} \models \psi$ holds. If ψ has no quantifiers, then ψ only consists of constants and the formula can be directly evaluated on the model $\mathcal{M}$.

Our contribution to making the model checking algorithm more efficient is to reduce the arity of the self-composition model. For every sub-formula of ψ, we check the number of quantifiers in the scope of each sub-formula. Based on this number of quantifiers in a sub-formula, we build the corresponding self-composition model. Doing so, there exists not one n-ary self-composition model but several smaller ones. Having smaller models, the verification time is reduced in comparison to the original implementation, which has to run the verification on one large model. While more models need to be built, the smaller size can lead to a run-time improvement, especially for large formulae.

This Master thesis is structured as follows: Chapter 2 covers the preliminaries where the definition of hyperproperty, HyperPCTL (its syntax and semantic), an example of HyperPCTL and the HyperPCTL model checking algorithm is briefly explained. Chapter 3 provides an overview of related work on HyperPCTL. Chapter 4 describes the implementation environment of the HyperPCTL model checking algorithm and the pseudocode of the algorithm. Chapter 5 starts with our contributions to making the algorithm more efficient for a fragment of the HyperPCTL logic. Furthermore, the implementation of the original HyperPCTL model checking algorithm, as well as the implementation of our improvements, are illustrated. Additionally, we discuss challenges which occurred during the implementation as well as their solutions. The pseudocode of the improved implementation is depicted as well. Chapter 6 shows a brief analysis of the original HyperPCTL model checking algorithm compared to the improved one. We compare the number of states and transitions of the self-composition models as well as the run-time of both implementations. The last Chapter 7 concludes our Master thesis and provides an outlook on possible future work.

Chapter 2

Preliminaries

In this chapter, we explain the fundamental basics for this Master thesis. We cover the definitions of *hyperproperty, HyperPCTL* with its syntax and the semantics, and present an example. In the end, we explain the HyperPCTL model checking algorithm.

2.1 Basic Terms

HyperPCTL is an extension of Probabilistic Computation Tree Logic (PCTL) which is an extension of Computation Tree Logic (CTL).

- CTL: CTL is a temporal logic first introduced by Clarke, Emerson and Sistla [CES86]. It is a branching time logic which allows path quantifiers (denoted as $\exists, \forall$). The logic is interpreted over a branching time structure like tree-like structure.
- PCTL: PCTL is an extension of CTL and was introduced by Hansson and Johnsson [HJ94]. In PCTL, the probabilistic quantification over paths and a comparison of a probability to a certain threshold are allowed. Properties on states (state formulae) as well as on paths (path formulae) can be expressed on a model.

In this Master thesis, we deal with a probabilistic system modelled as a discrete-time Markov Chain (DTMC).

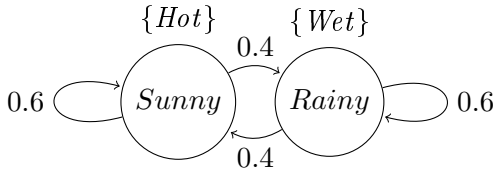
Definition 2.1.1. Definition DTMC
A DTMC is defined as a quadruple $\mathcal{M} = (S, P, AP, L)$ which contains:

- a finite non-empty set of states S
- a transition probability matrix $P : S \times S \to [0, 1]$
  By $P(s, s')$ we denote the probability where we move from s to s'. We require $\sum_{s' \in S} P(s, s') = 1$ for all states $s \in S$.

- a set of atomic propositions AP
- a labelling function $L : S \to 2^{AP}$
  It assigns to each state $s \in S$ the set $L(s)$ of atomic propositions $a \in AP$ which are valid in s.

A DTMC can be visualized as a graph. The states are represented as circles, the transitions with positive probability are depicted as arrows carrying the probability as a label and the state labels as a set next to a state. States which do not have a label are usually labelled with an empty set. For reasons of clarity and comprehensibility, the empty sets are omitted. Figure 2.1 shows a DTMC and its probability matrix P.

Example 2.1.1. Consider the following small DTMC example which represents a small weather report with its probability matrix.



$$P = \begin{pmatrix} & \textit{Sunny} & \textit{Rainy} \\ \textit{Sunny} & 0.6 & 0.4 \\ \textit{Rainy} & 0.4 & 0.6 \end{pmatrix}$$

Figure 2.1: Small DTMC example with its probability matrix P.

The columns and the rows of the probability matrix represent the states *Sunny* and *Rainy*.

For $s_1 = \textit{Sunny}$ and $s_2 = \textit{Rainy}$, the entry p_{ij} in the i-th row and j-th column of P stores the probability $P(s_i, s_j)$ for $i, j \in \{1,2\}$. A *Hot* sunny day is likely to be followed by another sunny day with a probability of 60% and a *Wet* rainy day is likely to be followed by another *Wet* rainy day with a probability of 60%. Since it is a stochastic matrix, the sum of the rows of P is 1.

An infinite sequence of states $\pi = s_0 s_1 \cdots \in S^\omega$ with $P(s_i, s_{i+1}) > 0$, for all $i \geq 0$, is called a *path* of a Markov chain $\mathcal{M} = (S, P, AP, L)$. We refer to the i-th state in the path π by $\pi[i]$.

$Paths^s(\mathcal{M})$ denotes the set of all infinite paths starting in the state s of $\mathcal{M}$. The set of all *finite paths* ($Paths^s_{fin}(\mathcal{M})$) denotes the set of all finite prefixes of paths from $Paths^s(\mathcal{M})$.

The probability space is defined by the smallest σ-algebra that contains the cylinder sets of all finite paths where the product of all transition probabilities gives the probability of the cylinder set of a finite path $\pi = s_0 s_1 \cdots s_n$: $Pr(cyl(\pi)) = \prod_{i=0}^{n-1} P(s_i, s_{i+1})$ [BK08].

2.2 A Hyperproperty

A *trace* is a sequence of observations along an execution. A set of traces is a *trace property* [Lam77]. A set of trace properties is called a *hyperproperty*. There are two basic classes of trace properties, namely safety properties and liveness properties. Temporal logics like Linear Temporal Logic (LTL) and CTL originated from these properties [AS85]. A trace property can only refer to a single execution at the same time. With a *hyperproperty* we can refer to multiple executions at a time. As a result, we can observe, compare and draw connections between the executions. The term hyperproperty was defined by Clarkson and Schneider [CS10].

2.3 Hyper Extension of PCTL – HyperPCTL

PCTL is an expansion of CTL and was introduced by Hansson and Johnsson [HJ94]. In CTL, we start in a given state where we verify whether a particular property holds for a single execution in a given model. In PCTL, as an extension to CTL, we start in a given state and quantify if a particular probabilistic property holds for a single execution in a given model. However, with PCTL (or CTL and LTL) we can only refer to a single execution at a time. Such an execution is a sequence of observations during an execution which is called a *trace*. A set of such traces is called a *trace property*. However, not all properties are verifiable over a single execution at a time, so the term *hyperproperty* is introduced. If we want to describe security policies for instance multilevel security policy, trace properties do not suffice.

In communication, security policies are essential. One security policy, like non-interference, is a system where its users are classified as low (not highly classified) or high (highly classified). A system's safety property is satisfied if any sequence of low input goes out as low outputs. Additionally, the low user should not notice any high inputs or outputs of high users during its usage of the system. Meaning the low user does not know the sensitive data usage of a high user. If we add probability to non-interference, we obtain probabilistic non-interference [III90]. The probability of a low observable trace is the same for every low-equivalent initial state [ÁB18]. The temporal logic, HyperPCTL, can express the requirements for probabilistic non-interference since we can address multiple paths at the same time. A HyperPCTL formula can be interpreted and modelled by a DTMC.

2.3.1 HyperPCTL Syntax

The HyperPCTL syntax has similarities and differences to other temporal logics.

A CTL state formula is a quantified formula over paths. If we add a probability operator to check if a particular property holds for the probability, we get a PCTL formula. In PCTL, we can compare the probabilities to a certain threshold. In HyperPCTL, an arbitrary usage of the probabilities in arithmetic constraint is allowed.

However, the main difference is that in CTL and PCTL we can describe single executions only whereas in HyperCTL* and HyperPCTL we can describe different concurrent executions and compare observations on them. This is realized in HyperCTL* through path quantification whereas in HyperPCTL state quantification is introduced as follows.

Definition 2.3.1. HyperPCTL Syntax
The following definition of the HyperPCTL syntax is from [ÁB18].
A HyperPCTL state formula ψ is defined as:

$$\psi ::= a_\sigma \mid \forall\sigma.\psi \mid \exists\sigma.\psi \mid (\psi \wedge \psi) \mid (\neg\psi) \mid p \sim p \mid \mathit{true}$$
$$p ::= c \mid \mathbb{P}(\varphi) \mid p + p \mid p - p \mid p \cdot p$$

where $\sim \in \{<, \leq, >, \geq, =\}$, $c \in \mathbb{Q}$, $a \in AP$ is an atomic proposition, σ is a state variable from a countably infinite supply of variables $V = \{\sigma_1, \sigma_2, \cdots\}$, p a probability expression and φ a path formula.
A HyperPCTL path formula φ is defined as:

$$\varphi ::= \bigcirc\psi \mid \psi \; \mathcal{U} \; \psi \mid \psi \; \mathcal{U}^{[k_1,k_2]} \; \psi$$

with ψ being a state formula and $k_1, k_2 \in \mathbb{N}_{\geq 0}$ where $k_1 \leq k_2$. The semantics of the operators $\bigcirc$ (Next), unbounded until $\mathcal{U}$ and bounded until $\mathcal{U}^{[k_1,k_2]}$ are the same as in the CTL defintion from [CES86].

The following syntactic sugar is used with $k_1, k_2 \in \mathbb{N}_{\geq 0}$ where $k_1 \leq k_2$:

- $\psi_1 \; \mathcal{U}^{\leq k} \; \psi_2 := \psi \; \mathcal{U}^{[0,k]} \psi$
- $\psi_1 \vee \psi_2 := \neg(\neg\psi_1 \wedge \neg\psi_2)$
- $\Diamond\psi := \text{true} \; \mathcal{U} \; \psi$
- $\Diamond^{[k_1,k_2]}\psi := \text{true} \; \mathcal{U}^{[k_1,k_2]}\psi$
- $\mathbb{P}(\Box\psi) := 1 - \mathbb{P}(\Diamond\neg\psi)$
- $\mathbb{P}(\Box^{[k_1,k_2]} \; \psi) := 1 - \mathbb{P}(\Diamond^{[k_1,k_2]} \; \neg\psi)$

We write $\mathcal{F}$, which denotes the set of all HyperPCTL state formulae. If an indexed atomic proposition a_σ occurs which is not in the scope of a quantifier bounding σ in ψ, this indexed atomic proposition is called *free*, otherwise it is *bound*. HyperPCTL *sentences* consist of HyperPCTL state formulae where all occurrences of an indexed atomic proposition are bound. HyperPCTL *(quantified) formulae* are HyperPCTL sentences.

Consider the following example of the HyperPCTL syntax.

Example 2.3.1. Syntax example:

$$\forall\sigma_1.\forall\sigma_2.\mathbb{P}(\Diamond a_{\sigma_1}) \sim \mathbb{P}(\Diamond b_{\sigma_2}), \text{where } \sim \in \{<, \leq, =, \geq, >\}$$

As seen in the formula, the atomic propositions a_{σ_1} and b_{σ_2} are in the scope of a quantifier's state variable σ_1 and σ_2. Thus, the two atomic propositions are *bound*. If we had an atomic proposition a_{σ_3} in this formula, it would have been called *free* since it is not bound to an occurring quantifier's state variable and this formula would not be a HyperPCTL formula.

Regarding the comprehension of this formula, it is to be understood as follows: We have two instantiated states s_1 and s_2 as σ_1 and σ_2, i.e., we have two paths, one starting in s_1 and the other one in s_2. We have two state labels a and b and a mathematical relation $\sim \in \{<, \leq, =, \geq, >\}$. Let us assume that the mathematical relation is $\leq$ without loss of generality. The formula $(\forall\sigma_1.\forall\sigma_2.\mathbb{P}(\Diamond a_{\sigma_1}) \leq \mathbb{P}(\Diamond b_{\sigma_2}))$ holds if and only if for each path the probability to eventually reach a state labelled with a from a path starting in s_1 (a_{σ_1}) is less or equal than the probability of reaching b from a path starting in s_2 (b_{σ_2}).

The purpose why we need indexed atomic proposition and what it means to be in the scope of a quantifier is explained in the following section.

2.3.2 HyperPCTL Semantics

The semantics is based on the n-ary self-composition of a DTMC. The executions for an instantiated state runs in one DTMC. To observe the executions of all instantiated states in one DTMC, we need copies of the DTMCs. Thus, we build and use the n-ary self-composition model DTMC $\mathcal{M}^n$ based on the number of n quantifiers in a HyperPCTL formula. Using this self-composition model, we can observe all executions at the same time.

Definition 2.3.2. HyperPCTL Semantic[ÁB18]
The n-ary self-composition of a DTMC $\mathcal{M} = (S, P, AP, L)$ results in a DTMC $\mathcal{M}^n = (S^n, P^n, AP^n, L^n)$ with the following components:

- $S^n = S \times \cdots \times S$ is an n-ary cartesian product of the set of states S.
- $P^n(s, s') = P(s_1, s'_1) \cdot \cdots \cdot P(s_n, s'_n)$ for all $s = (s_1, \cdots, s_n) \in S^n$ and $s' = (s'_1, \cdots, s'_n) \in S'$.
- $AP^n = \cup_{i=1}^n AP_i$, where $AP_i = \{a_i | a \in AP\}$ for $i \in \{1, \cdots, n\}$.
- $L^n(s) = \cup_{i=1}^n L_i(s_i)$ for all $s = (s_1, \cdots, s_n) \in S^n$ with $L_i(s_i) = \{a_i | a \in L(s_i)\}$ for $i \in \{1, \cdots, n\}$.

For a quantified HyperPCTL formula ψ which is satisfied by a DTMC $\mathcal{M} = (S, P, AP, L)$ we define the satisfaction relation as follows:

$$\mathcal{M} \models \psi \text{ iff } \mathcal{M}, () \models \psi$$

with () as the empty sequence of states. The satisfaction relation $\models$ defines the validity of the HyperPCTL state formulae, in the context of a DTMC $\mathcal{M}$ and n-tuple $s = (s_1, \cdots, s_n) \in S^n$ of states (for $n = 0$). Intuitively, $\mathcal{M} \models \psi$ means that the formula ψ is true for a sequence of states s in $\mathcal{M}$. In HyperPCTL we quantify over states. The probabilities are relevant for the context of states, i.e., it is crucial to know from which state s a path π starts as well as the probability of each transition in that path to evaluate a HyperPCTL formula ψ. To keep the reference between the quantifiers and the remaining formula we replace each σ with σ_i, and each corresponding a is replaced by $a_i, i \in \{1, \cdots, n\}$.

With that information, we can verify a formula like the one shown in Example 2.3.1.

The processing of a formula is called *structural recursion*. We evaluate a HyperPCTL formula ψ by starting to process it from left to right. If we evaluate Example 2.3.1, the formula would be evaluated as follows: We iterate the formula step by step and process the first part of the formula ψ which is the first instantiated quantifier with its state quantifier. The rest of the formula ψ is the sub-formula ψ' and is processed in the next iteration. We look at ψ' and start again from left to right. We take the first part of the sub-formula ψ' which is the second instantiated quantifier with its state quantifier. The rest of ψ' is the next sub-formula to be processed in structural recursion. This process is repeated until the whole formula is processed. The instantiated values for state variables are stored in the state sequence s.

The background on processing a formula using structural recursion is explained in the theoretical procedure of the HyperPCTL model checking algorithm in Section 2.4.

Definition 2.3.3. Semantics of DTMC formulae evaluation[ÁB18]
The formal notation of semantic rules to evaluate formulae in the context of a DTMC $\mathcal{M}(S, P, AP, L)$ and an n-tuple $s = (s_1, \cdots, s_n) \in S^n$ of states are as follows:

$$
\begin{array}{ll}
\mathcal{M}, s \models \text{true} & \\
\mathcal{M}, s \models \forall\sigma.\psi & \text{iff } \forall s_{n+1} \in S.\mathcal{M}, (s_0, \cdots, s_n, s_{n+1}) \models \psi[AP_{n+1}/AP_\sigma] \\
\mathcal{M}, s \models \exists\sigma.\psi & \text{iff } \exists s_{n+1} \in S.\mathcal{M}, (s_0, \cdots, s_n, s_{n+1}) \models \psi[AP_{n+1}/AP_\sigma] \\
\mathcal{M}, s \models a_i & \text{iff } a \in L(s_i) \\
\mathcal{M}, s \models \psi_1 \wedge \psi_2 & \text{iff } \mathcal{M}, s \models \psi_1 \text{and } \mathcal{M}, s \models \psi_2 \\
\mathcal{M}, s \models \neg\psi & \text{iff } \mathcal{M}, s \not\models \psi \\
\mathcal{M}, s \models p_1 \sim p_2 & \text{iff } [\![p_1]\!]_{\mathcal{M},s} \sim [\![p_2]\!]_{\mathcal{M},s} \\
[\![\mathbb{P}(\varphi)]\!]_{\mathcal{M},s} & = Pr\{\pi \in Paths^s(M^n) | \mathcal{M}, \pi \models \varphi\} \\
[\![c]\!]_{\mathcal{M},s} & = c \\
[\![p_1 + p_2]\!]_{\mathcal{M},s} & = [\![p_1]\!]_{\mathcal{M},s} + [\![p_2]\!]_{\mathcal{M},s} \\
[\![p_1 - p_2]\!]_{\mathcal{M},s} & = [\![p_1]\!]_{\mathcal{M},s} - [\![p_2]\!]_{\mathcal{M},s} \\
[\![p_1 \times p_2]\!]_{\mathcal{M},s} & = [\![p_1]\!]_{\mathcal{M},s} \cdot [\![p_2]\!]_{\mathcal{M},s}
\end{array}
$$

Here, ψ, ψ_1, ψ_2 are HyperPCTL state formulae, φ is a HyperPCTL path formula,

p_1 and p_2 are probability expressions, $\sim \in \{<, \leq, =, \geq, >\}$, $c \in \mathbb{Q}$ is a rational constant and $a \in AP$ an atomic proposition, $1 \leq i \leq n$. If there is an atomic proposition a_σ which is not in the scope of an instantiated quantifier, this atomic proposition is called *free*. The meaning of $\psi[AP_{n+1}/AP_\sigma]$ is that if a *free* atomic proposition a_σ occurs, each *free* occurrence of a_σ in ψ is replaced by a_{n+1} for each atomic proposition $a \in AP$.

The satisfaction relations for HyperPCTL path formulae, as defined in [ÁB18], are:

$$
\begin{array}{ll}
\mathcal{M}, \pi \models \bigcirc \psi & \text{iff } \mathcal{M}, \pi[1] \models \psi \\
\mathcal{M}, \pi \models \psi_1 \mathcal{U} \psi_2 & \text{iff } \exists j \geq 0.(\mathcal{M}, \pi[j] \models \psi_2 \wedge \forall i \in [0, j).\mathcal{M}, \pi[i] \models \psi_1) \\
\mathcal{M}, \pi \models \psi_1 \mathcal{U}^{[k_1, k_2]} \psi_2 & \text{iff } \exists j \in [k_1, k_2].(\mathcal{M}, \pi[j] \models \psi_2 \wedge \forall i \in [0, j).\mathcal{M}, \pi[i] \models \psi_1)
\end{array}
$$

with ψ, ψ_1, ψ_2 as HyperPCTL state formulae, $k_1, k_2 \in \mathbb{N}_{\geq 0}$ with $k_1 \leq k_2$ and π as a path of $\mathcal{M}^n$ for some $n \in \mathbb{N}_{>0}$.

Example 2.3.2. Semantics Example – 2 Dining philosophers
Consider the following DTMC $\mathcal{M}$ in Figure 2.2 where two philosophers are sitting at a table to have dinner together. In Figure 2.2, we start with the two philosophers without having any fork (s_0). The two forks are on the table and a philosopher always takes the right fork first. The probability that one, whether the first or the second philosopher, takes a fork is 0.5 (s_1 or s_3). If one philosopher has both forks, the philosopher starts to eat (s_2 or s_6). Once one philosopher has finished eating the probability that the forks are returned to the table is 1 ($s_6 \xrightarrow{1} s_7$, $s_2 \xrightarrow{1} s_5$). If one philosopher releases a fork, the right fork is always given first (s_5 or s_7). If both have one fork, then both philosophers are giving the forks back on the table (s_4).

To simplify this example only the probability of apparent states like s_0, where the probability of a philosopher taking one fork is 0.5, are deployed. Otherwise, the probabilities p and $1-p$ would have been chosen. In those transitions any probability for p is deployable.

To make it clear where left and right from a philosopher's perspective is, the left and right from its perspective are depicted in state s_0. A philosopher's perspective is omitted in the remaining states to keep it more explicit.

Consider $\mathcal{M}$ with the HyperPCTL formula ψ:

$$
\begin{aligned}
\forall \sigma_1 . \forall \sigma_2 . (I_1 \wedge I_2) \Rightarrow \mathbb{P}(\neg E_1^2 \mathcal{U} E_1^1) = \mathbb{P}(\neg E_2^2 \mathcal{U} E_2^1) \\
\wedge \mathbb{P}(\neg E_1^1 \mathcal{U} E_1^2) = \mathbb{P}(\neg E_2^1 \mathcal{U} E_2^2)
\end{aligned}
$$

This formula predicates that if for all pairs of initial states (labelled by the atomic proposition I), the probability $(\neg E_1^2 \mathcal{U} E_1^1)$ must be equal to $(\neg E_2^2 \mathcal{U} E_2^1)$ and the probability $(\neg E_1^1 \mathcal{U} E_1^2)$ must be equal to $(\neg E_2^1 \mathcal{U} E_2^2)$ for each $(s_i, s_j) \in S^2$ with $I \in L(s_i)$ and $I \in L(s_j)$. If that is the case, the formula is satisfied by $\mathcal{M}$ and it holds that $\mathcal{M}, (s_i, s_j) \models (\mathbb{P}(\neg E_1^2 \mathcal{U} E_1^1) = \mathbb{P}(\neg E_2^2 \mathcal{U} E_2^1)) \wedge (\mathbb{P}(\neg E_1^1 \mathcal{U} E_1^2) = \mathbb{P}(\neg E_2^1 \mathcal{U} E_2^2))$. In

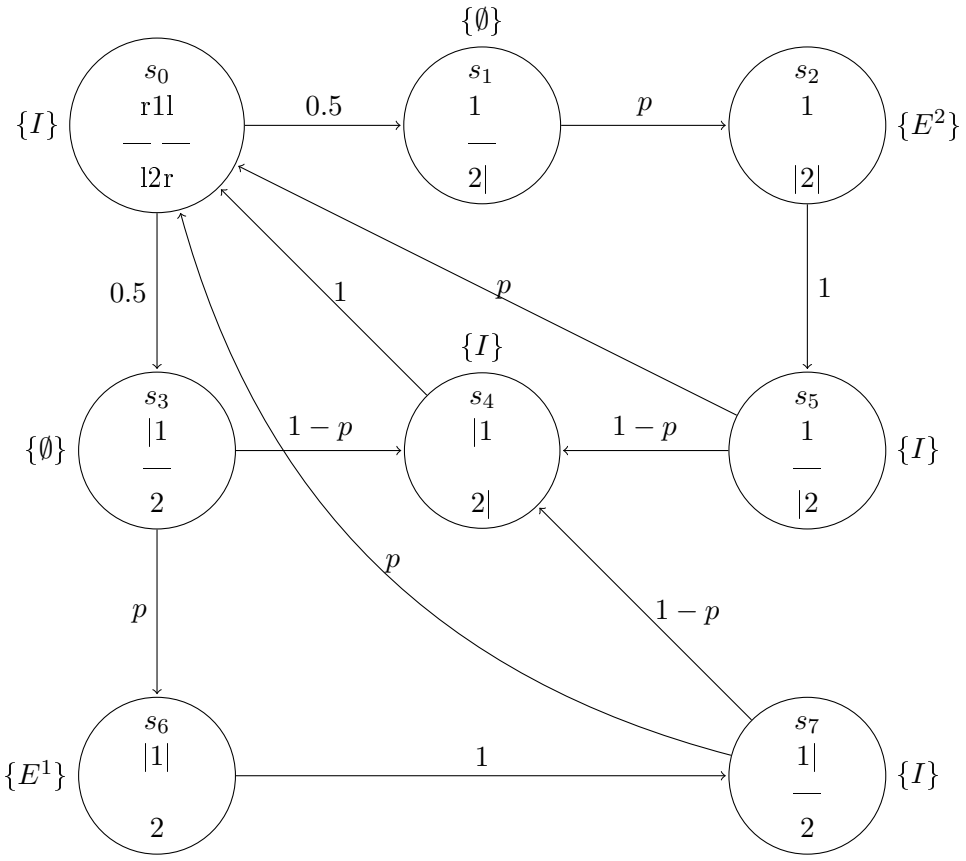


Figure 2.2: Two dining philosophers represented by DTMC $\mathcal{M}$.

other words, the probability of philosopher 2 (E^2) not eating until philosopher 1 (E^1) has begun eating, starting in one initial state (E_1^2) equals the probability of philosopher 1 (E^1) not eating until philosopher number 2 (E^2) has begun eating starting in another initial state (E_2^2) and vice versa.

By verifying the four initial states, we see that the probabilities are equal.

$$
\begin{aligned}
s_0 &= 0.5p + 0.5p && = p \\
s_4 &= 1 \cdot 0.5p + 1 \cdot 0.5p && = p \\
s_5 &= 0.5p^2 + 0.5p^2 + (1-p) \cdot 0.5p + (1-p) \cdot 0.5p \\
&= 1p^2 + 0.5p - 0.5p^2 + 0.5p - 0.5p^2 = 0.5p + 0.5p && = p \\
s_7 &= 0.5p^2 + 0.5p^2 + (1-p) \cdot 0.5p + (1-p) \cdot 0.5p \\
&= 1p^2 + 0.5p - 0.5p^2 + 0.5p - 0.5p^2 = 0.5p + 0.5p && = p
\end{aligned}
$$

Hence, for all pairs of initial states, the formula is satisfied by $\mathcal{M}$ ($\mathcal{M} \models \psi$).

2.4 HyperPCTL Model Checking Algorithm

Seyedehzahra Hosseini initially developed the implementation of the HyperPCTL model checking algorithm in the group of Borzoo Bonakdarpour, Iowa State University. The implementation is introduced in the paper of Ábrahám and Bonakdarpour, 2018 [ÁB18].

The procedure of the original implementation is initially explained in theory. The pseudocode of the algorithm is explained in detail in Chapter 4. The procedure works in four main steps which are introductory explained here.

We have a DTMC $\mathcal{M}^n$ and a HyperPCTL quantified formula ψ as input. The model checking algorithm verifies if a given model $\mathcal{M}^n$ satisfies the given property ψ. To check if a state s in $\mathcal{M}^n$ satisfies ψ, a set of labellings $\hat{L}^n(s)$ has to be recursively computed during the processing of ψ. In the set $L(s)$ we add atomic propositions whereas in the set $\hat{L}^n(s)$ we can also add sub-formulae. Each state s has a set $\hat{L}^n(s)$ that contains all atomic propositions $a \in AP$ and sub-formulae that are valid in s. While processing a formula step by step, the set of labellings $\hat{L}^n(s)$ is extended by the current atomic proposition or sub-formulae. Then, the algorithm evaluates if an arbitrary state is labelled with the correct set of labellings, $\psi \in \hat{L}^n(s)$, so that $\mathcal{M}^n$ satisfies ψ. Since each state has the same set of labellings $\hat{L}^n(s)$, it suffices to check if one arbitrary state s has $\psi \in \hat{L}^n(s)$. That means, either all states have the correct labelling, so that $\psi \in \hat{L}^n(s)$ holds, or none of them have $\psi \notin \hat{L}^n(s)$. If the labelling set has the correct labellings, the formula ψ is satisfied by $\mathcal{M}^n$, and the algorithm returns *true*, otherwise *false*. If ψ contains universal quantifiers, then all executions from the instantiated states s in $\mathcal{M}$ have to satisfy ψ. For an existential quantifier, only one execution from the instantiated states s in $\mathcal{M}$ has to satisfy ψ. A formula is processed by structural recursion, because during the recursive process of a formula, we build the set of labellings recursively. We evaluate all sub-formulae ψ' of ψ inside-out.

If we recall Example 2.3.1 and take equality as a mathematical relation, the algorithm evaluates the input as follows:

1. The quantifier state variables are renamed such that their names are $\sigma_1 \cdots \sigma_n$. The renaming is done to keep the context between the state variables in the formula.

2. Then the n-ary self-composition $\mathcal{M}^n$ is built based on the number of quantifiers n in ψ. In this case, a 2-ary self-composition $\mathcal{M}^2$ is built.

3. A labelling $\hat{L}^n(s)$ for all states $s \in S^n$ of $\mathcal{M}^n$ is recursively computed. Initially the set is empty, $\hat{L}^n(s) = \emptyset$ for all states $s \in S^n$. This process is implemented in the function `HyperPCTL` (see pseudocode in the Chapter 4). For all sub-formulae ψ' of ψ it does:

   (a) $\forall\sigma_1.\forall\sigma_2.\mathbb{P}(\diamond \underbrace{a_{\sigma_1}}_{\substack{\psi' = a_{\sigma_1}.\ \text{Add } \psi' \text{ to } \hat{L}^n(s).}}) = \mathbb{P}(\diamond \underbrace{b_{\sigma_2}}_{\substack{\psi' = b_{\sigma_2}.\ \text{Add } \psi' \text{ to } \hat{L}^n(s).}})$

(b) $\forall\sigma_1.\forall\sigma_2.$ $\underbrace{\mathbb{P}(\Diamond a_{\sigma_1})}_{p_1 = \mathbb{P}(\Diamond a_{\sigma_1})\text{. Compute values for } p_1.}$ $=$ $\underbrace{\mathbb{P}(\Diamond b_{\sigma_2})}_{p_2 = \mathbb{P}(\Diamond b_{\sigma_2})\text{. Compute values for } p_2.}$

(c) $\forall\sigma_1.\forall\sigma_2.$ $\underbrace{\mathbb{P}(\Diamond a_{\sigma_1}) = \mathbb{P}(\Diamond b_{\sigma_2})}_{\psi' = \mathbb{P}(\Diamond a_{\sigma_1}) = \mathbb{P}(\Diamond b_{\sigma_2})\text{. Check if } \psi' \text{ holds. If it holds, add } \psi' \text{ to } \hat{L}^n(s).}$

(d) $\forall\sigma_1.$ $\underbrace{\forall\sigma_2.\mathbb{P}(\Diamond a_{\sigma_1}) = \mathbb{P}(\Diamond b_{\sigma_2})}_{\psi' = \forall\sigma_2.\psi_1\text{. Add } \psi' \text{ to } \hat{L}^n(s) \text{ if } \psi_1 \text{ holds for all paths starting in } \sigma_2.}$

(e) $\underbrace{\forall\sigma_1.\forall\sigma_2.\mathbb{P}(\Diamond a_{\sigma_1}) = \mathbb{P}(\Diamond b_{\sigma_2})}_{\psi' = \forall\sigma_1.\psi_1\text{. Add } \psi' \text{ to } \hat{L}^n(s) \text{ if } \psi_1' \text{ holds for all paths starting in } \sigma_1.}$

4. At the end, the algorithm verifies for a state s in $\mathcal{M}^2$ if $\psi \in \hat{L}^n(s)$. If $\psi \in \hat{L}^n(s)$, then return *true* ($\mathcal{M}^2 \models \psi$), otherwise *false* ($\mathcal{M}^2 \not\models \psi$).

The function `ProbMC` proceeds with PCTL model checking, which was first introduced by Ciesinski and Größer [CG04]. The model checking algorithm for PCTL checks if a given model $\mathcal{M}^n$ satisfies the given property. It takes the model $\mathcal{M}^n$, the property p_n and the set of labellings $\hat{L}(s)$ as input from `HyperPCTL` so that `ProbMC` computes the values of the states s in p_i. The verification whether $\mathcal{M}^n \models \psi$ is done by `main` which comprises the fourth step.

The details of `ProbMC`, `HyperPCTL` and `main` are explained in Chapter 4 using pseudocode.

Chapter 3

Related Work

Non-probabilistic model checkers for the logics *HyperLTL* and *HyperCLT** are developed and discussed in [FRS15] and [CFST19]. The authors contribute to two aspects: First, the model checking algorithm for HyperLTL and HyperCTL* is based on alternating automata. Second, the first approach for model checking hardware systems for alternation-free HyperCTL* formulae is presented. Alternation-free means only one type of quantifier is allowed in the formula. A model checker MCHyper for HyperLTL is introduced in [CFST19]. Other tools for HyperLTL exists, for instance satisfiability solver EAHyper [FHS17] and MGHyper [FHH18]. Furthermore, there is a runtime monitoring tool RVHyper [FHST18].

For the probabilistic model checker, there exist three references for the temporal logics HyperPCTL and HyperPCTL*. As the temporal logic HyperPCTL has been first defined by Ábrahám and Bonakdarpour in 2018 [ÁB18], this field of research can be considered new. Consequently, there exist only three references yet which relate to HyperPCTL.

Probabilistic model checker for the logic HyperPCTL* is a statistical model checker (SMC) presented by Wang, Nalluri, Bonakdarpour and Pajic [WNBP19]. The SMC is not based on sequential probability ratio tests (SPRT) but Clopper-Pearson confidence intervals. This algorithm is defined for the non-nested grammar of HyperPCTL*.

A parameter synthesis problem for probabilistic hyperproperties is studied by Ábrahám, Bartocci, Bonakdarpour and Dobe [ÁBBD20]. The problem is discussed for a fragment of HyperPCTL.

A HyperPCTL model checking algorithm has been introduced by Ábrahám and Bonakdarpour [ÁB18]. The authors define the specification language *HyperPCTL*, for *probabilistic hyperproperties* of discrete-time Markov chains, and present a HyperPCTL model checking algorithm.

HyperPCTL allows us to express stochastic relations between multiple executions at the same time. Information-flow security policies like *non-interference* can be expressed with hyperproperties. Non-interference is a system where the users of a system are classified as low (not highly classified) or high (highly classified). A

system's safety property is satisfied if any sequence of low input goes out as low outputs. Additionally, the low user should not notice any high inputs or outputs of high users during its usage of the system. That means the low user does not know the sensitive data usage of a high user. If we add probability to non-interference, we obtain probabilistic non-interference [III90]. The probability of a low observable trace is the same for every low-equivalent initial state [ÁB18]. With Probabilistic Computation Tree Logic (PCTL), we can only express one execution. One execution represents one user. However, in this case we need to be able to express multiple executions to observe the users since the connection between them, whether one is a low or a high user, are relevant. With HyperPCTL, we can express multiple executions at the same time, so that we can observe the relation between the multiple executions.

The term *hyperproperty* was first defined in [CS10]. Hyperproperty can not only be applied to PCTL but to other temporal logics like Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Computation Tree Logic* (CTL*)*, PCTL and Probabilistic Computation Tree Logic* (PCTL*) resulting in their respective hyperproperty variants *HyperLTL* [FRS15], *HyperCTL, HyperCTL** [FRS15], *HyperPCTL* [ÁB18] and *HyperPCTL** [WNBP19].

In this Master thesis, we improve the efficiency for a fragment of HyperPCTL of the model checking algorithm of Ábrahám and Bonakdarpour. The system to be described by using a HyperPCTL quantified formula can be modelled as a discrete-time Markov Chain (DTMC). The algorithm takes a DTMC $\mathcal{M}$ and a HyperPCTL quantified formula ψ as input and checks if $\mathcal{M} \models \psi$ holds. The number of quantifiers n in ψ is essential for the algorithm. Based on n, the self-composition model $\mathcal{M}^n$ is built from $\mathcal{M}$ and with $\mathcal{M}^n$ the formula is verified. If there are no quantifiers in ψ, then the formula consists only of constants and the formula can be directly verified on $\mathcal{M}$. The pseudocode of the model checking algorithm is listed in Algorithm 1, Algorithm 2 and Algorithm 3 which are explained in Chapter 4.

In this Master thesis, we use PCTL model checking [BK08] and the model checker Storm [DJKV17, Sto20a] which we explain both in detail in Chapter 4.

Chapter 4

Referenced Implementation

The programming language of the original implementation is *Python* 3. The same programming language is used for the implementation of this Master thesis. In this chapter, we explain the details such as the software dependencies, the implementation environment and the pseudocode.

4.1 Software Dependencies

In this section, we shortly explain the used software.

4.1.1 Model Checker — Storm

Storm [DJKV17, Sto20a] is a tool developed at the chair of Software Modeling and Verification at RWTH Aachen University [IZw20] and programmed in the programming language C++. With Storm, we can evaluate random or probabilistic models. Storm takes a probabilistic model and a Probabilistic Computation Tree Logic (PCTL) formula as input and evaluates if the formula holds for the model. It supports several types of models, for instance, Markov chains and Markov decision processes (MDPs) with discrete as well as continuous time.

Storm uses different input languages to process a model and a formula. It takes models from different input languages like PRISM [KNP11], JANI [BDH+17] or GSPN [EHKZ13]. Each language represents different types of models as continuous-time Markov Chain (CTMC), Markov Decision Process (MDP), Petri net, Stochastic Petri net and some more. We have chosen to keep PRISM as the input language to ensure compatibility with the original implementation.

Storm requires one of the following operating systems: macOS 10.12 (and higher), DebianGNU/Linux 9 (and higher), Ubuntu Linux 16.10 (and higher). Additional dependencies are required for the compiler and general dependencies [Sto20b].

There are further optional dependencies for instance, *PyCarl* [PyC20], which allows us to represent rationales and rational functions in Storm. In our case, we are working with rationales and rational functions, so PyCarl is needed.

In order to be able to work with Storm, we need an API which allows us to use with Storm via Python. This API is called *Stormpy* [Sto20c].

4.1.2 API for Python and Storm — Stormpy and Dependency for Rationales and Rational Functions in Storm — PyCarl

Stormpy [Sto20c] is the API so that we can work with Storm using the programming language Python and a set of Python bindings is provided. However, with just these bindings, the input can not be evaluated yet since the formulae contain rationales and rational functions. Another binding necessary to work with arithmetic constraints is called PyCarl. Stormpy uses PyCarl internally.

Python **C**omputer **AR**ithmetic and **L**ogic library (PyCarl) [PyC20] is a set of Python bindings for the **C**omputer **AR**ithmetic and **L**ogic library (Carl). With this dependency, we can evaluate rationales and rational functions.

4.1.3 Input Language for Storm — Prism Language

The PRISM Language [Pri20, KNP11] is used to construct and analyse a model. This language is the input language for Storm to evaluate the property of a formula.

PRISM supports different types of models for instance, discrete-time Markov Chain (DTMC) CTMC etc. PRISM is used to construct and specify a DTMC.

4.1.4 Programming Language — Python

The implementation of the HyperPCTL model checking algorithm used in this thesis is written in the programming language *Python*. Seyedehzahra Hosseini implemented the code in the group of Borzoo Bonakdarpour, Iowa State University, USA.

The Python version has to be at least version 3 so that Stormpy can work. In this Master thesis, the Python version that has been used to evaluate the implementations is 3.6.9.

4.2 Implementation Environment

This section describes the environment of the implementation. The environments for the experiments and the necessary modules are listed. With this environment, the implementation of the HyperPCTL model checking algorithm can be compiled and executed.

For this Master thesis, the following instances were used:

- The implementation executions for this thesis were run on a ThinkPad E460 with a 2.3Ghz i5 processor and 12 GB RAM.
- The operating system used is Linux Ubuntu 18.04.

- The Python Version used is 3.6.9.

The following tools are necessary to compile the code:

- Storm [Sto20a] and the modules needed for its compilation can be read on the website [Sto20b].
- A Storm supporting operating system (in this case Ubunutu 18.04) and a `C/C++` compiler tool chain (in this case gcc).
- The API Stormpy [Sto20c].
- The dependency for evaluating rationales and rational functions PyCarl [PyC20].
- The Python Version 3.

4.3 Pseudocode

The most important functions in the implementation are `main`, `HyperPCTL`, `ProbMC` and `makeSelfComposition`. The pseudocode of the functions `main`, `HyperPCTL` and `ProbMC` are from the paper of Ábrahám and Bonakdarpour [ÁB18]. They are listed in Algorithm 1, Algorithm 2 and Algorithm 3 and explained in detail in this section.

Algorithm 1 Function `main`$(\mathcal{M}, \psi)$
HyperPCTL model checking algorithm [ÁB18].

Require: DTMC $\mathcal{M} = (S, P, AP, L)$, HyperPCTL quantified formula ψ
Ensure: Whether $\mathcal{M} \models \psi$
1: $n :=$ number of quantifiers in ψ
2: **if** $n = 0$ **then**
3: $\quad n := 1$ // n will be the arity of the self-composition
4: **end if**
5: let $\hat{L}^n : S^n \to 2^{\mathcal{F}}$ with $\hat{L}^n(s) = \emptyset$ for all $s \in S^n$
6: $\hat{L}^n :=$ HyperPCTL$(\mathcal{M}^n, \psi, n, \hat{L}^n)$ // (see Algorithm 2)
7: **if** $\psi \in \hat{L}^n(s)$ for some $s \in S^n$ **then**
8: $\quad$ **return** True
9: **else**
10: $\quad$ **return** False
11: **end if**

The function `main` in Algorithm 1 takes a DTMC model $\mathcal{M}$ and a HyperPCTL formula ψ as input and returns whether $\mathcal{M} \models \psi$ holds by evaluating the set of labellings $\hat{L}^n(s)$ for a state s in $\mathcal{M}$ (line 7–10). The set of labellings $\hat{L}^n(s)$ is computed by the function `HyperPCTL` (line 6). Before the set of labellings is created by the function `HyperPCTL`, the correct n-ary self-composition model has to be built so

that `main` can evaluate ψ. The number of quantifiers n in ψ determines the resulting n-ary self-composition model $\mathcal{M}^n$ which is built and returned by the function `makeSelfComposition` (lines 1–3). That model is passed to `HyperPCTL` and the function is able to create the set of labellings $\hat{L}^n(s)$ for all states $s \in S$ of the model $\mathcal{M}^n$.

The function `HyperPCTL` in Algorithm 2 is called by `main` and takes a discrete-time Markov Chain (DTMC) $\mathcal{M}^n$, a formula ψ, a non-negative integer n (the number of quantifiers in ψ) and a set of labellings $\hat{L}^n(s) = \emptyset$ as input. The function returns a set of labellings $\hat{L}^n(s)$. If the following cases occur in the sub-formulae ψ' of ψ, `HyperPCTL` does the following (inside-out):

- If the analysed sub-formula ψ' is *true*, then *true* is added to the set $\hat{L}^n(s)$ for each state $s \in S$ (lines 1–2).
- If the analysed sub-formula ψ' is an atomic proposition a_σ, then a_σ is added to the set $\hat{L}^n(s)$ for each state $s \in S$ (lines 3–4).
- If the analysed sub-formula ψ' has the form $\psi_1 \wedge \psi_2$, then we apply `HyperPCTL` to ψ_1 and ψ_2 separately and add ψ_1 and ψ_2 to the labelling set for each state $s \in S$ (lines 5–8).
- If the analysed sub-formula ψ' has the form $\neg\psi_1$, then $\neg\psi_1$ is added to $\hat{L}^n(s)$ for each state $s \in S$ with $\psi_1 \notin \hat{L}^n(s)$ (lines 9–11).
- If the analysed sub-formula ψ' has the form $p_1 \sim p_2$, then for all $\mathbb{P}(\varphi)$ appearing in $p_1 \sim p_2$ the function `ProbMC` is called which proceeds the standard PCTL model checking. The result of each probability-expression is returned from `ProbMC` to `HyperPCTL`. Then `HyperPCTL` checks if $p_1 \sim p_2$ holds. If it does, the labelling $p_1 \sim p_2$ is added to the labelling set $\hat{L}^n(s)$ for each state $s \in S$ (lines 12–15).
- If the analysed sub-formula ψ' has the form $\exists\sigma_i.\psi_1$, then `HyperPCTL` calls itself again to process the next sub-formula ψ_1 recursively. All states s are labelled with $\exists\sigma_i.\psi_1$ if and only if there exists a state $s'_i \in S$ in the labelling set $\hat{L}^n(s_1, \cdots, s_{i-1}, s'_i, s_{i+1}, \cdots, s_n)$ so that the sub-formula holds (lines 16–18).
- If the analysed sub-formula ψ' has the form $\forall\sigma_i.\psi_1$, then `HyperPCTL` calls itself again recursively to process the next sub-formula ψ_1. All states s are labelled with $\forall\sigma_i.\psi_1$ if and only if for all $s'_i \in S$ are in the labelling set $\hat{L}^n(s_1, \cdots, s_{i-1}, s'_i, s_{i+1}, \cdots, s_n)$ so that the sub-formula holds (lines 19-21).

The function `ProbMC` in Algorithm 3 is called by `HyperPCTL` and takes the DTMC $\mathcal{M}^n$, the probability expression p, the non-negative integer n (the number of quantifiers in ψ) and the set of labellings $\hat{L}^n(s)$ from `HyperPCTL` as input. Since `ProbMC` proceeds the standard PCTL model checking, the algorithm needs to compute a set $\hat{L}^n(s)$ recursively for all states s to verify if each state s is in $\hat{L}^n(s)$ to

check if $\mathcal{M} \models p$ holds or not. However, `ProbMC` does not verify if $[\![p]\!]_{\mathcal{M},s}$ but returns the values of p in all states $s \in S^n$ as $L_p^n(s)$. The values $L_p^n(s)$ are returned to `HyperPCTL` (line 30) so that `HyperPCTL` can compare the results of `ProbMC`. If the comparison holds, then the labellings of the states are added to the set $\hat{L}^n(s)$ in `HyperPCTL` (lines 12–15 in Algorithm 2).

There, `ProbMC` computes the values $L_p^n(s)$ of p which are zero in the beginning (line 1) for all states $s \in S$ as illustrated below:

- If p has the form $c, c \in \mathcal{Q}$, then the value c is set, $L_p^n(s) = \{c\}$ for all states $s \in S$ (lines 2–3).
- If p has the form p_1 *op* p_2 with *op* $\in \{+,-,\cdot\}$, then each sub-expression p_i, in this case p_1 and p_2 is called separately with `ProbMC` again to compute the values L_1^n, L_2^n for each sub-expression p_1, p_2 (lines 4–6).
- If p has the form $\mathbb{P}(\varphi)$, there can be several cases, namely:
  - If φ has the form $\bigcirc\psi$, then the function computes a matrix multiplication to obtain the value for the next operator $\bigcirc$ (lines 9–11).
  - If φ has the form $\psi_1 \mathcal{U} \psi_2$, we apply `HyperPCTL` separately on ψ_1 and ψ_2 first (lines 12–14) and extend the set of labellings as explained in Algorithm 2. With $\varphi = \psi_1 \mathcal{U} \psi_2$, we have to fulfil the until relation. To verify the until relation, we compute a linear equation system as the following:
    - For all states $s \in S$ the probability is 0 if neither ψ_1 nor ψ_2 are in the set of labellings $\hat{L}^n(s)$ (from the function `HyperPCTL`) or if no state s' where $\psi_2 \in \hat{L}^n(s')$ is reachable from state s. If there is no state s' with $\psi_2 \in \hat{L}^n(s')$ reachable from s where $s \in \hat{L}^n(s)$ then ψ_2 will not be in the labelling set $\hat{L}^n(s)$ and the until relation is not satisfied. with $\psi_2 \in \hat{L}^n(s')$ is reachable from state s (line 16).
    - For all states $s \in S$, the probability is 1 if ψ_2 is in the set of labellings $\hat{L}^n(s)$ passed from the function `HyperPCTL` (line 17).
    - For all other states, the probability is $\sum_{s' \in S^n} P^n(s,s') \cdot p_{s'}$. It means, we add all values up where we go from state s to some state s' multiplied with the probability $p_{s'}$ which satisfies the property in s' (line 8).
  - If φ has the form $\varphi = \psi_1 \mathcal{U}^{[k_1,k_2]} \psi_2$, we apply `HyperPCTL` separately on ψ_1 and ψ_2 first (lines 12–14) and then extend the set of labellings as explained in Algorithm 2. With $\varphi = \psi_1 \mathcal{U}^{[k_1,k_2]} \psi_2$, we have to satisfy the until relation $\mathcal{U}$ and additionally fulfil the property within an interval from k_1 to k_2. Thus, we can not proceed as in the case $\varphi = \psi_1 \mathcal{U} \psi_2$ but we have to go iteratively from $i = 1$ to k_2 to check the probability that fulfils the property in i steps (lines 24–26). At the end, the probabilities in the interval k_1 to k_2 are summed up (line 27).

Algorithm 2 Function `HyperPCTL`$(\mathcal{M}^n, \psi, n, \hat{L}^n)$
HyperPCTL model checking algorithm [ÁB18].

Require: DTMC $\mathcal{M}^n = (S^n, P^n, AP^n, L^n)$, HyperPCTL quantified formula ψ, non-negative integer n, $\hat{L}^n : S^n \to 2^{\mathcal{F}}$
Ensure: An extension of $\hat{L}^n$ to label each state $s \in S$ with sub-formulae of ψ that hold in s

if $\psi = \text{true}$ **then**
  for all $s \in S^n$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\text{true}\}$
else if $\psi = a_{\sigma_i}$ **then**
  for all $s \in S^n$ with $a_i \in L^n(s)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{a_{\sigma_i}\}$
else if $\psi = \psi_1 \wedge \psi_2$ **then**
  $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
  $\hat{L}^n :=$ HyperPCTL$(\mathcal{M}, \psi_2, n, \hat{L}^n)$
  for all states $s \in S^n$ with $\{\psi_1, \psi_2\} \subseteq \hat{L}^n(s)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\psi\}$
else if $\psi = \neg\psi_1$ **then**
  $\hat{L}^n :=$ HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
  for all states $s \in S^n$ with $\psi_1 \notin \hat{L}^n(s)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\psi\}$
else if $\psi = p_1 \sim p_2$ **then**
  $L_1^n :=$ ProbMC$(\mathcal{M}, p_1, n, \hat{L}^n)$ // (see Algorithm 3)
  $L_2^n :=$ ProbMC$(\mathcal{M}, p_2, n, \hat{L}^n)$ // (see Algorithm 3)
  for all states $s \in S^n$ with $L_1^n(s) \sim L_2^n(s)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\psi\}$
else if $\psi = \exists\sigma_i.\psi_1$ **then**
  $\hat{L}^n :=$ HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
  for all states $s = (s_1, \cdots, s_n) \in S^n$ with $\psi_1 \in \hat{L}^n(s')$ for some $s_i' \in S$ and $s' = (s_1, \cdots, s_{i-1}, s_i', s_{i+1}, \cdots, s_n)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\psi\}$
else if $\psi = \forall\sigma_i.\psi_1$ **then**
  $\hat{L}^n :=$ HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
  for all states $s = (s_1, \cdots, s_n) \in S^n$ with $\psi_1 \in \hat{L}^n(s')$ for all $s_i' \in S$ and $s' = (s_1, \cdots, s_{i-1}, s_i', s_{i+1}, \cdots, s_n)$ set $\hat{L}^n(s) := \hat{L}^n(s) \cup \{\psi\}$
end if
return $\hat{L}^n$

Algorithm 3 Function `ProbMC`$(\mathcal{M}^n, \psi, n, \hat{L}^n)$
HyperPCTL model checking algorithm [ÁB18].

Require: DTMC $\mathcal{M} = (S, P, AP, L)$, HyperPCTL probability expression p, non-negative integer n, $\hat{L}^n : S^n \to 2^{\mathcal{F}}$
Ensure: $L_p^n : S^n \to \mathbb{Q}$ specifying values $L_p^n(s)$ of p in all states $s \in S^n$

Let $L_p^n : S^n \to \mathbb{Q}$ with $L_p^n(s) = 0$ for all $s \in S$
if $p = c$ **then**
  for all $s \in S^n$ set $L_p^n(s) = c$
else if $p = p_1 \ op \ p_2$ with $op \in \{+, -, \cdot\}$ **then**
  $L_1^n :=$ ProbMC$(\mathcal{M}, p_1, n, \hat{L}^n)$
  $L_2^n :=$ ProbMC$(\mathcal{M}, p_2, n, \hat{L}^n)$
  for each $s \in S^n$ set $L_p^n(s) := L_1^n(s) \ op \ L_2^n(s)$
else if $p = \mathbb{P}(\varphi)$ **then**
  if $\varphi = \bigcirc\psi$ **then**
    $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi, n, \hat{L}^n)$
    for all $s \in S^n$ set $L_p^n(s) = \sum_{s' \in S^n, \psi \in L^n(s')} P^n(s, s')$
  else if $\varphi = \psi_1 \mathcal{U} \psi_2$ **then**
    $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
    $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi_2, n, \hat{L}^n)$
    compute unique solution v for the following equation system:
    (1) $p_s = 0$ for all states $s \in S^n$ with $\psi_1 \notin \hat{L}^n(s)$ and $\psi_2 \notin \hat{L}^n(s)$, or if no state s' with $\psi_2 \in \hat{L}^n(s')$ is reachable from s
    (2) $p_s = 1$ for all states $s \in S^n$ with $\psi_2 \in \hat{L}^n(s)$
    (3) $p_s = \sum_{s' \in S} P^n(s,s') \cdot p_{s'}$ for all other states
    for all $s \in S^n$ set $L_p^n(s) = v(p_s)$
  else if $\varphi = \psi_1 \mathcal{U}^{[k_1,k_2]} \psi_2$ **then**
    $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi_1, n, \hat{L}^n)$
    $\hat{L}^n :=$HyperPCTL$(\mathcal{M}, \psi_2, n, \hat{L}^n)$
    for each $s \in S^n$ set $P_0^n(s) = 1$ if $\psi_2 \in \hat{L}^n(s)$ and $P_0^n(s) = 0$ otherwise
    for $i = 1$ to k_2 **do**
      for each $s \in S^n$ set $P_i^n(s) = \sum_{s' \in S} P^n(s,s') \cdot P_{i-1}^n(s')$ if $\psi_1 \in \hat{L}^n(s)$ and $P_i^n(s) = 0$ otherwise
    end for
    for all $s \in S^n$ set $L_p^n(s) = \sum_{i=k_1}^{k_2} P_i^n(s)$
  end if
end if
return L_p^n

Chapter 5

Implementation of the Algorithm

This chapter covers the implementation of the HyperPCTL model checking algorithm. We explain our contribution to this thesis first. Then, the improvements and the execution of the improved implementation are explained. During the execution, we have faced challenges which are explained along with several solutions.

5.1 Our Contributions

We contribute two aspects which are explained more detailed in the following:

1. Exploiting the probability matrix.
2. Reducing the arity of the self-composition model.

Storm returns the probability values by computing the probability vector for a requested state. If we want to know the probability values of a different requested state, Storm is recalled again to compute the probability vector for that state. However, only one computation with Storm suffices since Storm computes the values for all states and only returns the values for a particular state. Instead, we should save Storm's computation for all states directly and store the computations for further requests. This is one of the two contributions we intended to add. However, while inspecting the implementation we found out that this improvement had already been made in the original implementation. The details of this improvement are explained shortly in Section 5.2.1.

The second contribution is reducing the arity of the self-composition model. As we see in the function `main` in Algorithm 1 the n-ary self-composition is computed and passed on to the function `HyperPCTL` in which the formula is processed. The improvement does not build the self-composition model before performing `HyperPCTL` but during processing the formula in our new function `getBoundedVariables`. The formula is split in sub-formulae and based on the number of dependent quantifiers in a sub-formula, we build the according self-composition model. Since a sub-formula is usually not dependent on all quantifiers, the self-composition model

is smaller than the n-ary self-composition model. Thus, we do not need to build the n-ary self-composition model in the beginning. The n-ary self-composition model is typically bigger than the sub-formula self-composition model. Thus, the original implementation has to evaluate more transitions and states for a sub-formula in the n-ary self-composition model, which is more time-consuming. We reduce the number of states and transitions and therefore the evaluation time by building only the necessary size of self-composition model.

More detailed aspects of the implementation and its challenges will be described in the next sections.

5.2 What Improvements are Envisaged

In this section, we explain our contributions in more detail. We first describe our idea how to exploit the probability matrix to save some computation steps. Afterwards, we describe our idea to make the HyperPCTL model checking algorithm more efficient by reducing the arity of the self-composition model.

5.2.1 First Improvement: Exploiting the Probability Matrix

Recalling Example 2.3.1, the formula is satisfied by a discrete-time Markov Chain (DTMC) if, for each instantiated state s_1, s_2, the probability to eventually reach a state labelled with a from s_1 is according to the mathematical relation $\sim \in \{<, \leq, =, \geq, >\}$ compared to the probability of reaching b from s_2.

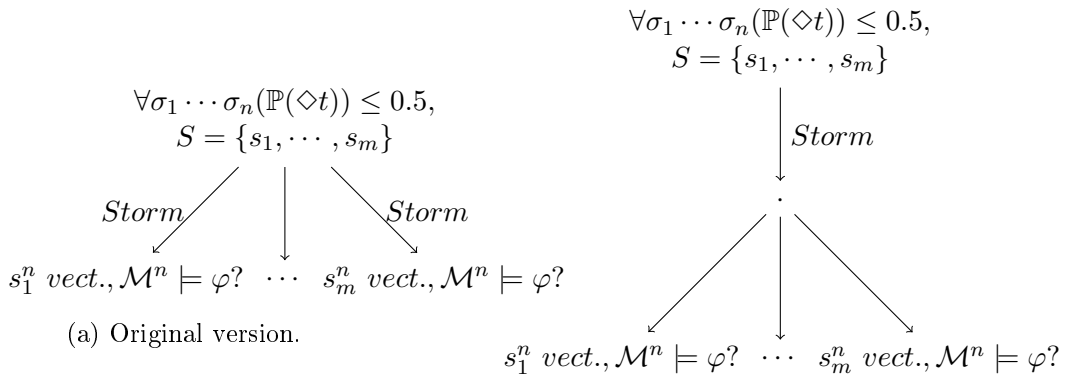
The original implementation computes a probability vector for every requested state via Storm. In Example 2.3.1, if we want to know the probability values of σ_1 and σ_2, the implementation computes two probability vectors. The computation of a probability vector computes the values for all states and returns the value for a requested state, in this case, σ_1. For the second probability vector, the implementation would compute the values for all states again and returns the value for σ_2. So in the improvement, it suffices to have one computation for all states instead of computations of every state, as shown in Figure 5.1.

Figure 5.1a is an illustration of the original process of Storm. In Figure 5.1b, the desired process of Storm is illustrated. While revising the implementation to improve it, as shown in Figure 5.1b, we discovered that the improvement had already been made. The function is called `check_property` which will be explained in Section 5.3.

5.2.2 Second Improvement: n-ary Self-Composition

In this implementation, we aim to reduce the arity of the self-composition model to make the model checking algorithm more efficient.

From Algorithm 1, we know that the arity of the self-composition model depends on the number of quantifiers n in ψ. However, not all states and transitions of the resulting self-composition model are relevant for a sub-formula ψ' during the process



(a) Original version.

(b) Desired version.

Figure 5.1: Storm call of the original and the desired version.

of ψ. To reduce the number of states and transitions, we have to reduce the arity of the self-composition model.

A vital restriction regarding this improvement is the HyperPCTL syntax: The original implementation allows nested formulae. In our improvement, we only allow non-nested formulae so that we can build smaller models for every sub-formula and reduce the arity of the self-composition models. The HyperPCTL syntax is the following:

Definition 5.2.1. Restricted HyperPCTL Syntax[ÁBBD20]
HyperPCTL state formula:

$$\psi ::= a_\sigma \mid \forall\sigma.\psi \mid \exists\sigma.\psi \mid (\psi \wedge \psi) \mid (\neg\psi) \mid p \sim p \mid true$$
$$p ::= c \mid \mathbb{P}(\bigcirc\varphi) \mid \mathbb{P}(\varphi\ \mathcal{U}\ \varphi) \mid p + p \mid p - p \mid p \cdot p$$

where $\sim \in \{<, \leq, >, \geq, =\}$, $c \in \mathbb{Q}$, $a \in AP$, σ a state variable from a countably infinite supply of variables $V = \{\sigma_1, \sigma_2, \cdots\}$, p a probability expression and φ a path formula.
HyperPCTL path formula:

$$\varphi ::= a_\sigma \mid \varphi \wedge \varphi \mid \neg\varphi \mid true$$

The syntactic sugar and the HyperPCTL semantics remain unchanged.

For the improvement of the n-ary self-composition we consider the following type of formulae:

$$\mathcal{Q}\sigma_1.\cdots\mathcal{Q}\sigma_n.((\bigwedge_{i=1}^{n} AP_{\sigma_i}) \Rightarrow (\bigwedge_{j=1}^{m} p_j \sim_j p_{j'}))$$

where $\mathcal{Q} \in \{\exists, \forall\}, \sim \in \{<, \leq, >, \geq, =\}, c \in \mathbb{Q}$, AP_{σ_i}: Boolean combination of atomic propositions $\in AP$; might also be *true* and n,$m \in \mathbb{N}_{\geq 0}$.

For the improvement of the n-ary self-composition, the number of state variables σ in p_j and $p_{j'}$ are relevant.

We name the parts of the formula as described below to simplify the explanations and figures.

$$\underbrace{\mathcal{Q}\sigma_1.\cdots\mathcal{Q}\sigma_n.}_{quantifiers}\underbrace{((\bigwedge_{i=1}^{n} AP_{\sigma_i})}_{initial\text{-}states} \Rightarrow \underbrace{(\bigwedge_{j=1}^{m} \underbrace{p_j \sim_j p_{j'}))}_{probability\text{-}formula_j}}_{probability\text{-}formula}$$

5.2.3 A First Attempt

The original implementation builds the n-ary self-composition model first and evaluates the formula via `HyperPCTL` with the resulting self-composition model as input model (see Algorithm 2 and Algorithm 3). However, it is not always necessary to build the n-ary self-composition model to evaluate the formula. Notably, we take a look at the sub-formulae $probability\text{-}formula_m$ first, more precisely $p_j, p_{j'}$, before building the self-composition model. The sub-expression p_j depends on some state variables and for this sub-expression we do not need the n-ary self-composition model to evaluate p_j but the self-composition model based on the dependent state variables in p_j. A sub-expression usually does not depend on all state variables in ψ. Thus, building the n-ary self-composition model to evaluate a sub-expression p_j would be too "large". In the n-ary self-composition model, there are some states and transitions which are not relevant for the evaluation of p_j. By building a self-composition model that is smaller than the n-ary self-composition model, we save states, transitions and computation time. The same procedure holds for the other sub-expression $p_{j'}$. If there are several $probability\text{-}formula_m$ the whole process is repeated for every $probability\text{-}formula_j$.

Example 5.2.1. Consider the examples:

(i) $\underbrace{\forall\sigma_1.\forall\sigma_2.(\text{init}_{\sigma_1} \wedge \text{init}_{\sigma_2})}_{\text{Add the labellings of state quantifiers and inital states with HyperPCTL with the original model instead of 2-ary self-composition.}} \Rightarrow \underbrace{(\underbrace{\mathbb{P}(\Diamond a_{\sigma_1})}_{p_1\text{. Compute 1-ary composition and the probabilities for all init states.}} = \underbrace{\mathbb{P}(\Diamond a_{\sigma_2})}_{p_2\text{. Compute 1-ary composition and the probabilities for all init states.}})}_{\text{Apply probMC with 1-ary composition on } p_1 \text{ and } p_2\text{. Check if } p_1 = p_2.}$

(ii) $\underbrace{\forall\sigma_1.\forall\sigma_2.\forall\sigma_3.(b_{\sigma_1} \wedge b_{\sigma_2} \wedge b_{\sigma_3})}_{\text{Add the labellings of state quantifiers and initial states with HyperPCTL with the original model instead of 3-ary self-composition.}} \Rightarrow \underbrace{(\underbrace{\mathbb{P}(\Diamond(a_{\sigma_1} \wedge a_{\sigma_2}))}_{p_1\text{. Compute 2-ary composition and the probabilities for all b states.}} < \underbrace{\mathbb{P}(\Diamond(a_{\sigma_2} \wedge a_{\sigma_3}))}_{p_2\text{. Compute 2-ary composition and the probabilities for all b states.}})}_{\text{Apply probMC with 2-ary composition on } p_1 \text{ and } p_2. \text{ Check if } p_1 < p_2.}$

The first step of the procedure stays the same as in the original implementation in Section 2.4.

The second step is to split the formula ψ into sub-expressions. To reduce the n-ary self-composition only the *probability-formula* on the right of "$\Rightarrow$" is relevant. In the sub-expression $p_1 = \mathbb{P}(\Diamond a_{\sigma_1})$, we see that the atomic proposition a is dependent on σ_1, so only the 1-ary self-composition has to be used instead of the 2-ary self-composition to evaluate the first sub-expression. The same holds for the other sub-expression $p_2 = \mathbb{P}(\Diamond a_{\sigma_2})$ accordingly. After those two sub-expressions have been evaluated, the two results are compared to the mathematical relation, in this case the equality relation. If the result is *true* we have to evaluate for each state satisfying $init_{\sigma_1}$ and $init_{\sigma_2}$ whether they satisfy the sub-formula. In contrast to the original implementation, we do not need the 2-ary self-composition model. There, we would have built a 2-ary self-composition model in the beginning and would have evaluated the whole formula on the 2-ary self-composition model by breaking down the formula by structural recursion (see Algorithm 2).

Informally, the implementation for the first attempt can be described as follows:

- Apply variable renaming such that the quantified state variables are named $\sigma_1 \cdots \sigma_n$.
- Split the formula ψ into two parts: quantifiers and the rest of the formula. We also do the labelling of ψ as in the original code. The labelling of the quantifiers is done by applying `HyperPCTL` to the quantifiers. The rest of the formula is an implication with the conjunction of the initial states as the premise and the probability-formula as the conclusion. If the initial state does not exist, we return *true*, otherwise we proceed. The formula is split into the sub-expressions $p_j, p_{j'}$.
- We count how many state variables σ are in the sub-expression p_j and build the self-composition model according to the number of state variables which appear as label index in the sub-expression p_j.
- The same holds for $p_{j'}$. We count how many state variables σ are in the sub-formula $p_{j'}$ and build the self-composition model according to the number of state variables which appear as label index in the sub-expression $p_{j'}$.

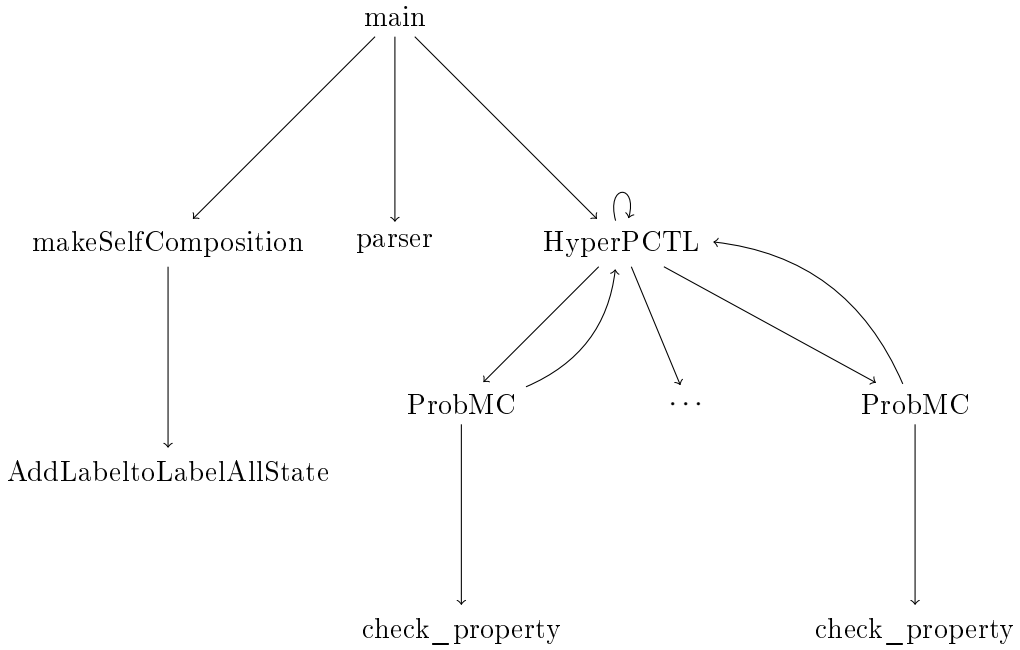


Figure 5.2: Procedure call hierarchy of the original implementation.

- We evaluate each sub-expression by performing model checking with `ProbMC` on p_j, $p_{j'}$ and compare the two results according to the mathematical relation for each state in the self-composition model.
- If the result is *true*, then the whole formula is satisfied, otherwise it is unsatisfied.
- If the formula has several probability-formulae ($probability\text{-}formula_1, \cdots,$ $probability\text{-}formula_m$) connected by conjunctions, then we evaluate each probability-formula one by one as explained in the previous steps.

5.3 Implementation of the Algorithm

During the implementation of the improved HyperPCTL model checking algorithm some challenges occurred. Their solutions are explained, and the resulting modified functions are listed with their parameters and variables of the improved HyperPCTL model checking algorithm. At the end of this chapter, the pseudocode of the improved HyperPCTL model checking algorithm is shown.

As introduction to the original implementation, a procedure call hierarchy is depicted in Figure 5.2.

In the original implementation, the function `main` gets as input a DTMC model $\mathcal{M}$, a HyperPCTL formula ψ and n as arity for the self-composition model. The

function `makeSelfComposition` is called by `main` where it passes the DTMC model $\mathcal{M}$ and the arity n for the self-composition model to that function. The function `makeSelfComposition` builds the self-composition model $\mathcal{M}^n$ and returns this model. It uses a helper function called `AddLabeltoLabelAllState` which computes the new resulting labellings during the building process of the new self-composition model. After the self-composition model is returned, the `main` function parses the HyperPCTL formula ψ with the function `parser`. Then, the function `HyperPCTL` is called where it gets the self-composition model $\mathcal{M}^n$ and the parsed function ψ as input. The function `HyperPCTL` processes ψ. Depending on ψ, the function `HyperPCTL` calls itself recursively as explained in the description of Algorithm 2. In Figure 5.2, the recursion is depicted as the self-loop on `HyperPCTL`. This function has a helper function called `ProbMC` which performs a Probabilistic Computation Tree Logic (PCTL) model checking. Its input is the self-composition model $\mathcal{M}^n$ and the probability-expression p from the function `HyperPCTL`. As explained in Algorithm 3, the function `ProbMC` calls the function `HyperPCTL` depending on the probability-expression. This is depicted as the loop between `HyperPCTL` and `ProbMC` in Figure 5.2. The function `ProbMC` returns the values of the given probability-expression p to `HyperPCTL`. The values of the probability-expression are computed with a helper function called `check_property` which computes the probability vector of the probability-expression via Storm and returns the values to `ProbMC`. Then, `ProbMC` returns the values to the function `HyperPCTL`. The function `HyperPCTL` returns the set of labellings $\hat{L}^n(s)$ for the model $\mathcal{M}^n$ after processing ψ. At the end, the function `main` takes the result of `HyperPCTL` and checks for an arbitrary state s if it has the correct labelling. If it has the correct labellings, `main` returns *true* and $\mathcal{M} \models \psi$ holds, otherwise the result is *false* and $\mathcal{M} \not\models \psi$.

In the improved implementation, the function `main` only calls the new function `getBoundedVariables`. The input parameters for the function `main` remains the same, the DTMC model $\mathcal{M}$, the arity n of the self-composition model and the HyperPCTL formula ψ. The function `getBoundedVariables` gets the formula ψ as input. It parses the formula ψ by calling the function `parser` and split the formula into quantifiers, initial states and probability-formula. If the initial states do not exist in the DTMC model, the function returns *true*. If the initial states do exist, then the probability-formula has to be evaluated. The probability-formula is split into the probability-expressions p_j and $p_{j'}$. For every probability-expression, the according self-composition model is built by calling the function `makeSelfComposition`. This function takes the model $\mathcal{M}$ and the number of state variables in a probability-expression for the arity n for the self-composition model as input. The process of the function `makeSelfComposition` and its helper function `AddLabeltoLabelAllState` remains the same as before. After the self-composition model is built for each probability-expression, the function `ProbMC` is called with the probability-expression and the self-composition model as input. The process of the function `ProbMC` and its help function `check_property` remains the same as before. After the value of each probability-expression is returned, we eval-

uate if the probability-expressions hold to the according to mathematical relation. If the relation holds, the function returns *true*, otherwise *false*. If the formula has multiple probability-formulae, this process is repeated for every probability-formula.

The following section explains the technical implementation of the original HyperPCTL model checking algorithm.

5.3.1 List of Functions and Variables of the original Implementation

We first explain the functions with the most critical parameters and the most essential variables of the original implementation in brief, so that the changes in the new implementation can be explained in comparison.

- `main(model, formula, arity_SC)`:
  The three parameters of the function `main` are the DTMC model $\mathcal{M}$, a HyperPCTL formula ψ and the arity n for the self-composition model. Intuitively, we assumed that the implementation would determine the arity by the number of quantifiers in ψ. However, the arity of the self-composition model is a parameter entered by the user as a command line argument.

  In the improved implementation, the arity of the self-composition model is not relevant any more because it is determined in `getBoundedVariables`. The parameter remains because the implementation needs this input for some other functions which are not relevant for the scope of this Master thesis.

- `makeSelfComposition(model, ···, arity_SC)`:
  This function returns the self-composition model $\mathcal{M}^n$. The most important parameters of this function are the `model` $\mathcal{M}$ and the arity `arity_SC` for the self-composition model. The function `main` passes the model and the arity to this function. During the building process of the self-composition model, `makeSelfComposition` uses a helper function `AddLabeltoLabelAllState` to determine the new labellings of the resulting self-composition model.

- `AddLabeltoLabelAllState(model, ···, arity_SC)`:
  This function is called from `makeSelfComposition` and returns the labellings for the new self-composition model.

- `HyperPCTL(formula, model, labellingSet, ···)`:
  After the function `makeSelfComposition` returns the self-composition model $\mathcal{M}^n$, `main` passes the parsed `formula` and the self-composition model `model` to `HyperPCTL` to create and return the set of labellings `labellingSet` $\hat{L}^n(s)$. In the beginning the set of labellings is empty. During the process of the formula the function `ProbMC` is called, where it computes and returns the values of the probability-expression p. To add the labellings of the probability-expressions to the labelling set, `HyperPCTL` has to check if the probability-

expressions hold for the mathematical relation. If it does, then the labellings are added to the labelling set and proceed with processing the formula.

- `ProbMC(probability-expression, model, labellingSet, ···)`:
  This function takes the `probability-expression` p, the `model` $\mathcal{M}^n$ and the `labellingSet` $\hat{L}^n(s)$ as input from the function `HyperPCTL`. Classic PCTL model checking is performed in `ProbMC` and returned to `HyperPCTL`. To determine the values, `ProbMC` has a helper function `check_property`.

- `check_property(probability-expression, model)`:
  This function determines the values for the probability-expression by computing the probability vector via Storm. The result is returned to `ProbMC`.

- Global variables:

  - List `AllState`:
    This list is used in the function `makeSelfComposition`. During the building process of the new self-composition model the function computes the new resulting states for the self-composition model. Those states are contained in this list.
  - List `VisitedState`:
    This list is used in the function `makeSelfComposition`. After a state from `AllState` is processed where the transitions and its labellings are created for that state, it is stored in this list. `makeSelfComposition` proceeds building the self-composition model if there are states in the list `AllState` which are not in `VisitedState`. Meaning, there are some states left which have not been processed at that time. Otherwise, the function terminates the building process and all states have been visited and handled.
  - Set `LabelAllState`:
    This set contains all labels of the processed model. If one recalls the model of the example in Section 2.2, the set contains the following labels for that model: $LabelAllState = \{q_0 : \{I\}, q_2 : \{E^2\}, q_4 : \{I\}, q_5 : \{I\}, q_6 : \{E^1\}, q_7 : \{I\}\}$.

The next section explains the implementation challenges and solutions. It is also explained why and how the methods were modified. The modified functions with their parameters and variables are listed below.

5.3.2 Implementation Challenges and their Solutions to Reduce the n-ary Self-composition

In order to reduce the arity as explained in Section 5.2.3 only non-nested formulae are allowed. However, the original procedure allows nested formulae. So the first step of the challenge is to change the nested HyperPCTL syntax to a non-nested

HyperPCTL syntax. Before the HyperPCTL syntax is changed from nested to non-nested, it is necessary to think about the process of the formulae since the process has changed as well.

Initially, a formula is processed by structural recursion. In the improvement, structural recursion is not applied on the whole formula ψ, but only at the beginning until the right arrow "⇒" of ψ. Up to the implication the formula is split into sub-expressions $p_1, \cdots p_m$ to evaluate each sub-expression.

Two examples from the paper of Ábrahám and Bonakdarpour [ÁB18] are applied to test the implementation. The first example features the HyperPCTL formula on the DTMC $\mathcal{M}$ as shown in Figure 5.3.

$$\forall\sigma.\forall\sigma'.(init_\sigma \wedge init_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond a_\sigma) = \mathbb{P}(\Diamond a_{\sigma'})) \tag{5.1}$$

This function is satisfied by the DTMC $\mathcal{M}$ if the probability to reach a is the same for all pairs of the initial states $init_\sigma$ and $init_{\sigma'}$.

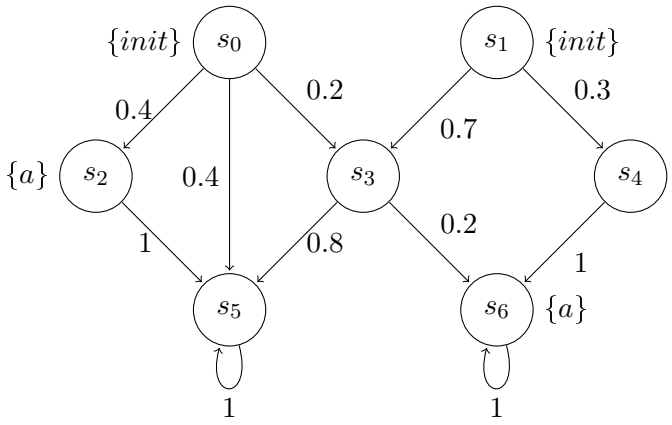


Figure 5.3: DTMC $\mathcal{M}$ of the first example tested in the implementation [ÁB18] to test formula 5.1.

The second example features the HyperPCTL formula on the DTMC $\mathcal{M}'$ as shown in Figure 5.4.

$$\begin{aligned}\forall\sigma.\forall\sigma'.[((t=n)_\sigma \wedge (t=y)_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond(r=n)_\sigma) \leq e^{ln3} \cdot \mathbb{P}(\Diamond(r=n)_{\sigma'}))]\wedge \\ [((t=y)_\sigma \wedge (t=n)_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond(r=y)_\sigma) \leq e^{ln3} \cdot \mathbb{P}(\Diamond(r=y)_{\sigma'}))]\end{aligned} \tag{5.2}$$

This example represents the randomized response protocol to ensure that some information about a user cannot be traced back to that user. This protocol is often used in a survey where the privacy of a user is maintained. Some users do not want to reveal their answers, so the users answer the questions of a survey incorrectly, or they do not answer them at all. Through randomizing a user's response, the privacy of a user can be ensured.

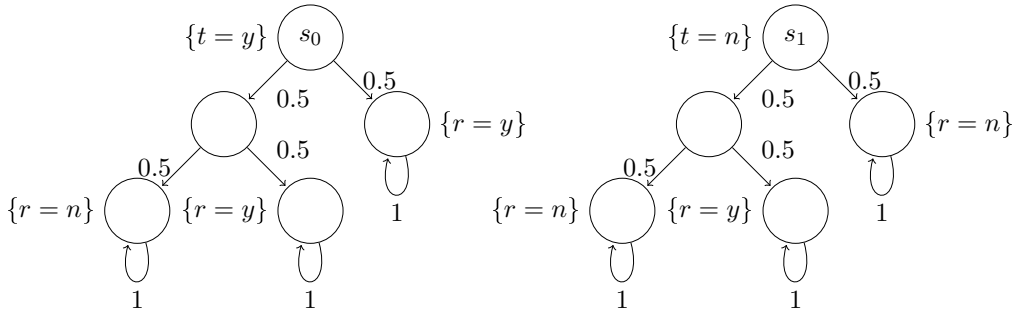


Figure 5.4: DTMC $\mathcal{M}'$ of the randomized response protocol to test formula 5.2 to test the second example in the implementation [ÁB18].

5.3.3 1. Challenge: How to process the Formula

The formula is processed consecutively as depicted in Section 2.4. Since the goal of the improvement is to reduce the arity of the self-composition, the first step of processing the formula is to look into the sub-expressions $p_j, p_{j'}$ first as shown in Section 5.2.3. According to the number of dependent state variables, the corresponding self-composition models of p_j and $p_{j'}$ are built. Then, we use PCTL model checking for all sub-expression, and we check if the results hold regarding the mathematical relation.

The original process of the formula in the implementation is depicted in Figure 5.5. The syntax is processed recursively as in structural recursion. Thus, the procedure favours depth-first approaches, as shown in Figure 5.5. In the improvement, processing the formula has changed since the sub-expressions p_j and $p_{j'}$ have to be inspected first. The restricted syntax is depicted in Figure 5.6.

5.3.4 2. Challenge: Implementation of the HyperPCTL Syntax

The HyperPCTL syntax in the original implementation was implemented as in Definition 2.3.1. Thus, nested functions are allowed. In the improvement of the implementation, no nested functions are allowed to reduce the arity of self-composition. The formula ψ is arbitrarily extendable by conjunction. In the implementation the HyperPCTL syntax is restricted as explained in Section 5.2.1.

5.3.5 3. Challenge: Applying the Indices during Self-composition

As explained in Section 5.2.1, we look at the number of the dependent state variables before the according self-composition model is built for the first sub-expression. Based on the number of state variables z in the sub-expression, the z-ary self-composition model is built and evaluated with this self-composition model. The same process holds for the second sub-expression. If the numbers of dependent state variables are the same in both sub-expressions as in the Example 5.2.1, the first intuition was to use the same self-composition model for the second sub-expression. In this

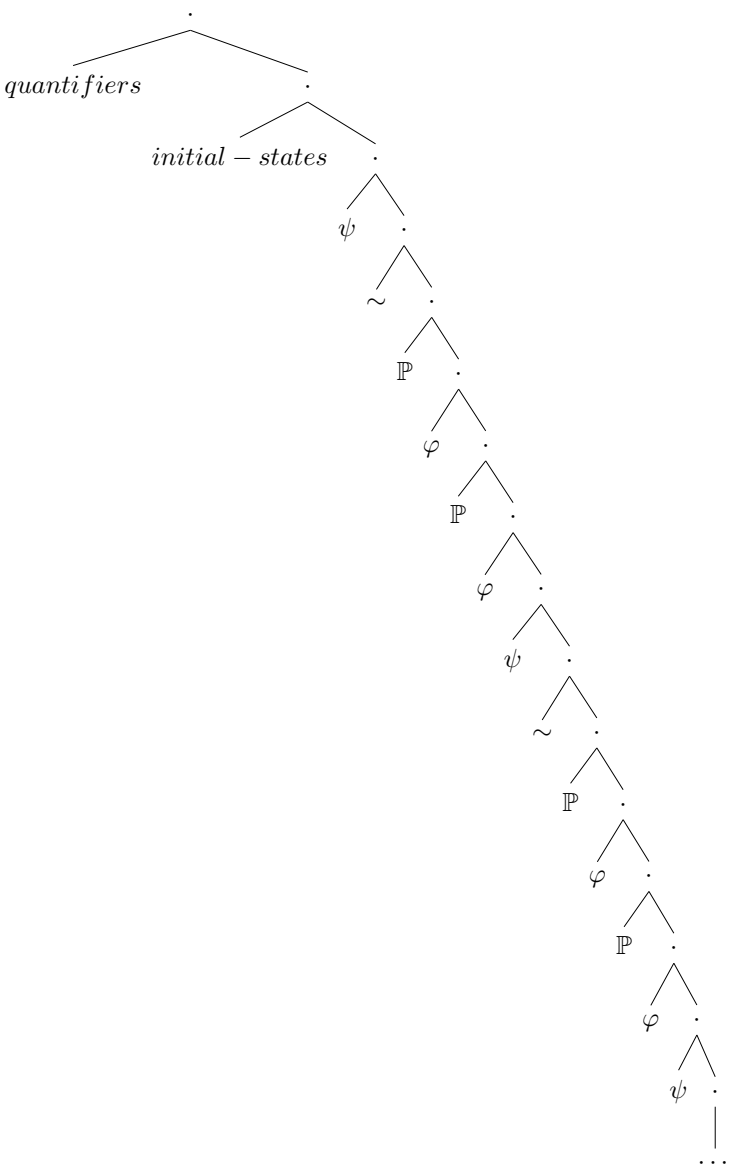


Figure 5.5: Processing tree of parsing a formula in the original implementation.

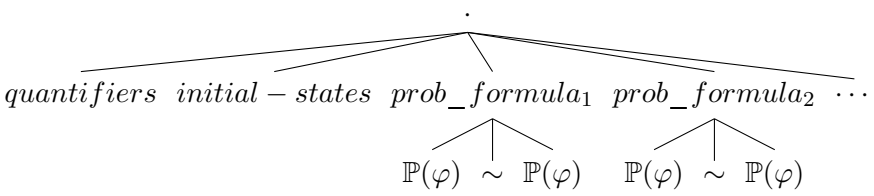


Figure 5.6: The new attempt of processing the formulae.

case, another self-composition computation could have been avoided. However, this did not work since the self-composition is also dependent on the indices i of the state variables. The implementation matches the labels of the self-composition model; i.e., the indices i of the model need to match with the indices i of the sub-expression in order to evaluate it. An error occurs with the message that the implementation could not find a labelling for σ_2 as in Example 5.2.1 since the self-composition model from the first sub-formula is labelled with σ_1. In the original implementation depicted in Algorithm 2, the n-ary self-composition is computed and then forwarded to `HyperPCTL`. The function `HyperPCTL` uses the self-composition model to evaluate the formula. The error that some labelling could not be found did not occur in the original implementation since the formula was not split into sub-expressions and all indices are contained in the self-composition model.

To solve this challenge, the function `makeSelfComposition` had to be adapted. The indices of a sub-expression have to be passed on to that function. Taking Example 5.2.1, the index passed to the function for the first sub-expression p_j is 1. For the second sub-expression $p_{j'}$, the index passed to the function is 2. Thus, a new parameter `indices` is added to `makeSelfComposition` and passed on to `AddLabeltoLabelAllState`. With the added information of the indices of the sub-expressions as a parameter for `makeSelfComposition`, the correct labelling can be computed and matched to the resulting model.

However, the implementation still did not compute the correct self-composition model. The execution aborted during computing the self-composition model for the second sub-formula with index 2. The global variable list `AllState` plays a key role in this function. After finishing the self-composition model for the first sub-expression with index 1, the function is recalled for the second sub-expression with index 2. Since `AllState` is a global list, it still preserves all the states from the previous operation. This function compares the `VisitedStates` with `AllState` to know which states are left to process. It seems that this function has already visited all states and aborts the building process. The global set `LabelAllState` is not used in this function any more since the global set also contains the labelling from the previous operations.

Consequently, new *local* variables have to be introduced:

- `AllState`
- `TempLabelAllState`

To preserve all the labellings in the implementation the set `TempLabelAllState` is updated with the global set `LabelAllState` after every self-composition procedure.

5.3.6 4. Challenge: Evaluating Multiple Formulae

After adapting the implementation of the HyperPCTL model checking algorithm according to the three challenges, we can evaluate ψ which has one probability-

formula. To evaluate various formulae, the example shown in Figure 5.4 was tested.

If multiple formulae are to be evaluated then the formula looks like the following:

$$\underbrace{\underbrace{\mathcal{Q}\sigma_1.\cdots\mathcal{Q}\sigma_n.}_{quantifiers}(\bigwedge_{i=1}^{n}(\underbrace{\bigwedge_{j=1}^{m}AP_{\sigma_j}}_{initial\text{-}states})\Rightarrow(\underbrace{\bigwedge_{k=1}^{l}\underbrace{p_k\sim_k p_{k'}}_{probability\text{-}formula_k}}_{probability\text{-}formulae}))}_{imply\_op}}_{imply\_ops}$$

The repeating parts of formula ψ, referred to as *imply_op*, are the *initial-states* and the *probability-formulae*. So according to the formula processing in Figure 5.6, evaluating several formulae is complicated. A better approach would be if the processing of formula is implemented as suggested in Figure 5.7. With this strategy, the number of probability-formulae is checked and processed accordingly as explained in Section 5.2.3.

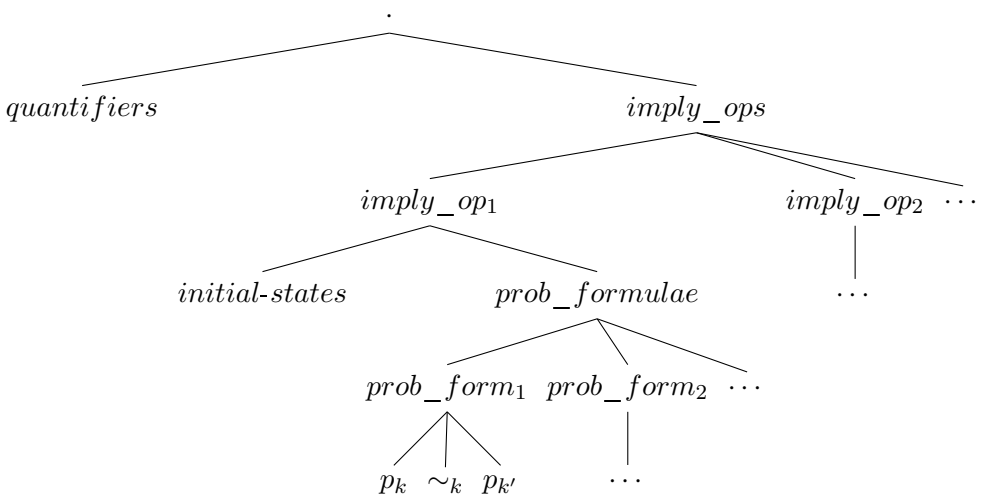


Figure 5.7: Final process tree of the HyperPCTL formula ψ.

5.3.7 Further Improvement: Storing the Self-composition Models

One further idea is to store the self-composition models.

The implementation computes the self-composition model for the according state variables σ for a sub-expression p_k. It could be the case that the same state variables occurs in a different sub-expression of p_l. Then, the same self-composition model has to be computed again. If the self-composition model with the corresponding state variables is stored, then a new computation of a self-composition model can be avoided if the exact same set of state variables occurs again.

In the implementation, the self-composition models are stored in a dictionary with their state variables as key.

Consider the example, shown in Figure 5.4, with the HyperPCTL formula

$$\psi = \forall\sigma.\forall\sigma'.[((t=n)_\sigma \wedge (t=y)_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond(r=n)_\sigma) \leq e^{ln^3} \cdot \mathbb{P}(\Diamond(r=n)_{\sigma'}))] \wedge$$
$$[((t=y)_\sigma \wedge (t=n)_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond(r=y)_\sigma) \leq e^{ln^3} \cdot \mathbb{P}(\Diamond(r=y)_{\sigma'}))]$$

To keep the vision, the quantifiers and initial states are left out. Only probability-formulae are depicted and considered. Usually, the probability-formulae would be evaluated as follows:

$$\underbrace{\Big(\underbrace{\mathbb{P}(\Diamond(r=n)_\sigma)}_{\text{1-ary self-composition for state variable } \sigma.} \leq \underbrace{e^{ln^3} \cdot \mathbb{P}(\Diamond(r=n)_{\sigma'})}_{\text{1-ary self-composition for state variable } \sigma'.}\Big) \text{ and}}_{probability\text{-}formula_1}$$
$$\underbrace{\Big(\underbrace{\mathbb{P}(\Diamond(r=y)_\sigma)}_{\text{1-ary self-composition for state variable } \sigma.} \leq \underbrace{e^{ln^3} \cdot \mathbb{P}(\Diamond(r=y)_{\sigma'})}_{\text{1-ary self-composition for state variable } \sigma'.}\Big)}_{probability\text{-}formula_2}$$

In $probability\text{-}formula_2$, the 1-ary self-composition is built for the state variable σ and σ' again although those models have already been built in $probability\text{-}formula_1$. The improvement now checks if such a model for the state variables already exist before computing a self-composition model. If a model for the state variables already exist, this self-composition model is used to evaluate the sub-expression p_l. If it does not exist, the self-composition model for the state variables is computed and stored for the case the state variables occur later in a sub-expression p_l again. If the state variables are used later in a sub-expression like in $probability\text{-}formula_2$, the corresponding self-composition model is reused.

In the following, the changed functions and variables are listed.

5.3.8 List of Functions and Variables of the improved Implementation

- `main(model, formula, arity_SC)`:
  This function gets three parameters as input, a DTMC `model` and a `formula` ψ. The parameter `arity_SC` does not play a role for the improved implementation but remains for the rest of the implementation so that the rest of the implementation can be reused for other types of models.

- `getBoundedVariables(formula, save_Model)`:
  This is the new and primary function of the improved algorithm implementation. It parses the passed `formula` and splits the formula into quantifiers, the *imply_op* initial states and probability-formulae. We call the function `HyperPCTL` to process the quantifier state variables so that the other variables of the formula can be matched to the quantifier state variable labels.

Otherwise, we do not have a connection between the quantifier state variables and the rest of the formula to evaluate it. If the formula consists of multiple *imply_ops*, then the following procedure is repeated for each *imply_op*.:

If the initial states do not exist, we return *true*, because a false implication is always *true*. If the initial states do exist, then we evaluate if the probability-formula holds. If there are several probability-formulae, then this process is repeated for every probability-formula. Each probability-formula is split into three parts, the probability expression p_k, the mathematical relation $\sim_k$ and the probability expression $p_{k'}$. For each probability-expression p the according self-composition model is built with the function `makeSelfComposition`. The number of state variables in a probability-expression is counted and passed to the function `makeSelfComposition` as arity for the self-composition model. In addition to the arity for the self-composition model, a new parameter is passed to that function, namely the indices for the self-composition model as explained in Section 5.3.5. The Boolean parameter `save_Model` is used to decide whether we want to store the self-composition model so that we can reuse that model later in a formula again, as explained in Section 5.3.7. A helper function `searchBuildCompositionModel` is introduced to search, store and return the self-composition models. The resulting self-composition model is then passed to the function `ProbMC` to compute the values of the probability-expression. The process of the functions `ProbMC` and `check_property` remains the same as in the original implementation. It is checked whether the values of the probability-expression hold for the relation. If they hold, then this probability-formula evaluates to *true*, otherwise *false*.

- `HyperPCTL(formula, model, labellingSet, ···)`:
  One small change has been made in this function, because the comparison part of every probability-expression to the mathematical relationship is not needed anymore. The rest remains the same.

- `makeSelfComposition(model, ···, arity_SC, indices)`:
  One new parameter is added to this function is called `indices`. This parameter is needed to apply the correct labellings to the resulting self-composition model, as explained in Section 5.3.5.

- `AddLabeltoLabelAllState(model, ···, arity_SC, indices)`:
  Since this function applies the labellings of the resulting self-composition model, the parameter `indices` has to be passed to this function as well.

- `searchBuildCompositionModel`
  `(model, ···, arity_SC, indices)`:
  This function uses the same parameters as `makeSelfComposition` because it returns a self-composition model. If the boolean `save_Model` is *false*, then we simply build the self-composition model with `makeSelfComposition` and return it to `getBoundedVariables`. If the boolean `save_Model` is *true*,

then we check if there already exists a self-composition model for the required sub-expression. If there already exists one, we return this self-composition model. Otherwise, we build the self-composition model, store it and return this model.

The global variables remain the same as in the original implementation. The changes are new local variables, as explained in Section 5.3.5.

- Local variables:
  - List `AllState`:
    This list originally was global. In the improved implementation, this list is changed to a local list. This is needed to reset itself with every call of the function `makeSelfComposition`. If there are some states in `VisitedState` which are not in `AllState`, then not all states have been processed yet, and the function proceeds to build the self-composition model. Otherwise, it omits building the self-composition model. If a new call for a self-composition model is made, then the states from the previous self-composition model are still contained in this list, and the function aborts the building process. It aborts the building process because it recognizes that all states have already been visited. To reset this list, the originally global list is changed to a local list.
  - List `VisitedState`:
    This list was originally a global list as well. With every call of the function `makeSelfComposition`, this list is reset so that the states of the previous process do not interfere with the new process for the new self-composition model.
  - Set `TempLabelAllState`:
    This new local set is used in the function `makeSelfComposition` instead of the global set `LabelAllState`. It is reset with every function call so that the labels of the previous process are not in this set. With the correct index labelling in the according self-composition model of a probability-expression, the evaluation of that probability-expression can be evaluated correctly. At the end of `makeSelfComposition`, the labels of the current resulting self-composition model are added to the global set `LabelAllState` so that all labels are contained in the global set.

5.3.9 Code Procedure and Pseudocode

After all the changes the final informal implementation procedure is as follows:
We abbreviate original model (OG-model) and self-composition model (SC-model).

Example 5.3.1. Consider the examples:

(i) $\underbrace{\forall\sigma_1.\forall\sigma_2.}\ \underbrace{(\text{init}_{\sigma_1} \wedge \text{init}_{\sigma_2})} \Rightarrow (\ \underbrace{\mathbb{P}(\Diamond a_{\sigma_1})}\ = \underbrace{\mathbb{P}(\Diamond a_{\sigma_2})}\ )$

- $\forall\sigma_1.\forall\sigma_2.$: Add labelling for state quantifiers with HyperPCTL and OG-model instead of 2-ary SC-model.
- $(\text{init}_{\sigma_1} \wedge \text{init}_{\sigma_2})$: If the initial states do not exist, return *true*. Otherwise, proceed.
- $\mathbb{P}(\Diamond a_{\sigma_1})$: p_1. Build 1-ary SC-model and the probabilities for all init states. Save 1-ary composition for state variable σ_1. Apply probMC with its 1-ary SC-model.
- $\mathbb{P}(\Diamond a_{\sigma_2})$: p_2. Build 1-ary SC-model and the probabilities for all init states. Save 1-ary SC-model for state variable σ_2. Apply probMC with its 1-ary SC-model.

Check if $p_1 = p_2$.

(ii) $\underbrace{\forall\sigma.\forall\sigma'.}\ [\underbrace{((t = n)_\sigma \wedge (t = y)_{\sigma'})} \Rightarrow (\ \underbrace{\mathbb{P}(\Diamond(r = n)_\sigma)}\ \leq \underbrace{e^{ln^3} \cdot \mathbb{P}(\Diamond(r = n)_{\sigma'})})]$

- $\forall\sigma.\forall\sigma'.$: HyperPCTL with OG-model.
- $((t = n)_\sigma \wedge (t = y)_{\sigma'})$: If the initial states do not exist, return *true*. Otherwise, proceed.
- $\mathbb{P}(\Diamond(r = n)_\sigma)$: p_1 of *probability-formula*$_1$. Check if SC-model for state variable σ exists: If yes, use that model. Else, build 1-ary SC-model for σ. Apply probMC on SC-model.
- $e^{ln^3} \cdot \mathbb{P}(\Diamond(r = n)_{\sigma'})$: p_2 of *probability-formula*$_1$. Check if SC-model for state variable σ exists: If yes, use that model. Else, build 1-ary SC-model for σ'. Apply probMC on SC-model.

Check if $p_1 \leq p_2$.

probability-formula$_1$

$\wedge\ [\underbrace{((t = y)_\sigma \wedge (t = n)_{\sigma'})} \Rightarrow (\ \underbrace{\mathbb{P}(\Diamond(r = y)_\sigma)}\ \leq\ \underbrace{e^{ln^3} \cdot \mathbb{P}(\Diamond(r = y)_{\sigma'})}\ )]$

- $((t = y)_\sigma \wedge (t = n)_{\sigma'})$: If the initial states do not exist, return *true*. Otherwise, proceed.
- $\mathbb{P}(\Diamond(r = y)_\sigma)$: p_1 of *probability-formula*$_2$. Check if SC-model exists for state variable σ: If yes, use that model. Else build 1-ary SC-model for σ. Apply probMC on SC-model.
- $e^{ln^3} \cdot \mathbb{P}(\Diamond(r = y)_{\sigma'})$: p_2 of *probability-formula*$_2$. Check if SC-model for state variable σ exists: If yes, use SC-model. Else build 1-ary SC-model for σ'. Apply probMC on SC-model.

Evaluate if $p_1 \leq p_2$.

probability-formula$_2$

In the first example, the implementation does not check if a self-composition for a particular state variable exists because ψ has only one probability-formula. In the second example, the implementation checks if a self-composition for a particular state variable has already been built because ψ has more than one probability-formula.

Informally, the implementation of the improvement can be summarized as follows:

- Apply variable renaming such that the quantified state variables are named $\sigma_1 \cdots \sigma_n$.
- Split the formula into two parts: $quantifiers$ and $imply\_op$. The labelling of the quantifiers is done by applying the function `HyperPCTL` with the model passed from `main`.

  The procedure for an $imply\_op$ is as follows:

  - If the initial states do not exist, return *true*. If the initial states do exist, proceed with the probability-formula.
  - Split $probability\text{-}formula_1$ into sub-expressions $p_k, p_{k'}$ and save the mathematical relation $\sim_k$ to compare the sub-expressions $p_k, p_{k'}$.
  - Counts how many state variables occur in sub-formula p_k and build the self-composition according to the number of dependent state variables in p_k.
  - If there are several probability-formulae, then check if there already exists a self-composition model for the dependent state variables of this sub-expression. If yes, then use this self-composition model, otherwise build the self-composition model and store it with the according state variables.
  - Evaluate each sub-formula by applying model checking on the according self-composition model.
  - Repeat this procedure for every p_k in $probability\text{-}formula_l$.
  - Compare the results of $p_k, \cdots, p_l$ according to the saved mathematical relation.
  - If the result is *true*, then the whole formula is satisfied. Otherwise, the formula is unsatisfied.

- If multiple probability-formulae exist, repeat the process for each probability-formula as explained.
- If the formula contains multiple $implies\_ops$, then repeat the procedure for each $imply\_op$.

With all changes in the implementation, the final pseudocode is shown according to the changes.

The pseudocode of the new functions are depicted in Algorithm 4, Algorithm 5 and Algorithm 6.

Algorithm 4 Changed function `main`($\mathcal{M}, \psi$)
HyperPCTL model checking based on algorithm [ÁB18].

Require: DTMC $\mathcal{M} = (S, P, AP, L)$, HyperPCTL quantified formula ψ
1: getBoundedVariables(ψ)

The function `main`, shown in Algorithm 4, gets a DTMC $\mathcal{M}$ and a formula ψ as input. It does not call `makeSelfcomposition` and `HyperPCTL` any more. Also, the function `main` does not verify if $\mathcal{M} \models \psi$ any more, hence it calls the function `getBoundedVariables` and passes the formula ψ to it.

The function `getBoundedVariables` depicted in Algorithm 5 verifies whether $\mathcal{M} \models \psi$ holds. First, the function extracts the quantifiers of the ψ and applies `HyperPCTL` on the quantifiers (lines 1–2). The model that `HyperPCTL` gets as parameter is the original DTMC $\mathcal{M}$. If we have multiple *imply_ops*, we repeat the procedure for each *imply_op* like the following (line 3): We extract the initial states and the probability-formulae. If we have multiple probability-formulae, we repeat the procedure for each probability-formula and do the following (lines 4–5). A *probability-formula*$_1$ is divided into p_k and $p_{k'}$ to see on how many state variables each sub-expression is dependent on (lines 7–8, 16–17). If we want to search and store the self-composition models the Boolean `save_Models` is set to *true*, and we call the helper function `searchBuildCompositionModel` to search and store the self-composition model. If we do not want to search and store a self-composition model, the state variables are counted, and a corresponding self-composition model is built (lines 9–13). Based on that model `ProbMC` is called with p_k as the passed formula and the corresponding self-composition model as a model. The result of p_k is saved (line 14). The same procedure is repeated for $p_{k'}$ (lines 16–23). Then the results of p_k and $p_{k'}$ are compared according to the mathematical relation in ψ. If the result is *true*, then the formula is satisfied. Otherwise, it returns *false* (lines 26–28).

The function `searchBuildCompositionModel` in Algorithm 6 has the same parameters as the function `makeSelfComposition` since it returns a model as `makeSelfComposition`. It searches for a model, and if it exists, this model is returned (lines 1-2). Otherwise, a self-composition model is built, stored and returned (lines 4-6).

The function `HyperPCTL` remains almost the same as the original procedure as depicted in Algorithm 2. Two things have been changed. First, `HyperPCTL` does not call the function `ProbMC` any more. Last, this function does not have to compare the results of `ProbMC` any more. That part is taken care of the function `getBoundedVariables`.

The function `ProbMC` remains unchanged and is depicted in Algorithm 3.

Algorithm 5 Function `getBoundedVariables(formula, boolean save_model)`
Improving HyperPCTL model checking algorithm with the possibility to store the self-composition model. We abbreviate `makeSelfComposition` with `makeSC` and `compositionModel` with `CM`.

Require: HyperPCTL quantified formula ψ
Ensure: Whether $\mathcal{M} \models \psi$

```
1: quantifiers := extract quantifiers from ψ
2: HyperPCTL(quantifiers, M, labellingSet,···)
3: for (length of imply_ops > 1) do
4:   initStates := extract initial states from formula
5:   for (length of probability-formulae > 1) do
6:
7:     p_k := left side of probability-formula_k
8:     varLeft := state variables of p_k
9:     if save_Models = True then
10:      CM :=searchBuildCM(model, len(varLeft), indices)
11:    else
12:      CM := makeSC(model, arity_SC, indices)
13:    end if
14:    result1 := ProbMC(p_k, CM, labellingSet, ···)
15:
16:    p_k' := right side of probability-formula_k
17:    varRight := state variables of p_k'
18:    if save_Models = True then
19:      CM :=searchBuildCM(model, len(varRight), indices)
20:    else
21:      CM := makeSC(model, arity_SC, indices)
22:    end if
23:    result2 :=ProbMC(p_k', CM, labellingSet, ···)
24:
25:    ∼_k :=mathematical Relation ∼_k
26:    if result1 ≁_k result2 then
27:      return False
28:    end if
29:  end for
30: end for
```

Algorithm 6 Function `searchBuildCompositionModel(model, arity_SC indices)`
Function in which we search and store a self-composition model as well as return a self-composition model if it has been built in ψ before.

Require: model, arity for self-composition model, indices
Ensure: $\mathcal{M}^n$

```
1: if model exists then
2:     return model
3: else
4:     CM :=makeSelfComposition(model, arity_SC, indices)
5:     save CM
6:     return CM
7: end if
```

Chapter 6

Evaluation

In this chapter, we present the evaluation of the implementation. The following aspects were considered in the original implementation and the new improved one:

- The number of states and transitions of the self-composition models.
- The run-time of the implementations.

6.1 Number of States and Transitions of Self-composition Models

Regarding the number of states and transitions of the original input model $\mathcal{M}$ and those of the corresponding self-composition model $\mathcal{M}^n$ we obtain the following measured results in the original implementation compared to the new one without storing the self-composition models:

For the first example, shown in Figure 5.3: The original model $\mathcal{M}$ consists of 7 states and 11 transitions. The corresponding 2-ary self-composition model $\mathcal{M}^2$ consists of 23 states and 53 transitions. In the improved one, the 1-ary self-composition model $\mathcal{M}^1$ consists of the same number of states and transitions as the original model. One difference is that we do not have one model as in the original implementation but two 1-ary self-composition models since we build a 1-ary self-composition model for each sub-formula. Altogether we have 14 states and 22 transitions. However, the number of transitions and states is distinctly smaller than the 2-ary self-composition model. If we consider the improved implementation where we store the self-composition models, the count of self-composition models stays the same since the formula has only one probability-formula.

For the second example, shown in Figure 5.4: The original model $\mathcal{M}$ consists of 5 states and 8 transitions. The corresponding 2-ary self-composition model $\mathcal{M}^2$ consists of 13 states and 32 transitions. In the improved implementation, the corresponding 1-ary self-composition model $\mathcal{M}^1$ has the same number of states and transitions as the original model. In this case, we have four 1-ary self-composition models

because we build the according self-composition model to the corresponding state variables for each sub-formula. Altogether we have 20 states and 32 transitions. If we consider the improved implementation where we store the self-composition models, the count of self-composition models reduces to two having 10 states and 16 transitions.

We tested a third example which is the model of the first example, shown in Figure 5.3, with the following formula to be evaluated:

$$\psi = \forall\sigma.\forall\sigma'.\forall\sigma''.[(init_\sigma \wedge init_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond a_\sigma) = \mathbb{P}(\Diamond a_{\sigma'})) \wedge (init_{\sigma'} \wedge init_{\sigma''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma'}) = \mathbb{P}(\Diamond a_{\sigma''}))]$$

The resulting self-composition model of the original implementation results in 79 states and 257 transitions. The resulting self-composition model of the improved one without storing the self-composition models results in four models, each with seven states and eleven transitions. Overall we have 28 states and 44 transitions. If we compare the improved implementation where we store the self-composition models, the count of self-composition models reduces to two having 14 states and 22 transitions.

The fourth example is an extension of the first example, shown in Figure 5.3, as well with the following formula.

$$\psi = \forall\sigma.\forall\sigma'.\forall\sigma''.\forall\sigma'''.[(init_\sigma \wedge init_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond a_\sigma) = \mathbb{P}(\Diamond a_{\sigma'})) \wedge (init_{\sigma'} \wedge init_{\sigma''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma'}) = \mathbb{P}(\Diamond a_{\sigma''})) \wedge (init_{\sigma''} \wedge init_{\sigma'''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma''}) = \mathbb{P}(\Diamond a_{\sigma'''}))]$$

The resulting self-composition model of the original implementation results in 287 states and 1221 transitions. The resulting self-composition model of the improved one without storing the self-composition models results in six models, each with 7 states and 11 transitions. In total, we have 42 states and 66 transitions. If we consider the improved implementation where we store the self-composition models, the count of self-composition models reduces to four having 28 states and 44 transitions.

The fifth and last example is an extension of the first example, shown in Figure 5.3, as well with the following formula.

$$\psi = \forall\sigma.\forall\sigma'.\forall\sigma''.\forall\sigma'''.\forall\sigma''''[(init_\sigma \wedge init_{\sigma'}) \Rightarrow (\mathbb{P}(\Diamond a_\sigma) = \mathbb{P}(\Diamond a_{\sigma'})) \wedge (init_{\sigma'} \wedge init_{\sigma''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma'}) = \mathbb{P}(\Diamond a_{\sigma''})) \wedge (init_{\sigma''} \wedge init_{\sigma'''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma''}) = \mathbb{P}(\Diamond a_{\sigma'''})) \wedge (init_{\sigma'''} \wedge init_{\sigma''''}) \Rightarrow (\mathbb{P}(\Diamond a_{\sigma'''}) = \mathbb{P}(\Diamond a_{\sigma''''}))]$$

The resulting self-composition model of the original implementation results in 1036 states and 5092 transitions. The resulting self-composition model of the improved one without storing the self-composition models results in six models, each with 7 states and 11 transitions. In total, we have 77 states and 88 transitions. If we compare the improved implementation where we store the self-composition models, the count of self-composition models reduces to four having 35 states and 55 transitions.

Ex.	OG	Improved impl. without storing the SC-models	Improved impl. with storing the SC-models
1.	1 SC-model: 23 states, 57 transitions	2 SC-models each: 7 states, 11 transitions Total: 14 states, 22 transitions	2 SC-models each: 7 states, 11 transitions Total: 14 states, 22 transitions
2.	1 SC-model: 13 states, 32 transitions	4 SC-models each: 5 states, 8 transitions Total : 20 states, 32 transitions	2 SC-models each: 5 states, 8 transitions Total: 10 states, 16 transitions
3.	1 SC-model: 79 states, 257 transitions	4 SC-models: 7 states, 11 transitions Total: 28 states, 44 transitions	2 SC-models: 7 states, 11 transitions Total: 14 states, 22 transitions
4.	1 SC-model: 287 states, 1221 transitions	6 SC-models: 7 states, 11 transitions Total: 42 states, 66 transitions	4 SC-models: 7 states, 11 transitions Total: 28 states, 44 transitions
5.	1 SC-model: 1036 states, 5092 transitions	8 SC-models: 7 states, 11 transitions Total: 77 states, 88 transitions	5 SC-models: 7 states, 11 transitions Total: 35 states, 55 transitions

Table 6.1: Number of states and transitions of self-composition models of the implementations.

We observe that the sizes of self-compositions are reduced, hence during model checking we have to evaluate fewer states and transitions. We have an even lower count of self-composition models if we apply the implementation where we store them. A summary with the number of states and transitions is listed in Table 6.1.

We abbreviate original (OG), implementation (impl.) and self-composition model (SC-model) in the tables.

With the improved implementation, the number of states and transitions are reduced except for in Example 2. With the improved implementation where we store the self-composition models, the number of states and transitions of the self-composition models are reduced at least by half in comparison to the original implementation.

6.2 Runtime Evaluation

In our improvement, we reduce the arity of the self-composition models to make the model checking algorithm more efficient. In comparison to the original implementation, we do not have one n-ary self-composition model $\mathcal{M}^n$ but at least two smaller self-composition models. We evaluate fewer states and transitions at the expense of time to build, load and store the models which introduces a noticeable overhead during the time measurement.

We observe for all three implementation variants (original implementation, improved implementation without storing the self-composition models, improved implementation with storing the self-compisition models) that more quantifiers in a

Ex.	OG impl.	Improved impl. without storing the SC-model	Speed-up: OG impl./Improved impl.
1.	0.047s	0.054s	0.87
2.	0.048s	0.092s	0.52
3.	0.101s	0.095s	1.06
4.	0.665s	0.134s	4.81
5.	4.939s	0.280s	17.64

Table 6.2: The run-times of the original implementation and the improved one without storing the self-composition models.

Ex.	OG impl.	Improved impl. with storing the SC-model	Speed-up: OG impl./Improved impl.
1.	0.047s	0.055s	0.85
2.	0.048s	0.055s	0.87
3.	0.101s	0.077s	1.31
4.	0.665s	0.094s	7.07
5.	4.939s	0.258s	19.14

Table 6.3: The run-times of the original implementation and the improved one with storing the self-composition models and the achieved speedup.

formula cause a longer runtime (see Table 6.2 and 6.3).

Comparing the original implementation with the improved implementation (with or without storing the self-composition models), the original implementation has a shorter runtime for Examples 1 and 2, while the improved implementation has a shorter runtime for Examples 3, 4 and 5. The original implementation is faster for Examples 1 and 2 because building and using several smaller self-composition models is more time-consuming than building and using a single larger model for formulae with few quantifiers. The improved implementation (with or without storing the self-composition models) is faster for Examples 3, 4 and 5 because building and using several smaller self-composition models is faster than building and using a single large model for formulae with more quantifiers. We can see a trend that the relative speed-up of the improved implementation increases with the number of quantifiers in the formula, i.e., we expect the speed-up to grow for even more quantifiers. If we additionally store the self-composition models, then the runtime of the improved implementation decreases even further, because the self-composition models can be reused.

Chapter 7

Conclusion and Outlook

In this Master thesis, we increased the efficiency of a model checking algorithm for a fragment of the logic HyperPCTL. Ábrahám and Bonakdarpour [ÁB18] proposed the original HyperPCTL model checking algorithm. A discrete-time Markov Chain (DTMC) can interpret a HyperPCTL state formula. The algorithm takes a DTMC $\mathcal{M}$ and a HyperPCTL formula ψ as input and verifies if $\mathcal{M} \models \psi$ holds. The original algorithm takes the given DTMC $\mathcal{M}$ and builds an n-ary self-composition model $\mathcal{M}^n$. There, n is the number of quantifiers in the HyperPCTL formula ψ. Using that self-composition model $\mathcal{M}^n$, the formula ψ (i.e., every sub-expression of ψ) is verified. The original HyperPCTL model checking algorithm allows nested formulae.

In our improved implementation, we reduce the arity of the self-composition model by analysing the HyperPCTL formula ψ first before computing the self-composition model. To reduce the arity, we only allow non-nested HyperPCTL formulae. We count how many state variables the sub-formula has. Based on the number of the state variables a self-composition model with the number of state variables as arity is built, and we verify the sub-expression on this self-composition model. We do not have one big n-ary self-composition model as in the original implementation, but multiple small self-composition models since a sub-expression usually does not dependent on all state variables in ψ. Thus, we do not need to build an n-ary self-composition model $\mathcal{M}^n$. A further improvement is that we store the self-composition model with the corresponding state variable. If a sub-expression has the same state variables as a different sub-expression that appeared in ψ before, then the self-composition model does not have to be built again.

In our evaluation, we tested five examples to compare the original implementation and the improved one. We compared two aspects, namely the number of states and transitions of the self-composition models as well as the runtime of the implementations. The original implementation was compared to the improved one where we do not store the self-composition models as well as the improved one where we store the self-composition models. We have built bigger examples where the HyperPCTL formula ψ contains up to five quantifiers to see what impact the n-ary self-composition model has. The number of states and transitions is reduced in the self-composition

models compared to the n-ary self-composition model $\mathcal{M}^n$. Additionally, we store the self-composition models. Thereby, we can reuse them for other sub-formulae with the same state variables and save computation time. We assumed correctly that the bigger the number of quantifiers in ψ gets, the longer the original implementation needs to verify if $\mathcal{M} \models \psi$ holds. The improved implementation without storing the self-composition model has a speed-up of 17.64 compared to the original implementation for an example where ψ contains five quantifiers. The speed up of the improved implementation with storing the self-composition model is by factor 19.14 compared to the original implementation. With a more common HyperPCTL formula where we mostly have two quantifiers, the original implementation can be faster than the improved one, depending on the input model $\mathcal{M}$ and formula ψ, since the building, loading and storing of self-composition models is time-consuming.

As for future work, one possibility is to reduce the arity of the self-composition models as in this Master thesis, but for nested HyperPCTL formulae. A further aspect could be to implement the model checking algorithm without using the function `HyperPCTL`. Also, one could try to improve the function `makeSelfComposition`. Instead of building each self-composition model from scratch, one could reuse a self-composition model with an identically sized set of state variables and adapt the labellings accordingly.

Glossary

PCTL Probabilistic Computation Tree Logic

PCTL* Probabilistic Computation Tree Logic*

CTL Computation Tree Logic

CTL* Computation Tree Logic*

LTL Linear Temporal Logic

DTMC discrete-time Markov Chain

CTMC continuous-time Markov Chain

MDP Markov Decision Process

SMC statistical model checker

PyCarl **Py**thon **C**omputer **AR**ithmetic and **L**ogic library

Carl **C**omputer **AR**ithmetic and **L**ogic library

OG-model original model

SC-model self-composition model

OG original

impl. implementation

Bibliography

[ÁB18] Erika Ábrahám and Borzoo Bonakdarpour. HyperPCTL: A temporal logic for probabilistic hyperproperties. In *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*, pages 20–35, 2018.

[ÁBBD20] Erika Ábrahám, Ezio Bartocci, Borzoo Bonakdarpour, and Oyendrila Dobe. Parameter synthesis for probabilistic hyperproperties. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 12–31. EasyChair, 2020.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.

[BDH+17] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. JANI: Quantitative model and tool interaction. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 151–168, 2017.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CFST19] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume

11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019.

[CG04] Frank Ciesinski and Marcus Größer. On probabilistic computation tree logic. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 147–188. Springer, 2004.

[CS10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A Storm is coming: A modern probabilistic model checker. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 592–600. Springer, 2017.

[EHKZ13] Christian Eisentraut, Holger Hermanns, Joost-Pieter Katoen, and Lijun Zhang. A semantics for every GSPN. In José Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, volume 7927 of *Lecture Notes in Computer Science*, pages 90–109. Springer, 2013.

[FHH18] Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. MGHyper: Checking satisfiability of HyperLTL formulas beyond the ∃*∀* fragment*. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 521–527. Springer, 2018.

[FHS17] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 564–570. Springer, 2017.

[FHST18] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. RVHyper: A runtime verification tool for temporal hyperproperties. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences*

on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II, volume 10806 of *Lecture Notes in Computer Science*, pages 194–200. Springer, 2018.

[FRS15] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[III90] James W. Gray III. Probabilistic interference. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 7-9, 1990*, pages 170–179. IEEE Computer Society, 1990.

[IZw20] Software Modeling and Verification i2, RWTH Aachen University. `https://moves.rwth-aachen.de/`, 2020.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[Pri20] The PRISM language introduction. `http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction`, 2020.

[PyC20] PyCarl documentation. `https://moves-rwth.github.io/pycarl/`, 2020.

[Sto20a] A modern model checker for probabilistic systems. `http://www.stormchecker.org/`, 2020.

[Sto20b] Storm dependencies. `http://www.stormchecker.org/documentation/obtain-storm/dependencies.html`, 2020.

[Sto20c] Stormpy documentation. `https://moves-rwth.github.io/stormpy/`, 2020.

[WNBP19] Yu Wang, Siddhartha Nalluri, Borzoo Bonakdarpour, and Miroslav Pajic. Statistical model checking for probabilistic hyperproperties. *CoRR*, abs/1902.04111, 2019.