MEASURING PROGRESS OF PROBABILISTIC LTL MODEL CHECKING

ELISE CORMIE

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JUNE 2012

MEASURING PROGRESS OF PROBABILISTIC LTL MODEL CHECKING

by **Elise Cormie**

a thesis submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

MEASURING PROGRESS OF PROBABILISTIC LTL MODEL CHECKING

by **Elise Cormie**

Examination Committee Members:

1. Franck van Breugel
2. Eric Ruppert
3. Nick Cercone
4. Stephen Watson

Abstract

Zhang and Van Breugel recently developed a progress measure for probabilistic model checking. It expresses the minimum probability of a linear temporal logic (LTL) property being satisfied, given a partially-explored state space. They showed how to calculate this measure for invariants.

In this thesis, we expand on the above research. Previously, how to calculate progress for LTL formulas other than invariants was unknown. We show how to calculate progress for a large subset of LTL formulas, which we call unipolar. We also present an efficient algorithm to calculate progress for properties of the form $p \,\mathcal{U}\, q$, where p and q are atomic propositions. Furthermore, we prove several other properties of the progress measure, such as the relationship between the progress for a property and for the negation of that property. In addition, we use a graphics processing unit to improve the speed of progress calculations for invariants.

Acknowledgements

First of all, I am extremely thankful to Franck van Breugel, who was everything I could have wanted in a supervisor. This work would not have been possible without his help, advice, encouragement and brilliant ideas.

I would also like to thank Xin Zhang, who patiently helped me set up and use his program. Thanks to Nick Cercone, Eric Ruppert and Stephen Watson, for serving on my exam committee and for their thoughtful feedback on my thesis. I also thank Eric Ruppert for his help with my presentation. Thanks as well to Kim Larsen, who gave us the idea for Proposition 3.5. And thanks to Ouma Jaipaul Gill, who as graduate assistant always had helpful answers to my countless questions.

Table of Contents

1 Introduction

Due to the infamous state space explosion problem [23, Section 1], model checking a property of source code that contains randomization often fails. In many cases, the probabilistic model checker simply runs out of memory without reporting any useful information. In [30], Zhang and Van Breugel propose a progress measure for probabilistic model checkers. This measure captures the amount of progress the model checker has made with its verification effort. Even if the model checker runs out of memory, the amount of progress may provide useful information.

Our aim is to develop a theory that is applicable to probabilistic model checkers in general. Our initial development has been guided by a probabilistic extension of the model checker Java PathFinder (JPF) created by Zhang [27]. This extension can check properties, expressed in linear temporal logic (LTL), of Java code containing randomized sequential algorithms.

We model the code under verification as a probabilistic transition system (PTS), and the systematic search of the system by the model checker as the set of explored transitions of the PTS. We focus on linear-time properties, in particular those expressed in LTL. The progress measure is defined in terms of the set of explored transitions and the linear-time property under verification. The progress measure returns a real number in the interval $[0, 1]$. The larger this number, the more progress the model checker has made with its verification effort.

Zhang and Van Breugel showed that their progress measure provides a lower bound for the measure of the set of execution paths that satisfy the linear-time property under verification. If, for example, the progress is 0.9999, then the probability of encountering a violation of the linear-time property when we run the code is at most 0.0001. Hence, despite the fact the model checker may fail by running out of memory, the verification effort may still be a success by providing an acceptable upper bound on the probability of a violation of the property.

Zhang and Van Breugel showed how to calculate the progress measure when the property under verification is an invariant. However, how to calculate progress for other types of linear-time property remained an open question. In this thesis, we present algorithms that can calculate progress for a wide range of properties.

In Chapter 4, we show how to calculate the progress measure for properties

expressed by a large subset of LTL which we call *unipolar*. Unipolar LTL formulas are those that can be written in positive normal form without both an atomic proposition and the negation of that same proposition. For instance, $a \vee \neg b$ is unipolar, and $a \vee \neg a$ is not.

A wide range of useful formulas can be written in unipolar LTL. Examples include properties of the well-known mutual exclusion problem for concurrent processes. That two threads cannot be in their critical sections at the same time can be expressed as $\Box(\neg \text{crit}_1 \vee \neg \text{crit}_2)$, and both threads being able to eventually enter their critical sections can be written as $(\Box\Diamond \text{crit}_1) \wedge (\Box\Diamond \text{crit}_2)$ [2, Chapter 5]. Both of these LTL formulas are unipolar.

The time complexity of the algorithm for unipolar LTL is exponential in the size of the formula, and polynomial in the size of the searched space. Since the size of LTL formulas is normally small, we believe this algorithm will be useful.

In Chapter 6, we present an algorithm for calculating progress for LTL formulas of the form $p\mathcal{U}\,q$, where p and q are atomic propositions. Progress for these formulas could be calculated using the unipolar LTL algorithm in Chapter 4. However, we develop a more efficient algorithm that allows the progress toward verifying $p\mathcal{U}\,q$ to be calculated by determining the reachability probability of a state in a modified system.

In addition to algorithms to compute progress, we present other useful algorithms and theorems. Chapter 5 contains a polynomial-time algorithm to calculate a lower bound for the progress measure, which can be used for any LTL formula. The lower bound is calculated in the same manner as progress for invariants in [30], so in effect this algorithm has already been implemented and tested. The bound is tight for invariants, and possibly other classes of properties. This lower bound is useful in principle because, like the progress measure, it provides a minimal probability of a property being satisfied in the system. However, in some situations the lower bound is very low while the actual progress is high, making it less informative.

In Chapter 3, we show some relationships between the progress of verifying a property, the progress of verifying the negation of the property, and finding a violation of the property. We also show that the progress of a property, and the progress of its negation, form upper and lower bounds on the actual measure of executions which satisfy that property. The proofs in Chapter 3 help to further define the properties of the progress measure, and could be useful in future work.

Chapter 7 and 8 depart from the theoretical proofs of previous chapters. In them, we empirically explore the possibility of using a graphics processing unit (GPU) to accelerate the calculation of progress. Zhang and Van Breugel [30] showed that progress for invariants can be determined by calculating the probability of reaching a particular state in a modified system. As shown in Chapter 6,

the progress for properties of the form $p \mathcal{U} q$ can be calculated in a similar fashion. Thus, the efficient computation of reachability probabilities is useful for progress calculations. These reachability probabilities play a key role in several other areas, including probabilistic model checking more generally (see, for example, [2, Chapter 10]) and performance evaluation (see, for example, [16]). Hence, these results also have more general applications.

The general reachability probability problem can be stated as follows: given a PTS, an initial state of the PTS, and a set of goal states of the PTS, we are interested in the probability of reaching any of the goal states from the initial state. This probability is known as the reachability probability.

As we will sketch in Chapter 7, computing reachability probabilities can be reduced to solving a linear equation of the form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$, where $\mathbf{A}$ is an $n \times n$-matrix and $\mathbf{b}$ is an n-vector. Although the equation can be solved by inverting the matrix $\mathbf{A}$, such an approach becomes infeasible for large matrices due to the computational complexity of matrix inversion. For instance, Gauss-Jordan elimination has time complexity $O(n^3)$ (see, for example, [22, Section 2]). For large matrices, iterative methods are used instead.

The iterative methods compute successive approximations to obtain a more accurate solution to the linear system at each iteration. In this thesis, we consider two linear methods, namely the Jacobi method and the biconjugate gradient stabilized (BiCGStab) method. We have implemented a sequential version of each method in C, and a parallel version using NVIDIA's compute unified device architecture (CUDA). CUDA allows us to run C code on a GPU with hundreds of cores. Currently, CUDA is only supported by NVIDIA GPUs.

To compare the performances of our four implementations, we constructed three sets of tests. First of all, we randomly generated matrices with varying sizes and densities. Our experiments show that the BiCGStab method is superior to the Jacobi method for denser matrices. They also demonstrate a fairly consistent performance benefit from using CUDA to implement the Jacobi method. However, we observe that the CUDA version of the BiCGStab method is only beneficial for larger, denser matrices. For the smaller and sparser matrices, the sequential version of the BiCGStab method outperforms the CUDA version.

Secondly, we used the extension of the model checker JPF [27] to generate transition probability matrices corresponding to the Java code of two randomized sequential algorithms. The Jacobi method performed better than the BiCGStab method for these matrices. This supports the conjecture by Bosnacki et al. [6, 7] that the Jacobi method is superior to Krylov subspace methods, a class to which the BiCGStab method belongs, for probabilistic model checking.

Finally, we randomly generated matrices with the same sizes and densities as

the matrices produced by JPF. We obtained very similar results. This suggests that size and density are the main determinants of which implementation performs best on probabilistic model checking data, and whether CUDA will be beneficial, rather than other properties unique to matrices found in probabilistic model checking.

We also examine the related problem of calculating whether or not each state in a PTS can reach a set of goal states, which we refer to as binary reachability. This calculation is performed to reduce the size of the matrix before using an iterative solver such as the Jacobi or BiCGStab method. We test a sequential and a GPU binary reachability algorithm, and observe that the GPU version gives improved calculation speed. However, due to the construction of this particular algorithm, it is most useful on dense matrices, and model checking problems for which the graph of the state space has a high vertex degree.

The work of Chapter 4 to 7 is by Cormie, and Chapter 3 is joint work with Van Breugel. Parts of Chapter 7 and 8 appeared in [9], and some material in Chapter 3 and 4 appeared in [10].

2 Background

2.1 Measure Theory

The concept of the progress measure is rooted in measure theory. Thus, in this section we review some of the basic concepts of measure theory which are used to define the progress measure. We also present concepts that are used in later chapters to prove properties of the progress measure. Our definitions are based on [13] and [5].

First, we present the idea of a countable set. A set is countable if there is a mapping from each element of the set to a unique natural number. The sets we discuss in this thesis are all countable, a fact that we use in several definitions and proofs.

Definition 2.1 *A set X is* countable *if there exists a injective function $f : X \to \mathbb{N}$.*

Example 2.2 *Examples of countable sets include any finite set, and any infinite subset of $\mathbb{N}$, such as the set of all prime numbers. On the other hand, the set of real numbers, $\mathbb{R}$, is not countable.*

Below, we present three well-known properties of countable sets.

Proposition 2.3 *If a set X is countable, and $Y \subseteq X$, then Y is countable.*

Proof See [13, Chapter 2]

Proposition 2.4 *A countable union of countable sets is countable.*

Proof See [17, Chapter 7].

Proposition 2.5 *Let X be a countable set. Then X^*, the set of finite sequences of elements of X, is countable.*

Proof See [26, Theorem 7.2.4].

Next, we introduce the concepts that define a *measure*, such as the progress measure. Firstly, we define the σ-algebra, which forms the basis of a measurable space.

Definition 2.6 *Let X be a set. A* σ-algebra over X *is a set Σ of subsets of X that satisfy the following conditions:*

- $X \in \Sigma$*;*
- *if $A, B \in \Sigma$, then $A \setminus B \in \Sigma$;*
- *closed under countable unions: for every sequence $\langle A_n \rangle_{n \in \mathbb{N}}$ in Σ, $\bigcup_{n \in \mathbb{N}} A_n \in \Sigma$.*

Given a σ-algebra, we can create a measure as defined below.

Definition 2.7 *Let X be a set, and let Σ be a σ-algebra over X. A* measure *on Σ is a function $\mu : \Sigma \to [0, \infty]$ such that:*

- $\mu(\emptyset) = 0$*;*
- *μ is countably additive: for every sequence of pairwise disjoint sets $\langle A_n \rangle_{n \in \mathbb{N}}$ in Σ, $\mu\left(\bigcup_{n \in \mathbb{N}} A_n\right) = \sum_{n \in \mathbb{N}} \mu(A_n)$.*

If, in addition to the above, $\mu(X) = 1$, then μ is a probability measure.

Definition 2.8 *A* measure space *is a triple $\langle X, \Sigma, \mu \rangle$, where X is a set, Σ is a σ-algebra over X, and μ is a measure on Σ. If μ is a probability measure, then $\langle X, \Sigma, \mu \rangle$ can also be called a* probability space.

Below, we state an important property of a measure.

Proposition 2.9 *Let μ be a measure on a σ-algebra Σ. Then μ is* monotone*: for all $A, B \in \Sigma$, if $A \subseteq B$ then $\mu(A) \leq \mu(B)$.*

2.2 Probabilistic Model Checking

In probabilistic model checking, a model of a program is created, then checked to see if it satisfies desired properties. In this section, we present a mathematical definition of a program model used in probabilistic model checking, called a probabilistic transition system. We also present linear temporal logic, which is used to specify properties of the model.

2.2.1 Probabilistic Transition Systems

A program with probabilistic characteristics can be modeled as a probabilistic transition system (PTS). A PTS has states that correspond to program states, and transitions between states that correspond to possible execution steps. These transitions are associated with probability distributions: each transition has a probability in the interval $(0, 1]$, and the sum of outgoing transition probabilities from each state is 1. Each state is associated with a set of *atomic propositions*, or *labels*, which capture whether the state satisfies some property.

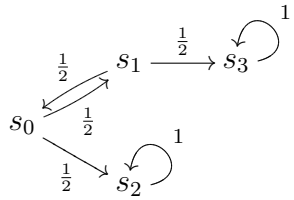
Definition 2.10 *A* probabilistic transition system *(PTS) is a tuple* $\langle S, T, AP, s_0, \text{source}, \text{target}, \text{prob}, \text{label}\rangle$ *consisting of*

- *a countable set* S *of states,*
- *a countable set* T *of transitions,*
- *a set* AP *of atomic propositions,*
- *an initial state* s_0*,*
- *a function* $\text{source} : T \to S$*,*
- *a function* $\text{target} : T \to S$*,*
- *a function* $\text{prob} : T \to (0, 1]$*, and*
- *a function* $\text{label} : S \to 2^{AP}$

such that

- $s_0 \in S$ *and*
- *for all* $s \in S$*,* $\sum_{source(t)=s} \text{prob}(t) = 1$*.*

Example 2.11 *The probabilistic transition system* $\mathcal{S}$ *depicted by*



has four states and six transitions. In this example, we use the indices of the source and target to name the transitions. For example, the transition from s_0 *to* s_2 *is*

named t_{02}. *Given this naming convention, the functions* $\text{source}_{\mathcal{S}}$ *and* $\text{target}_{\mathcal{S}}$ *are defined in the obvious way. For example,* $\text{source}_{\mathcal{S}}(t_{02}) = s_0$ *and* $\text{target}_{\mathcal{S}}(t_{02}) = s_2$. *The function* $\text{prob}_{\mathcal{S}}$ *can be easily extracted from the above diagram. For example,* $\text{prob}_{\mathcal{S}}(t_{02}) = \frac{1}{2}$. *All states are labelled with the atomic proposition* p *and the states* s_1 *and* s_2 *are also labelled with the atomic proposition* q. *Hence, for example,* $\text{label}_{\mathcal{S}}(s_2) = \{p, q\}$.

Instead of $\langle S, T, AP, s_0, \text{source}, \text{target}, \text{prob}, \text{label}\rangle$ we usually write $\mathcal{S}$ and we denote, for example, its set of states by $S_{\mathcal{S}}$. We model the potential executions of the system under verification as execution paths of the PTS.

Definition 2.12 *An* execution path *of a PTS* $\mathcal{S}$ *is an infinite sequence of transitions* $t_1 t_2 \ldots$ *such that*

- *for all* $i \geq 1$, $t_i \in T_{\mathcal{S}}$,
- *for all* $i \geq 1$, $\text{target}_{\mathcal{S}}(t_i) = \text{source}_{\mathcal{S}}(t_{i+1})$.

The set of all execution paths such that $s = \text{source}_{\mathcal{S}}(t_1)$ *is denoted by* $\text{Exec}^s_{\mathcal{S}}$. *The set of all execution paths that begin in the initial state,* $\text{Exec}^{s_0 \mathcal{S}}_{\mathcal{S}}$, *is denoted by* $\text{Exec}_{\mathcal{S}}$.

Example 2.13 *Consider the PTS of Example 2.11. For this system,* ${t_{02}t_{22}}^{\omega}$, ${t_{01}t_{13}t_{33}}^{\omega}$, *and* ${t_{01}t_{10}t_{02}t_{22}}^{\omega}$ *are examples of execution paths in* $\text{Exec}_{\mathcal{S}}$.

We denote the set of finite prefixes of execution paths in $\text{Exec}^s_{\mathcal{S}}$ by $\text{pref}(\text{Exec}^s_{\mathcal{S}})$. We call the prefix of length zero ϵ, and define $\text{source}_{\mathcal{S}}(\epsilon) = \text{target}_{\mathcal{S}}(\epsilon) = s_{0\mathcal{S}}$.

Given $e \in \text{Exec}^s_{\mathcal{S}}$, and $i \geq 0$, we use $e[i]$ to denote the prefix of e that consists of the first i transitions in e. For instance, consider the PTS of Example 2.11, and the execution path $e = t_{01}t_{13}{t_{33}}^{\omega}$. Then $e[0] = \epsilon$, $e[1] = t_{01}$, and $e[4] = t_{01}t_{13}t_{33}t_{33}$.

For $e \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$, we use $|e|$ to denote the length of e. For instance, if $e = t_{01}t_{13}t_{33}$, then $|e| = 3$. We use $e[i]$ as we do for execution paths, with the additional condition that $i \leq |e|$.

Given $e \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$, we use $\text{target}_{\mathcal{S}}(e)$ to denote the target of the last transition in e, and $\text{source}_{\mathcal{S}}(e)$ to denote the source of the first transition in e. For instance, consider the PTS $\mathcal{S}$ of Example 2.11, and $e = t_{01}t_{13}t_{33}$. Then $\text{source}_{\mathcal{S}}(e) = s_0$, and $\text{target}_{\mathcal{S}}(e) = s_3$.

Given $e \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$ or $e \in \text{Exec}^s_{\mathcal{S}}$, and a transition t, we use $t \in e$ to indicate that the transition t occurs in e. For example, if $e = t_{01}t_{13}t_{33}$, then $t_{01} \in e$.

Below, we define the prefix relation, which expresses whether one sequence of transitions is a prefix of another.

Definition 2.14 *The relation* $\sqsubseteq \subseteq \text{pref}(\text{Exec}_{\mathcal{S}}) \times \text{pref}(\text{Exec}_{\mathcal{S}})$ *is defined by*

$$e_1 \sqsubseteq e_2 \text{ if } e_1 \text{ is a prefix of } e_2.$$

Example 2.15 *Consider the PTS of Example 2.11. Then* $t_{01} \sqsubseteq t_{01}t_{13}$, $t_{02} \sqsubseteq t_{02}$, *and* $t_{02} \not\sqsubseteq t_{01}t_{13}$.

Proposition 2.16 $\langle \text{pref}(\text{Exec}_{\mathcal{S}}), \sqsubseteq \rangle$ *is a partial order.*

Proof See, for example, [12, Chapter 1.9]. □

We refer to two prefixes as incomparable if neither one is a prefix of the other.

Definition 2.17 *Let* $e_1, e_2 \in \text{pref}(\text{Exec}_{\mathcal{S}})$. *Then* e_1 *and* e_2 *are* incomparable *if* $e_1 \not\sqsubseteq e_2$, *and* $e_2 \not\sqsubseteq e_1$.

Example 2.18 *Consider* $e_1 = t_0t_1$, $e_2 = t_0$, *and* $e_3 = t_1$. *Then* e_1 *and* e_3 *are incomparable, as are* e_2 *and* e_3. *But* $e_2 \sqsubseteq e_1$, *so* e_1 *and* e_2 *are not incomparable.*

We also use $e_1 \sqsubset e_2$ to denote that $e_1 \sqsubseteq e_2$ and $e_1 \neq e_2$.

Below, we define minimal elements and subsets of $\text{pref}(\text{Exec}_{\mathcal{S}})$ with respect to the prefix relation.

Definition 2.19 *Let* $A \subseteq \text{pref}(\text{Exec}_{\mathcal{S}})$. *An element* $e \in A$ *is* minimal *in* A *if, for all* $e' \in A$, *if* $e' \sqsubseteq e$ *then* $e' = e$.

Definition 2.20 *Let* $A \subseteq \text{pref}(\text{Exec}_{\mathcal{S}})$. *The set* $\min(A)$ *of minimal elements of* A *is defined by*

$$\min(A) = \{e \in A \mid \ e \text{ is minimal in } A \ \}.$$

Example 2.21 *Consider the PTS in Example 2.11. Let* $A = \{t_{01}, t_{01}t_{13}, t_{02}\}$. *Then* $\min(A) = \{t_{01}, t_{02}\}$.

Next, we present the concept of a basic cylinder set. Given a PTS $\mathcal{S}$, and an execution prefix e, the set of all executions of $\mathcal{S}$ that begin with e is a basic cylinder set.

Definition 2.22 *Let* $e \in \text{pref}(\text{Exec}_{\mathcal{S}})$. *Its* basic cylinder set $B^e_{\mathcal{S}}$ *is defined by*

$$B^e_{\mathcal{S}} = \{\, e' \in \text{Exec}_{\mathcal{S}} \mid e \sqsubseteq e' \,\}.$$

Now we prove that two basic cylinder sets are disjoint when their prefixes are incomparable.

Proposition 2.23 *Let* $\mathcal{S}$ *be a PTS, and* $e_1, e_2 \in \text{pref}(\text{Exec}_{\mathcal{S}})$. *If* e_1 *and* e_2 *are incomparable, then* $B^{e_1}_{\mathcal{S}}$ *and* $B^{e_2}_{\mathcal{S}}$ *are disjoint.*

Proof Let $e \in B^{e_1}_{\mathcal{S}}$, and let $e' \in B^{e_2}_{\mathcal{S}}$. Since e_1 and e_2 are incomparable and e_1 is a prefix of e, e_2 is not a prefix of e. Therefore, $e \notin B^{e_2}_{\mathcal{S}}$. Similarly, $e' \notin B^{e_1}_{\mathcal{S}}$. Therefore, $B^{e_1}_{\mathcal{S}} \cap B^{e_2}_{\mathcal{S}} = \emptyset$. □

2.2.2 Linear-Time Properties

In probabilistic model checking, we can express characteristics of the system that we want to examine as linear-time properties.

Definition 2.24 *Let AP be a set of atomic propositions. A* linear-time property *is a subset of* $(2^{AP})^\omega$.

Now, we define when a sequence of sets of atomic propositions satisfies a linear-time property.

Definition 2.25 *Let* $\sigma \in (2^{AP})^\omega$, *and let* ϕ *be a linear-time property. The* satisfaction relation, $\models$, *is defined by*

$$\sigma \models \phi \text{ iff } \sigma \in \phi.$$

As per Definition 2.10, the function $\text{label}_{\mathcal{S}}$ assigns to each state the set of atomic propositions that hold in the state. This function is extended to execution paths and their prefixes as follows.

Definition 2.26 *The function* $\text{trace}_{\mathcal{S}} : \text{Exec}_{\mathcal{S}} \to (2^{AP_{\mathcal{S}}})^\omega$ *is defined by*

$$\text{trace}_{\mathcal{S}}(t_1 t_2 \ldots) = \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t_1))\text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t_2)) \ldots$$

The function $\text{trace}_{\mathcal{S}} : \text{pref}(\text{Exec}_{\mathcal{S}}) \to (2^{AP_{\mathcal{S}}})^*$ *is defined by*

$$\text{trace}_{\mathcal{S}}(t_1 \ldots t_n) = \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t_1)) \ldots \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t_n))\text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t_n))$$

Example 2.27 *Consider the PTS* $\mathcal{S}$ *of Example 2.11.*

$$\begin{aligned} \text{trace}_{\mathcal{S}}(t_{02}{t_{22}}^\omega) &= \{p\}\{p,q\}^\omega \\ \text{trace}_{\mathcal{S}}(t_{01}t_{13}{t_{33}}^\omega) &= \{p\}\{p,q\}\{p\}^\omega \\ \text{trace}_{\mathcal{S}}(t_{01}t_{10}t_{02}{t_{22}}^\omega) &= \{p\}\{p,q\}\{p\}\{p,q\}^\omega \end{aligned}$$

Based on this notion, we define when an execution path of a PTS satisfies a linear-time property.

Definition 2.28 *Given a PTS* $\mathcal{S}$, $e \in \text{Exec}_{\mathcal{S}}$, *and a linear-time property* ϕ, *the satisfaction relation* $\models_{\mathcal{S}}$ *is defined by*

$$e \models_{\mathcal{S}} \phi \text{ iff } \text{trace}_{\mathcal{S}}(e) \models \phi.$$

Many linear-time properties can be expressed using linear temporal logic (LTL). We first define the syntax of LTL, then define its semantics. Our definitions are based on [2, Chapter 5].

Definition 2.29 *Let AP be a set of atomic propositions.* Linear temporal logic *(LTL) formulas over AP are formed by the following grammar (where $p \in AP$):*

$$\phi := \ \mathit{true} \mid p \mid \phi \wedge \phi \mid \neg\phi \mid \bigcirc\phi \mid \phi \,\mathcal{U}\, \phi$$

Brackets, though not included in the simplified grammar above, can also be used in LTL formulas to specify precedence.

For the following definition, given $\sigma \in (2^{AP})^\omega$, we use $\sigma[i...]$ to represent the suffix of σ beginning with the i^{th} set in σ. For instance, if $\sigma = P_0P_1{P_2}^\omega$, then $\sigma[0...] = \sigma$, $\sigma[1...] = P_1{P_2}^\omega$, and $\sigma[2...] = {P_2}^\omega$. As for executions, we use $\sigma[i]$ to represent the prefix of σ containing the first i sets, so in our example $\sigma[1] = P_0$, $\sigma[2] = P_0P_1$, etc.

Definition 2.30 *Let $\sigma \in (2^{AP})^\omega$. The satisfaction relation $\models$ is defined as follows:*

	$\sigma \models \mathit{true}$		
	$\sigma \models p$	*iff*	$p \in \sigma[1]$
(and)	$\sigma \models \phi_1 \wedge \phi_2$	*iff*	$\sigma \models \phi_1$ *and* $\sigma \models \phi_2$
(until)	$\sigma \models \phi_1 \,\mathcal{U}\, \phi_2$	*iff*	$\exists j \geq 0$ *such that* $\sigma[j...] \models \phi_2$ *and* $\forall 0 \leq i < j : \sigma[i...] \models \phi_1$
(not)	$\sigma \models \neg\phi$	*iff*	$\sigma \not\models \phi$
(next)	$\sigma \models \bigcirc\phi$	*iff*	$\sigma[1...] \models \phi$

The syntax and semantics above can be used to define further operators. The following are used in this thesis:

	false	$= \neg$ true
(eventually)	$\Diamond\phi$	$=$ true $\mathcal{U}\phi$
(always)	$\Box\phi$	$= \neg\Diamond\neg\phi$
(or)	$\phi_1 \vee \phi_2$	$= \neg(\neg\phi_1 \wedge \neg\phi_2)$
(release)	$\phi_1 \,\mathcal{R}\, \phi_2$	$= \neg(\neg\phi_1 \,\mathcal{U}\, \neg\phi_2)$
(weak until)	$\phi_1 \,\mathcal{W}\, \phi_2$	$= (\phi_1 \,\mathcal{U}\, \phi_2) \vee \Box\phi_1$

In [30] Zhang and Van Breugel show how to calculate progress for a category of linear-time properties called invariants, defined below. Some of the results in this thesis also involve invariants.

Definition 2.31 *An* invariant *is a property that can be expressed as $\Box p$ for some atomic proposition p.*

The definition above is a simplified version of [2, Definition 3.20], in which the condition that must hold in each state is a propositional logic formula rather than an atomic proposition. Since whether or not a propositional logic formula is satisfied in each state can be converted to an atomic proposition, the definitions are equivalent for our purposes.

2.3 A Progress Measure

In this section, we review some of Zhang and Van Breugel's key notions and results on the progress measure, from [30].

Given a PTS $\mathcal{S}$, we generate a σ-algebra over $\text{Exec}_{\mathcal{S}}$ from the basic cylinder sets of $\mathcal{S}$. We then define a measure on these cylinder sets to create a measure space, which forms the basis of the progress measure.

Definition 2.32 *Let $\mathcal{S}$ be a PTS. The σ-algebra over $\text{Exec}_{\mathcal{S}}$, $\Sigma_{\mathcal{S}}$, is defined as the smallest σ-algebra containing $\mathcal{B}_{\mathcal{S}} = \{B^e_{\mathcal{S}} \mid e \in \text{pref}(\text{Exec}_{\mathcal{S}})\}$.*

Definition 2.33 *Given a PTS $\mathcal{S}$, the measure $\mu_{\mathcal{S}}$ is defined on a basic cylinder set $B^{t_1 \cdots t_n}_{\mathcal{S}}$ by*

$$\mu_{\mathcal{S}}(B^{t_1 \cdots t_n}_{\mathcal{S}}) = \prod_{1 \leq i \leq n} \text{prob}_{\mathcal{S}}(t_i).$$

The function $\mu_{\mathcal{S}}$ is a probability measure on the set of basic cylinder sets of $\mathcal{S}$. As discussed in [30], this measure can be extended to a probability measure on the σ-algebra $\Sigma_{\mathcal{S}}$. Thus the triple $\langle \text{Exec}_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \mu_{\mathcal{S}} \rangle$ is a probability space, as per Definition 2.8.

The verification effort of the probabilistic model checker is represented by its search of the PTS. The search is captured by the set of transitions that have been explored during the search.

Definition 2.34 *A* search *of a PTS $\mathcal{S}$ is a finite subset of $T_{\mathcal{S}}$.*

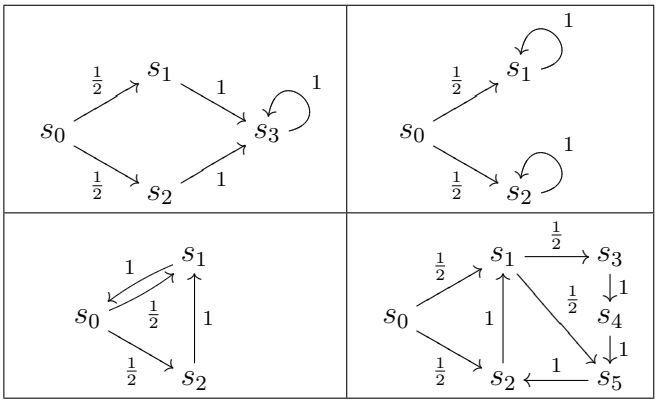
Example 2.35 *Consider the PTS of Example 2.11. The sets $\emptyset$, $\{t_{01}\}$, $\{t_{02}\}$, $\{t_{01}, t_{02}\}$ and $\{t_{01}, t_{02}, t_{10}, t_{13}, t_{22}, t_{33}\}$ are examples of searches.*

A PTS is said to extend a search if the transitions of the search are part of the PTS. We will use this notion in the definition of the progress measure.

Definition 2.36 *The PTS $\mathcal{S}'$* extends *the search T of the PTS $\mathcal{S}$ if for all $t \in T$,*

- $t \in T_{\mathcal{S}'}$,
- $\text{source}_{\mathcal{S}'}(t) = \text{source}_{\mathcal{S}}(t)$,
- $\text{target}_{\mathcal{S}'}(t) = \text{target}_{\mathcal{S}}(t)$,
- $\text{prob}_{\mathcal{S}'}(t) = \text{prob}_{\mathcal{S}}(t)$,
- $\text{label}_{\mathcal{S}'}(\text{source}_{\mathcal{S}'}(t)) = \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t))$,
- $\text{label}_{\mathcal{S}'}(\text{target}_{\mathcal{S}'}(t)) = \text{label}_{\mathcal{S}}(\text{target}_{\mathcal{S}}(t))$, *and*
- $s_{0\mathcal{S}} = s_{0\mathcal{S}'}$.

Example 2.37 *Consider the PTS of Example 2.11 and the search* $\{t_{01}, t_{02}\}$. *The following extend the search:*



Note that for any search of a PTS $\mathcal{S}$, $\mathcal{S}$ itself extends the search.

PTSs that extend a particular search give rise to the same set of execution paths if we restrict ourselves to those execution paths that only consist of transitions explored during the search.

Proposition 2.38 *If the PTS* $\mathcal{S}'$ *extends the search* T *of the PTS* $\mathcal{S}$, *then*

(a) $T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}}) = T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}'})$ *and*

(b) $T^\omega \cap \text{Exec}_{\mathcal{S}} = T^\omega \cap \text{Exec}_{\mathcal{S}'}$.

Proof We only prove part (a). Part (b) can be proved similarly. Assume that $t_1 \dots t_n \in T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}})$. Since $\mathcal{S}'$ extends T, s_0 is the initial state of both $\mathcal{S}$ and $\mathcal{S}'$. Since $t_1 \dots t_n \in T^*$ and $\mathcal{S}'$ extends T, we have that for all $1 \leq i \leq n$,

(1) $\text{source}_{\mathcal{S}}(t_i) = \text{source}_{\mathcal{S}'}(t_i)$ and

(2) $\text{target}_{\mathcal{S}}(t_i) = \text{target}_{\mathcal{S}'}(t_i)$.

Since $t_1 \ldots t_n \in \text{pref}(\text{Exec}_{\mathcal{S}})$, we have that

(3) $\text{source}_{\mathcal{S}}(t_{i+1}) = \text{target}_{\mathcal{S}}(t_i)$ for all $1 \leq i < n$ and

(4) $\text{source}_{\mathcal{S}}(t_1) = s_0$.

For all $1 \leq i < n$,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(t_{i+1}) &= \text{source}_{\mathcal{S}}(t_{i+1}) \ \ [(1)] \\ &= \text{target}_{\mathcal{S}}(t_i) \ \ [(3)] \\ &= \text{target}_{\mathcal{S}'}(t_i) \ \ [(2)]. \end{aligned}$$

Furthermore,

$$\begin{aligned} \text{source}_{\mathcal{S}'}(t_1) &= \text{source}_{\mathcal{S}}(t_1) \ \ [(1)] \\ &= s_0 \ \ [(4)]. \end{aligned}$$

Hence, $t_1 \ldots t_n \in T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}'})$. □

PTSs that extend a particular search also assign the same measure to basic cylinder sets of prefixes of execution paths only consisting of transitions explored during the search.

Proposition 2.39 *If the PTS $\mathcal{S}'$ extends the search T of the PTS $\mathcal{S}$, then for all $e \in T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}})$:*

$$\mu_{\mathcal{S}}(B^e_{\mathcal{S}}) = \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'})$$

Proof

Let $e = t_1, ..., t_n \in T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}})$.

$$\begin{aligned} \mu_{\mathcal{S}}(B^e_{\mathcal{S}}) &= \prod_{1 \leq i \leq n} \text{prob}_{\mathcal{S}}(t_i) \\ &= \prod_{1 \leq i \leq n} \text{prob}_{\mathcal{S}'}(t_i) \ [\mathcal{S}' \text{ extends } T \text{ of } \mathcal{S}] \\ &= \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'}) \ [\text{Proposition 2.38 }] \end{aligned}$$

□

For PTSs that extend a particular search, those execution paths that only consist of transitions explored by the search satisfy the same linear-time properties.

Proposition 2.40 *Let ϕ be a linear-time property. If the PTS $\mathcal{S}'$ extends the search T of the PTS $\mathcal{S}$, then for all $e \in T^\omega \cap \mathrm{Exec}_{\mathcal{S}}$, $e \models_{\mathcal{S}} \phi$ iff $e \models_{\mathcal{S}'} \phi$.*

Proof Since $\mathcal{S}'$ extends T of $\mathcal{S}$, $\mathrm{trace}_{\mathcal{S}}(e) = \mathrm{trace}_{\mathcal{S}'}(e)$ for all $e \in T^\omega \cap \mathrm{Exec}_{\mathcal{S}}$. $\square$

Next, we define a union of basic cylinder sets that satisfy a property, and whose prefixes are part of a search. This union is used in the definition of the progress measure.

Definition 2.41 *Let the PTS $\mathcal{S}'$ extend the search T of PTS $\mathcal{S}$ and let ϕ be a linear-time property. The set $\mathcal{B}^{\phi}_{\mathcal{S}'}(T)$ is defined by*

$$\mathcal{B}^{\phi}_{\mathcal{S}'}(T) = \bigcup \{\, B^{e}_{\mathcal{S}'} \mid e \in T^* \wedge \forall e' \in B^{e}_{\mathcal{S}'} : e' \models_{\mathcal{S}'} \phi \,\}.$$

As shown below, the union $\mathcal{B}^{\phi}_{\mathcal{S}'}(T)$ is a subset in $\Sigma_{\mathcal{S}'}$. Therefore, it can be measured by $\mu_{\mathcal{S}'}$.

Proposition 2.42 *Let ϕ be a linear time property, and let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then $\mathcal{B}^{\phi}_{\mathcal{S}'}(T) \in \Sigma_{\mathcal{S}'}$.*

Proof For each $e \in T^*$, we have that $B^{e}_{\mathcal{S}'} \in \mathcal{B}_{\mathcal{S}'}$, and, hence, $B^{e}_{\mathcal{S}'} \in \Sigma_{\mathcal{S}'}$. The set $\{B^{e}_{\mathcal{S}'} \in \mathcal{B}_{\mathcal{S}'} \mid e \in T^* \wedge \forall e' \in B^{e}_{\mathcal{S}'} : e' \models_{\mathcal{S}'} \phi\}$ is countable since the set T^* is countable (Proposition 2.5). Since $\Sigma_{\mathcal{S}'}$ is a σ-algebra and, hence, closed under countable unions, the desired result follows. $\square$

The set $\mathcal{B}^{\phi}_{\mathcal{S}'}(T)$ is the union of those basic cylinder sets $B^{e}_{\mathcal{S}'}$ the execution paths of which satisfy the linear-time property ϕ. Hence, $B^{e}_{\mathcal{S}'}$ does not contain any execution paths violating ϕ. Since the set $\mathcal{B}^{\phi}_{\mathcal{S}'}(T)$ is measurable, the measure $\mu_{\mathcal{S}'}$ assigns it a real number in the unit interval. This number represents the "size" of the basic cylinder sets that do not contain any violations of ϕ. This number captures the amount of progress of the search T verifying ϕ, *provided that* the PTS under consideration is $\mathcal{S}'$. However, we have no knowledge of the transitions other than the search. Therefore, we consider all extensions $\mathcal{S}'$ of T and consider the worst case in terms of progress.

Using these concepts, we introduce the notion of a progress measure. Given a search of a PTS and a linear-time property, it captures the amount of progress the search of the probabilistic model checker has made towards verifying the linear-time property.

Definition 2.43 *The* progress *of the search T of the PTS $\mathcal{S}$ of the linear-time property ϕ is defined by*

$$\mathrm{prog}_{\mathcal{S}}(T,\phi) = \inf\left\{ \mu_{\mathcal{S}'}\left(\mathcal{B}^{\phi}_{\mathcal{S}'}(T)\right) \mid \mathcal{S}' \text{ extends } T \text{ of } \mathcal{S}\right\}.$$

Example 2.44 *Consider the PTS $\mathcal{S}$ of Example 2.11 and the linear temporal logic formulas $\Box p$, $\Diamond p$, $\Diamond q$ and $\bigcirc q$. In the table below, we present the progress of these properties for a number of searches.*

search	$\Box p$	$\Diamond p$	$\Diamond q$	$\bigcirc q$
$\emptyset$	0	1	0	0
$\{t_{01}\}$	0	1	$\frac{1}{2}$	$\frac{1}{2}$
$\{t_{02}\}$	0	1	$\frac{1}{2}$	$\frac{1}{2}$
$\{t_{01}, t_{02}\}$	0	1	1	1
$\{t_{01}, t_{13}, t_{33}\}$	$\frac{1}{4}$	1	$\frac{1}{2}$	$\frac{1}{2}$
$\{t_{01}, t_{10}, t_{13}, t_{33}\}$	$\frac{1}{3}$	1	$\frac{1}{2}$	$\frac{1}{2}$

In [30, Theorem 1], Zhang and Van Breugel prove the following key property of their progress measure. They show that it is a lower bound for the probability that the linear-time property holds.

Theorem 2.45 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a linear-time property. Then*

$$\mathrm{prog}_{\mathcal{S}}(T,\phi) \leq \mu_{\mathcal{S}}(\{\, e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi \,\}).$$

Proof According to [25, Corollary 2.4], $\{e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}$ is measurable. From the definition of prog, Proposition 2.9, and the fact that $\mathcal{S}$ extends T of $\mathcal{S}$, we can conclude that it suffices to show that $\mathcal{B}^{\phi}_{\mathcal{S}}(T)$ is a subset of $\{e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}$. Let $e \in T^*$ and assume that $\forall e' \in B^{\mathcal{S}}_e : e' \models_{\mathcal{S}} \phi$. It suffices to show that $B^e_{\mathcal{S}}$ is a subset of $\{e \in \mathrm{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}$. Let $e' \in B^e_{\mathcal{S}}$. Then $e' \in \mathrm{Exec}_{\mathcal{S}}$ and $e' \models_{\mathcal{S}} \phi$. $\Box$

The setting in this thesis is slightly different from the one in [30]. In our definition of a PTS, we require that the sum of probabilities of outward transitions from each state must equal 1. Thus, we assume that PTSs do not have final states. This assumption can be made without loss of any generality: simply add a self loop with probability one to each final state.

2.4 Transient States

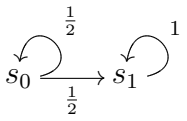
Now that we have defined a σ-algebra over the execution paths of a PTS, we use its measure to define when a state of a PTS is transient.

Note that we use $\text{Exec}^s_{\mathcal{S}}$ to denote the set of execution paths of $\mathcal{S}$ which begin in state s. Furthermore, note that the set $\text{pref}(\text{Exec}^s_{\mathcal{S}})$ is countable, as per Proposition 2.5. Thus $\bigcup\{B^e_{\mathcal{S}} \mid e \in \text{pref}(\text{Exec}^s_{\mathcal{S}}) \setminus \{\epsilon\} \wedge \text{target}_{\mathcal{S}}(e) = s\}$ is a countable union of measurable sets, so it is measurable.

Definition 2.46 *Let $\mathcal{S}$ be a PTS and let $s \in S_{\mathcal{S}}$. Then s is* transient *if*

$$\mu_{\mathcal{S}}\left(\bigcup\{\, B^e_{\mathcal{S}} \mid e \in \text{pref}(\text{Exec}^s_{\mathcal{S}}) \setminus \{\epsilon\} \wedge \text{target}_{\mathcal{S}}(e) = s \,\}\right) < 1.$$

Example 2.47 *Consider the probabilistic transition system $\mathcal{S}$ depicted below:*



The state s_0 is transient, and s_1 is not transient.

Intuitively, a state is transient when, if an execution visits that state, it is not guaranteed to visit it again. Below, we define two particular circumstances in which we can prove that a state is transient.

Proposition 2.48 *Let $\mathcal{S}$ be a PTS, and let $s \in S_{\mathcal{S}}$. If there exists $e \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$ such that for all $t \in e' \in B^e_{\mathcal{S}}$: $\text{target}_{\mathcal{S}}(t) \neq s$, then s is transient.*

Proof Let $Z = \bigcup\{\, B^e_{\mathcal{S}} \mid e \in \text{pref}(\text{Exec}^s_{\mathcal{S}}) \setminus \{\epsilon\} \wedge \text{target}_{\mathcal{S}}(e) = s \,\}$. Let $e \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$ such that for all $t \in e' \in B^e_{\mathcal{S}}$, $\text{target}_{\mathcal{S}}(t) \neq s$.

Since no execution in $B^e_{\mathcal{S}}$ contains any transition with target s, $B^e_{\mathcal{S}} \cap Z = \emptyset$. Since all transitions of $\mathcal{S}$ have probability greater than 0, $\mu_{\mathcal{S}}(B^e_{\mathcal{S}}) > 0$. Therefore:

$$\begin{aligned}
&\mu_{\mathcal{S}}(Z) \\
&\qquad = 1 - \mu_{\mathcal{S}}(\text{Exec}^s_{\mathcal{S}} \setminus Z) \\
&\qquad \leq 1 - \mu_{\mathcal{S}}(B^e_{\mathcal{S}}) \\
&\qquad < 1
\end{aligned}$$

□

Proposition 2.49 *Let $\mathcal{S}$ be a PTS. Let $r \in S_{\mathcal{S}}$ have a probability-one self-loop, and let $s \in S_{\mathcal{S}} \setminus \{r\}$. If r is reachable from s, then s is transient.*

Proof Suppose r is reachable from s. Let $e = t_1, ..., t_n \in \text{pref}(\text{Exec}^s_{\mathcal{S}})$ be a shortest execution path with $\text{target}_{\mathcal{S}}(e) = r$. Then for all $1 \leq i \leq n$, $\text{target}_{\mathcal{S}}(t_i) \neq s$.

Since $\text{target}_{\mathcal{S}}(e) = r$ and r has a probability-one self-loop, for all $t \in e' \in B^e_{\mathcal{S}}$, $\text{target}_{\mathcal{S}}(t) \neq s$. So by Proposition 2.48, s is transient. □

Sets of execution paths that remain in a finite set of transient states forever have measure zero, a fact that we use in later proofs.

Proposition 2.50 *Let $\mathcal{S}$ be a PTS, let $S_{trans} \subseteq S_{\mathcal{S}}$ be a finite set of transient states, and $T_{trans} = \{t \in T_{\mathcal{S}} \mid \text{source}_{\mathcal{S}}(t) \in S_{trans} \wedge \text{target}_{\mathcal{S}}(t) \in S_{trans}\}$. For all $s \in S_{\mathcal{S}}$,*

$$\mu_{\mathcal{S}}(\text{Exec}^s_{\mathcal{S}} \cap {T_{trans}}^{\omega}) = 0.$$

Proof See, for instance, [1, Chapter 7.3]. □

3 Negation and Violations

In this section, we consider the relationship between making progress towards verifying a linear-time property and finding a violation of its negation. First, we formalize that a search has not found a violation of a linear-time property.

Definition 3.1 *The search T of the PTS $\mathcal{S}$ has not found a violation of the linear-time property ϕ if there exists a PTS $\mathcal{S}'$ which extends T of $\mathcal{S}$ such that $e \models_{\mathcal{S}'} \phi$ for all $e \in \mathrm{Exec}_{\mathcal{S}'}$.*

Hence, a search has found a violation if no matter how we extend the search, there always exists an execution path that does not satisfy the linear-time property.

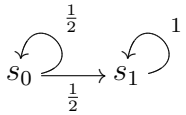
This definition is slightly stronger than the one given in [30, Definition 7]. All results of [30] remain valid for this stronger version. Next, we prove that if a search has made some progress towards verifying a linear-time property $\neg\phi$, then that search has also found a violation of ϕ.

Proposition 3.2 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a linear-time property. If $\mathrm{prog}_{\mathcal{S}}(T, \neg\phi) > 0$ then T has found a violation of ϕ.*

Proof By the definition of prog, $\mu_{\mathcal{S}'}(\mathcal{B}^{\neg\phi}_{\mathcal{S}'}(T)) > 0$ for each PTS $\mathcal{S}'$ which extends T of $\mathcal{S}$. Hence, $\mathcal{B}^{\neg\phi}_{\mathcal{S}'}(T) \neq \emptyset$. Therefore, there exists $e \in T^*$ such that $B^e_{\mathcal{S}'} \neq \emptyset$ and $\forall e' \in B^e_{\mathcal{S}'} : e' \models_{\mathcal{S}'} \neg\phi$. Hence, $e' \not\models \phi$ and $e' \in \mathrm{Exec}_{\mathcal{S}'}$. Therefore, T has found a violation of ϕ. □

The reverse implication does not hold in general, as shown in the following example.

Example 3.3 *Consider the PTS*



Assume that the state s_0 satisfies the atomic proposition p and the state s_1 does not. Consider the linear-time property $\Box p$ and the search $\{t_{00}\}$. Note that ${t_{00}}^\omega \not\models \neg\Box p$ and, hence, $\{t_{00}\}$ has found a violation of $\neg\Box p$. Also note that $\mathrm{prog}_{\mathcal{S}}(\{t_{00}\}, \Box p) = 0$.

We conjecture that the reverse implication does hold for safety properties (see, for example, [2, Definition 3.22] for a formal definition of safety property). However, so far we have only been able to prove it for invariants.

Proposition 3.4 *If the search T of the PTS $\mathcal{S}$ has found a violation of the invariant ϕ, then* $\text{prog}_{\mathcal{S}}(T, \neg\phi) > 0$.

Proof For every PTS $\mathcal{S}'$ that extends T, $e \not\models_{\mathcal{S}'} \Box p$ for some $e \in \text{Exec}_{\mathcal{S}'}$. Since it is possible to extend the search using only states that satisfy p, some execution prefix in T^* must reach a state that does not satisfy p. Since all extensions of the search share T, this prefix must exist in every extension. So, assume e contains this execution prefix. Hence, $e = e_f t e_\ell$ for some $e_f \in T^* \cap \text{pref}(\text{Exec}_{\mathcal{S}'})$ and t such that $p \notin \text{label}_{\mathcal{S}'}(\text{source}_{\mathcal{S}'}(t))$. Therefore, for all $e' \in B^{e_f}_{\mathcal{S}'}$ we have that $e' \models_{\mathcal{S}'} \neg\Box p$ and $B^{e_f}_{\mathcal{S}'} \neq \emptyset$. Hence, $\mu_{\mathcal{S}'}(B^{e_f}_{\mathcal{S}'}) > 0$ and, therefore, $\text{prog}_{\mathcal{S}}(T, \neg\Box p) > 0$. □

By finding the progress for a property and its negation, we can put upper and lower bounds on the measure of executions that satisfy that property.

Proposition 3.5 *Let T be a search of the PTS $\mathcal{S}$, and let ϕ be a linear-time property. Then*

$$\text{prog}_{\mathcal{S}}(T, \phi) \leq \mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}) \leq 1 - \text{prog}_{\mathcal{S}}(T, \neg\phi)$$

Proof

$$\begin{aligned}
\text{prog}_{\mathcal{S}}(T, \neg\phi) &\leq \mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \neg\phi\}) \text{ [Theorem 2.45]} \\
1 - \mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \neg\phi\}) &\leq 1 - \text{prog}_{\mathcal{S}}(T, \neg\phi) \\
\mu_{\mathcal{S}}(\text{Exec}_{\mathcal{S}} \setminus \{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \neg\phi\}) &\leq 1 - \text{prog}_{\mathcal{S}}(T, \neg\phi) \\
\mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}) &\leq 1 - \text{prog}_{\mathcal{S}}(T, \neg\phi) \\
\text{prog}_{\mathcal{S}}(T, \phi) \leq \mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \phi\}) &\leq 1 - \text{prog}_{\mathcal{S}}(T, \neg\phi) \text{ [Theorem 2.45]}
\end{aligned}$$

□

4 Progress for a Unipolar Fragment of LTL

In this chapter, we introduce a unipolar fragment of linear temporal logic (LTL), and show how to measure progress for it.

Section 4.1 defines the unipolar fragment of LTL. This fragment is composed of formulas that can be written in positive normal form without both an atomic proposition and the negation of the same proposition. It includes a wide range of properties.

In Section 4.2 we show how to compute the progress for any formula in this fragment. For a unipolar LTL formula ϕ, we prove that the progress measure is equal to the measure of executions that satisfy ϕ in a modified system. Others have developed algorithms to compute this measure, which are exponential in the size of ϕ and polynomial in the size of the searched space.

4.1 Defining Unipolar LTL

In this section, we define the subset of LTL formulas that makes up unipolar LTL. Our definition of unipolar LTL begins with the definition of positive normal form (PNF) for LTL, which we outline below.

Definition 4.1 *The logic PNF is defined by*

$$\phi ::= \text{true} \mid \text{false} \mid p \mid \neg p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bigcirc \phi \mid \phi \,\mathcal{U}\, \phi \mid \phi \,\mathcal{R}\, \phi$$

where $p \in AP$.

For each LTL formula, there exists an equivalent PNF formula (see, for example, [2, Section 5.1.5]). Next, we restrict our attention to a particular class of PNF formulas. To define this class, we define the positive and negative occurrences of atomic propositions in a PNF formula as follows.

Definition 4.2 *The function* $\mathrm{pos} : \mathrm{PNF} \to 2^{AP}$ *is defined by*

$$\begin{aligned}
\mathrm{pos}(\mathrm{true}) &= \emptyset \\
\mathrm{pos}(\mathrm{false}) &= \emptyset \\
\mathrm{pos}(p) &= \{p\} \\
\mathrm{pos}(\neg p) &= \emptyset \\
\mathrm{pos}(\phi_1 \wedge \phi_2) &= \mathrm{pos}(\phi_1) \cup \mathrm{pos}(\phi_2) \\
\mathrm{pos}(\phi_1 \vee \phi_2) &= \mathrm{pos}(\phi_1) \cup \mathrm{pos}(\phi_2) \\
\mathrm{pos}(\bigcirc \phi) &= \mathrm{pos}(\phi) \\
\mathrm{pos}(\phi_1 \,\mathcal{U}\, \phi_2) &= \mathrm{pos}(\phi_1) \cup \mathrm{pos}(\phi_2) \\
\mathrm{pos}(\phi_1 \,\mathcal{R}\, \phi_2) &= \mathrm{pos}(\phi_1) \cup \mathrm{pos}(\phi_2)
\end{aligned}$$

The function $\mathrm{neg} : \mathrm{PNF} \to 2^{AP}$ *is defined by*

$$\begin{aligned}
\mathrm{neg}(\mathrm{true}) &= \emptyset \\
\mathrm{neg}(\mathrm{false}) &= \emptyset \\
\mathrm{neg}(p) &= \emptyset \\
\mathrm{neg}(\neg p) &= \{p\} \\
\mathrm{neg}(\phi_1 \wedge \phi_2) &= \mathrm{neg}(\phi_1) \cup \mathrm{neg}(\phi_2) \\
\mathrm{neg}(\phi_1 \vee \phi_2) &= \mathrm{neg}(\phi_1) \cup \mathrm{neg}(\phi_2) \\
\mathrm{neg}(\bigcirc \phi) &= \mathrm{neg}(\phi) \\
\mathrm{neg}(\phi_1 \,\mathcal{U}\, \phi_2) &= \mathrm{neg}(\phi_1) \cup \mathrm{neg}(\phi_2) \\
\mathrm{neg}(\phi_1 \,\mathcal{R}\, \phi_2) &= \mathrm{neg}(\phi_1) \cup \mathrm{neg}(\phi_2)
\end{aligned}$$

Example 4.3 *Let* $\phi = (p \vee \neg q) \,\mathcal{U}\, r$. *Then* $\mathrm{pos}(\phi) = \{p, r\}$, *and* $\mathrm{neg}(\phi) = \{q\}$.

We restrict ourselves to PNF formulas in which an atomic proposition cannot occur both positively and negatively. We call these PNF formulas unipolar.

Definition 4.4 *A PNF formula* ϕ *is unipolar if* $\mathrm{pos}(\phi) \cap \mathrm{neg}(\phi) = \emptyset$.

Example 4.5 *Let* $\phi_1 = p \,\mathcal{U}\, (q \wedge \neg r)$. *Then* $\mathrm{pos}(\phi_1) \cap \mathrm{neg}(\phi_1) = \{p, q\} \cap \{r\} = \emptyset$. *So* ϕ_1 *is unipolar.*

Let $\phi_2 = p \,\mathcal{U}\, (q \wedge \neg p)$. *In this case,* $\mathrm{pos}(\phi_2) \cap \mathrm{neg}(\phi_2) = \{p, q\} \cap \{p\} = \{p\} \neq \emptyset$. *Thus,* ϕ_2 *is not unipolar.*

If a PNF formula ϕ is unipolar, then $\mathrm{neg}(\phi) \subseteq AP \setminus \mathrm{pos}(\phi)$.

A property of unipolar PNF formulas that is key for our development is presented next. Suppose you have a property ϕ, a sequence of label sets σ, and a set of labels L such that $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$. We prove that if σ is extended using only labels in L, and this extension satisfies ϕ, then any extension of σ satisfies ϕ.

Proposition 4.6 *For all unipolar PNF formulas ϕ, $\sigma \in (2^{AP})^*$, and L such that* $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$, $\sigma L^\omega \models \phi$ *iff* $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$.

Proof We prove two implications. Let ϕ be a unipolar PNF formula, $\sigma \in (2^{AP})^*$ and $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$. Assume that $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$. Since $L^\omega \in (2^{AP})^\omega$, we can immediately conclude that $\sigma L^\omega \models \phi$.

The other implication is proved by structural induction on ϕ. Let $\sigma \in (2^{AP})^*$ and $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$. We distinguish the following cases.

- In case $\phi = \mathrm{true}$, clearly $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$ and, hence, the property is satisfied.
- In case $\phi = \mathrm{false}$, obviously $\sigma L^\omega \models \phi$ is not satisfied and, therefore, the implication holds.
- Let $\phi = p$. Assume that $\sigma L^\omega \models \phi$. Since $L \subseteq AP \setminus \mathrm{pos}(\phi)$, we have that $p \notin L$. Therefore, $|\sigma| > 0$ and $p \in \sigma[0]$ and, hence, $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$.
- Let $\phi = \neg p$. Assume that $\sigma L^\omega \models \phi$. Since $\mathrm{neg}(\phi) \subseteq L$, we have that $p \in L$. Therefore, $|\sigma| > 0$ and $p \notin \sigma[0]$ and, hence, $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$.
- Let $\phi = \phi_1 \wedge \phi_2$. Assume that $\sigma L^\omega \models \phi$. Then $\sigma L^\omega \models \phi_1$ and $\sigma L^\omega \models \phi_2$. Since $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$, we have that $\mathrm{neg}(\phi_1) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi_1)$ and $\mathrm{neg}(\phi_2) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi_2)$. By induction, $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi_1$ and $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi_2$. Hence, $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$.
- The case $\phi = \phi_1 \vee \phi_2$ is similar to the previous case.
- For $\bigcirc\phi$ we distinguish the following two cases. Assume $|\sigma| = 0$. Suppose $\sigma L^\omega \models \bigcirc\phi$. Then $L^\omega[1\ldots] = L^\omega \models \phi$. By induction, $\forall \rho \in (2^{AP})^\omega : \rho \models \phi$. Hence, $\forall \rho \in (2^{AP})^\omega : \rho \models \bigcirc\phi$.

  Assume $|\sigma| \geq 1$. Suppose $\sigma L^\omega \models \bigcirc\phi$. Then $(\sigma L^\omega)[1\ldots] = \sigma[1\ldots]L^\omega \models \phi$. By induction, $\forall \rho \in (2^{AP})^\omega : \sigma[1\ldots]\rho \models \phi$. Since $\sigma[1\ldots]\rho = (\sigma\rho)[1\ldots]$, we have that $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \bigcirc\phi$.
- Let $\phi = \phi_1 \,\mathcal{U}\, \phi_2$. Since $\mathrm{neg}(\phi) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi)$, we have that $\mathrm{neg}(\phi_1) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi_1)$ and $\mathrm{neg}(\phi_2) \subseteq L \subseteq AP \setminus \mathrm{pos}(\phi_2)$. Hence, we can apply the induction hypothesis to ϕ_1 and ϕ_2

  Assume that $\sigma L^\omega \models \phi$. Then there exists some $j \geq 0$ such that

  (a) $(\sigma L^\omega)[i\ldots] \models \phi_1$ for all $0 \leq i < j$ and
  (b) $(\sigma L^\omega)[j\ldots] \models \phi_2$.

  We distinguish two cases. Suppose $j < |\sigma|$. From (a) we can conclude that for all $0 \leq i < j$, $(\sigma L^\omega)[i\ldots] = \sigma[i\ldots]L^\omega \models \phi_1$. By induction, $\forall \rho \in (2^{AP})^\omega$:

$\sigma[i\ldots]\rho \models \phi_1$. Since $\sigma[i\ldots]\rho = (\sigma\rho)[i\ldots]$, we have that $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[i\ldots] \models \phi_1$. From (b) we can deduce that $(\sigma L^\omega)[j\ldots] = \sigma[j\ldots]L^\omega \models \phi_2$. By induction, $\forall\rho \in (2^{AP})^\omega : \sigma[j\ldots]\rho \models \phi_2$. Since $\sigma[j\ldots]\rho = (\sigma\rho)[j\ldots]$, we have that $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[j\ldots] \models \phi_2$. Combining the above, we get $\forall\rho \in (2^{AP})^\omega : \sigma\rho \models \phi_1 \,\mathcal{U}\, \phi_2$.

Suppose $j \geq |\sigma|$. For $0 \leq i < |\sigma|$, the argument for (a) is the same as above. For $|\sigma| \leq i < j$, (a) simply says that $L^\omega \models \phi_1$, which, by induction, implies that $\forall\rho \in (2^{AP})^\omega : \rho \models \phi_1$. Hence, $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[i\ldots] \models \phi_1$ for all $0 \leq i < j$. In this case, (b) means $L^\omega \models \phi_2$, which, by induction, implies that $\forall\rho \in (2^{AP})^\omega : \rho \models \phi_2$. Hence, $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[j\ldots] \models \phi_2$. Combining the above, we obtain that $\forall\rho \in (2^{AP})^\omega : \sigma\rho \models \phi_1 \,\mathcal{U}\, \phi_2$.

- Finally, we consider $\phi_1 \mathcal{R} \phi_2$. According to [2, page 256], $\phi_1 \mathcal{R} \phi_2 \equiv \neg(\neg\phi_1 \mathcal{U} \neg\phi_2)$ and $\neg(\phi_1 \,\mathcal{U}\, \phi_2) \equiv (\neg\phi_2)\,\mathcal{W}\,(\neg\phi_1 \wedge \neg\phi_2)$. According to [2, page 252], $\phi_1 \,\mathcal{W}\, \phi_2 \equiv (\phi_1 \,\mathcal{U}\, \phi_2) \vee \Box\phi_1$. Hence, we can derive that $\phi_1 \,\mathcal{R}\, \phi_2 \equiv (\phi_2 \,\mathcal{U}\, (\phi_1 \wedge \phi_2)) \vee \Box\phi_2$. Therefore, proving that the property is satisfied by $\Box\phi$, combined with the proofs for $\wedge$, $\vee$ and $\mathcal{U}$ above, suffices as proof for $\phi_1 \,\mathcal{R}\, \phi_2$.

  Thus, we consider $\Box\phi$. Assume that $\mathrm{neg}(\Box\phi) = \mathrm{neg}(\phi) \subseteq L \subseteq AP \backslash \mathrm{pos}(\Box\phi) = AP \setminus \mathrm{pos}(\phi)$. Suppose that $\sigma L^\omega \models \Box\phi$. Then $(\sigma L^\omega)[j\ldots] \models \phi$ for all $j \geq 0$. We distinguish two cases. For all $0 \leq j < |\sigma|$, we have that $(\sigma L^\omega)[j\ldots] = \sigma[j\ldots]L^\omega \models \phi$. By induction, $\forall\rho \in (2^{AP})^\omega : \sigma[j\ldots]\rho \models \phi$ and, hence, $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[j\ldots] \models \phi$.

  For all $j \geq |\sigma|$, we have that $(\sigma L^\omega)[j\ldots] = L^\omega \models \phi$. By induction, $\forall\rho \in (2^{AP})^\omega : \rho \models \phi$ and, therefore, $\forall\rho \in (2^{AP})^\omega : (\sigma\rho)[j\ldots] \models \phi$. Combining the above, we get $\forall\rho \in (2^{AP})^\omega : \sigma\rho \models \Box\phi$. □

Corollary 4.7 *For all unipolar PNF formulas ϕ and $\sigma \in (2^{AP})^*$, $\sigma(\mathrm{neg}(\phi))^\omega \models \phi$ iff $\forall\rho \in (2^{AP})^\omega : \sigma\rho \models \phi$.*

The above result does not hold for all LTL formulas, as shown in the following example.

Example 4.8 *Consider the LTL formula $\phi = (\Diamond\Box q) \vee (\Diamond\Box\neg q)$. Note that this formula is not unipolar. Let $\sigma = \epsilon$. Obviously, $\{q\}^\omega \models \phi$, but $(\{q\}\emptyset)^\omega \not\models \phi$ and, hence, it is not the case that $\forall\rho \in (2^{AP})^\omega : \rho \models \phi$.*

4.2 An Algorithm to Compute Progress

To obtain an algorithm to compute the progress for the unipolar fragment of LTL, we present an alternative characterization of the progress measure. This alternative

characterization is cast in terms of a PTS built from the search as follows. We start from the transitions of the search and their source and target states. We add a sink state, which has a transition to itself with probability one. This sink state has label set L such that $\text{neg}(\phi) \subseteq L \subseteq AP_{\mathcal{S}} \setminus \text{pos}(\phi)$, where ϕ is the unipolar LTL formula being checked, and $AP_{\mathcal{S}}$ is the set of atomic propositions of the original system being searched. For simplicity, $L = \text{neg}(\phi)$ will be used. For each state which has not been fully explored yet, that is, the sum of the probabilities of its outgoing transitions is less than one, we add a transition to the sink state with the remaining probability. This PTS can be viewed as the minimal extension of the search (we will formalize this in Proposition 4.18). The PTS is defined as follows.

Definition 4.9 *Let T be a search of the PTS $\mathcal{S}$. The set $S_{\mathcal{S}}^T$ is defined by*

$$S_{\mathcal{S}}^T = \{\, \text{source}_{\mathcal{S}}(t) \mid t \in T \,\} \cup \{\, \text{target}_{\mathcal{S}}(t) \mid t \in T \,\} \cup \{s_0\}.$$

For each $s \in S_{\mathcal{S}}^T$,

$$\text{out}_{\mathcal{S}}(s) = \sum_{t \in T \,|\, \text{source}_{\mathcal{S}}(t) = s} \text{prob}_{\mathcal{S}}(t).$$

Let ϕ be a unipolar PNF formula. The PTS $\mathcal{S}_{T,\phi}$ is defined by

- $S_{\mathcal{S}_{T,\phi}} = S_{\mathcal{S}}^T \cup \{s_\perp\}$
- $T_{\mathcal{S}_{T,\phi}} = T \cup \{\, t_s \mid s \in S_{\mathcal{S}}^T \wedge \text{out}_{\mathcal{S}}(s) < 1 \,\} \cup \{t_\perp\}$
- $AP_{\mathcal{S}_{T,\phi}} = AP_{\mathcal{S}}$
- $\text{source}_{\mathcal{S}_{T,\phi}}(t) = \begin{cases} \text{source}_{\mathcal{S}}(t) & \text{if } t \in T \\ s & \text{if } t = t_s \\ s_\perp & \text{if } t = t_\perp \end{cases}$
- $\text{target}_{\mathcal{S}_{T,\phi}}(t) = \begin{cases} \text{target}_{\mathcal{S}}(t) & \text{if } t \in T \\ s_\perp & \text{if } t = t_\perp \text{ or } t = t_s \end{cases}$
- $\text{prob}_{\mathcal{S}_{T,\phi}}(t) = \begin{cases} \text{prob}_{\mathcal{S}}(t) & \text{if } t \in T \\ 1 - \text{out}_{\mathcal{S}}(s) & \text{if } t = t_s \\ 1 & \text{if } t = t_\perp \end{cases}$
- $\text{label}_{\mathcal{S}_{T,\phi}}(s) = \begin{cases} \text{neg}(\phi) & \text{if } s = s_\perp \\ \text{label}_{\mathcal{S}}(s) & \text{otherwise} \end{cases}$

The above definition is very similar to [30, Definition 10]. The main difference is that we do not have final states and that we label the sink state differently.

Proposition 4.10 *Let T be a search of the PTS $\mathcal{S}$. Then the PTS $\mathcal{S}_{T,\phi}$ extends T.*

Proof Follows immediately from the definition of $\mathcal{S}_{T,\phi}$. □

Next, we will show that the PTS $\mathcal{S}_{T,\phi}$ is the minimal extension of the search T of the PTS $\mathcal{S}$. More precisely, we will prove that for any other extension $\mathcal{S}'$ of T we have that $\mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}) \leq \mu_{\mathcal{S}'}(\mathcal{B}^{\phi}_{\mathcal{S}'})$. To prove this result, we introduce a new notion and some of its properties.

Definition 4.11 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a linear-time property. The set $E^{\phi}_{\mathcal{S}}(T)$ is defined by*

$$E^{\phi}_{\mathcal{S}}(T) = \{\, e \in T^* \cap \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}}) \mid \forall e' \in B^e_{\mathcal{S}} : e' \models_{\mathcal{S}} \phi \,\}.$$

The set $E^{\phi}_{\mathcal{S}_{T,\phi}}(T)$ is minimal among the $E^{\phi}_{\mathcal{S}'}(T)$ where $\mathcal{S}'$ extends T.

Proposition 4.12 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. For any unipolar LTL formula ϕ, $E^{\phi}_{\mathcal{S}_{T,\phi}}(T) \subseteq E^{\phi}_{\mathcal{S}'}(T)$.*

Proof Let $e \in E^{\phi}_{\mathcal{S}_{T,\phi}}(T)$. Then $e \in T^* \cap \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}_{T,\phi}})$. Since $\mathcal{S}'$ and $\mathcal{S}_{T,\phi}$ both extend T, we can conclude from Proposition 2.38(a) that $e \in T^* \cap \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}'})$.

It remains to prove that $e' \models_{\mathcal{S}'} \phi$ for all $e' \in B^e_{\mathcal{S}'}$. Let $e' \in B^e_{\mathcal{S}'}$. We distinguish two cases. If $e' \in T^\omega$ then $e' \in B^e_{\mathcal{S}_{T,\phi}}$ by Proposition 2.38(b). Since also $e \in E^{\phi}_{\mathcal{S}_{T,\phi}}(T)$, we have that $e' \models_{\mathcal{S}_{T,\phi}} \phi$. By Proposition 2.40, $e' \models_{\mathcal{S}'} \phi$.

Assume that $e' \notin T^\omega$. Then $e' = e_f t e_\ell$ such that $e_f \in T^*$ and $t \notin T$. Since $e \in T^*$ and $e' \in B^e_{\mathcal{S}'}$, we have that e is a prefix of e_f. From the construction of $\mathcal{S}_{T,\phi}$ we can derive that $e_f t_s t^\omega_\perp \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}}$, where s is the final state of e_f. Since e is a prefix of e_f, we have that $e_f t_s t^\omega_\perp \in B^e_{\mathcal{S}_{T,\phi}}$. Because $e \in E^{\phi}_{\mathcal{S}_{T,\phi}}(T)$, we can conclude that $e_f t_s t^\omega_\perp \models_{\mathcal{S}_{T,\phi}} \phi$. Since $\mathrm{label}_{\mathcal{S}_{T,\phi}}(s_\perp) = \mathrm{neg}(\phi)$, we know that $\mathrm{trace}_{\mathcal{S}_{T,\phi}}(e_f t_s t^\omega_\perp) = \sigma \mathrm{neg}(\phi)^\omega$ for some $\sigma \in (2^{AP})^*$. Hence, $\mathrm{trace}_{\mathcal{S}_{T,\phi}}(e_f) = \sigma$. Because $\sigma \mathrm{neg}(\phi)^\omega \models \phi$, we have $\forall \rho \in (2^{AP})^\omega : \sigma\rho \models \phi$ by Proposition 4.6. Since $e_f \in T^*$, we have that $\mathrm{trace}_{\mathcal{S}_{T,\phi}}(e_f) = \mathrm{trace}_{\mathcal{S}'}(e_f) = \sigma$ and, hence, we can conclude that $\mathrm{trace}_{\mathcal{S}'}(e') = \sigma\rho$ for some $\rho \in (2^{AP})^\omega$. Hence, $e' \models_{\mathcal{S}'} \phi$. □

Next, we restrict our attention to those elements of $E^{\phi}_{\mathcal{S}}(T)$ which are minimal with respect to the prefix order, as described in Definition 2.20.

Proposition 4.13 *For each* $A \subseteq \text{pref}(\text{Exec}_{\mathcal{S}})$*,*

$$\bigcup\{\, B^e_{\mathcal{S}} \mid e \in A \,\} = \bigcup\{\, B^e_{\mathcal{S}} \mid e \in \min(A) \,\}.$$

Proof Since $\min(A) \subseteq A$, we can conclude that $\bigcup\{\, B^e_{\mathcal{S}} \mid e \in A \,\} \supseteq \bigcup\{\, B^e_{\mathcal{S}} \mid e \in \min(A) \,\}$.

By the definition of $\min(A)$, for each $e \in A$, there exists $e_m \in \min(A)$ such that $e_m \sqsubseteq e$. Thus for each $B^e_{\mathcal{S}}$ such that $e \in A$, there exists $B^{e_m}_{\mathcal{S}}$ such that $e_m \in \min(A)$ and $B^e_{\mathcal{S}} \subseteq B^{e_m}_{\mathcal{S}}$. Therefore, $\bigcup\{\, B^e_{\mathcal{S}} \mid e \in A \,\} \subseteq \bigcup\{\, B^e_{\mathcal{S}} \mid e \in \min(A) \,\}$. □

Proposition 4.14 *For each* $A \subseteq \text{pref}(\text{Exec}_{\mathcal{S}})$*,*

$$\mu_{\mathcal{S}}\left(\bigcup\{\, B^e_{\mathcal{S}} \mid e \in \min(A) \,\}\right) = \sum_{e \in \min(A)} \mu_{\mathcal{S}}(B^e_{\mathcal{S}}).$$

Proof Observe that if $e_1, e_2 \in \min(A)$ and $e_1 \neq e_2$, then e_1 and e_2 are incomparable, and hence $B^{e_1}_{\mathcal{S}} \cap B^{e_2}_{\mathcal{S}} = \emptyset$. Since the set $\text{pref}(\text{Exec}_{\mathcal{S}})$ is countable, the set A is countable as well (Proposition 2.3). Because a probability measure is countably additive (see Definition 2.7), $\mu_{\mathcal{S}}(\bigcup\{\, B^e_{\mathcal{S}} \mid e \in \min(A) \,\}) = \sum_{e \in \min(A)} \mu_{\mathcal{S}}(B^e_{\mathcal{S}})$. □

The following two corollaries show that Proposition 4.13 and Proposition 4.14 hold for unions of basic cylinder sets in an extension of a search, even when the prefixes of those basic cylinder sets are taken from a different extension.

Corollary 4.15 *Let the PTSs* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *extend the search* T *of the PTS* $\mathcal{S}$ *and let* ϕ *be a linear-time property. Then*

$$\bigcup_{e \in \min\left(E^{\phi}_{\mathcal{S}_2}(T)\right)} B^e_{\mathcal{S}_1} = \bigcup_{e \in E^{\phi}_{\mathcal{S}_2}(T)} B^e_{\mathcal{S}_1}.$$

Proof This is a direct result of Proposition 2.38 and Proposition 4.13. □

Corollary 4.16 *Let the PTSs* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *extend the search* T *of the PTS* $\mathcal{S}$*, and let* ϕ *be a linear-time property. If* $E^{\phi}_{\mathcal{S}_1}(T) \subseteq E^{\phi}_{\mathcal{S}_2}(T)$ *then*

$$\mu_{\mathcal{S}_2}\left(\bigcup\{\, B^e_{\mathcal{S}_2} \mid e \in \min(E^{\phi}_{\mathcal{S}_1}(T)) \,\}\right) = \sum_{e \in \min(E^{\phi}_{\mathcal{S}_1}(T))} \mu_{\mathcal{S}_2}(B^e_{\mathcal{S}_2}). \tag{4.1}$$

Proof This is a direct result of Proposition 2.38 and Proposition 4.14. □

Next, we show that measures of unions of basic cylinder sets prefixed by transitions within a search are the same for any extension of that search.

Proposition 4.17 *Let the PTSs $\mathcal{S}_1$ and $\mathcal{S}_2$ extend the search T of the PTS $\mathcal{S}$. For all $A \subseteq \text{pref}(\text{Exec}_{\mathcal{S}}) \cap T^*$,*

$$\mu_{\mathcal{S}_1}\left(\bigcup_{e\in A} B^e_{\mathcal{S}_1}\right) = \mu_{\mathcal{S}_2}\left(\bigcup_{e\in A} B^e_{\mathcal{S}_2}\right).$$

Proof

$$\begin{aligned}
\mu_{\mathcal{S}_1}\left(\bigcup_{e\in A} B^e_{\mathcal{S}_1}\right) &= \mu_{\mathcal{S}_1}\left(\bigcup_{e\in \min(A)} B^e_{\mathcal{S}_1}\right) \quad \text{[Proposition 4.13]} \\
&= \sum_{e\in\min(A)} \mu_{\mathcal{S}_1}(B^e_{\mathcal{S}_1}) \quad \text{[Proposition 4.14]} \\
&= \sum_{e\in\min(A)} \mu_{\mathcal{S}_2}(B^e_{\mathcal{S}_2}) \quad \text{[Proposition 2.39]} \\
&= \mu_{\mathcal{S}_2}\left(\bigcup_{e\in A} B^e_{\mathcal{S}_2}\right) \quad \text{[symmetric argument]}
\end{aligned}$$

□

Now, we are ready to prove that the PTS $\mathcal{S}_{T,\phi}$ is the minimal extension of the search T of the PTS $\mathcal{S}$.

Proposition 4.18 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$ and let ϕ be a unipolar PNF formula. Then*

$$\mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) \leq \mu_{\mathcal{S}'}(\mathcal{B}^{\phi}_{\mathcal{S}'}(T)).$$

Proof

$$
\begin{aligned}
&\mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) \\
&\quad = \mu_{\mathcal{S}_{T,\phi}}\left(\bigcup\{\, B^{e}_{\mathcal{S}_{T,\phi}} \mid e \in E^{\phi}_{\mathcal{S}_{T,\phi}}(T)\,\}\right) \\
&\quad = \mu_{\mathcal{S}_{T,\phi}}\left(\bigcup\{\, B^{e}_{\mathcal{S}_{T,\phi}} \mid e \in \min(E^{\phi}_{\mathcal{S}_{T,\phi}}(T))\,\}\right) \quad \text{[Proposition 4.13]} \\
&\quad = \sum_{e\in\min(E^{\phi}_{\mathcal{S}_{T,\phi}}(T))} \mu_{\mathcal{S}_{T,\phi}}(B^{e}_{\mathcal{S}_{T,\phi}}) \quad \text{[Proposition 4.14]} \\
&\quad = \sum_{e\in\min(E^{\phi}_{\mathcal{S}_{T,\phi}}(T))} \mu_{\mathcal{S}'}(B^{e}_{\mathcal{S}'}) \quad \text{[Proposition 2.39]} \\
&\quad = \mu_{\mathcal{S}'}\left(\bigcup\{\, B^{e}_{\mathcal{S}'} \mid e \in \min(E^{\phi}_{\mathcal{S}_{T,\phi}}(T))\,\}\right) \quad \text{[Proposition 4.12 and Corollary 4.16]} \\
&\quad = \mu_{\mathcal{S}'}\left(\bigcup\{\, B^{e}_{\mathcal{S}'} \mid e \in E^{\phi}_{\mathcal{S}_{T,\phi}}(T)\,\}\right) \quad \text{[Corollary 4.15]} \\
&\quad \leq \mu_{\mathcal{S}'}\left(\bigcup\{\, B^{e}_{\mathcal{S}'} \mid e \in E^{\phi}_{\mathcal{S}'}(T)\,\}\right) \quad \text{[Proposition 4.12]} \\
&\quad = \mu_{\mathcal{S}'}(\mathcal{B}^{\phi}_{\mathcal{S}'}(T))
\end{aligned}
$$

□

The above proposition gives us an alternative characterization of the progress measure.

Theorem 4.19 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a unipolar PNF formula. Then*

$$\mathrm{prog}_{\mathcal{S}}(T,\phi) = \mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)).$$

Proof This is a direct consequence of the definition of the progress measure and Proposition 4.18. □

Hence, in order to compute $\mathrm{prog}_{\mathcal{S}}(T,\phi)$, it suffices to compute the measure of $\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)$. Next, we will show that the latter is equal to the measure of the set of execution paths of $\mathcal{S}_{T,\phi}$ that satisfy ϕ. The proof consists of two parts. First, we prove the following inclusion.

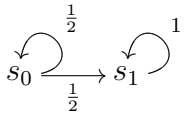
Proposition 4.20 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a unipolar PNF formula. Then*

$$\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T) \subseteq \{\, e \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi\,\}.$$

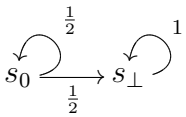
Proof Let $e \in \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)$. Then $e \in B^{e'}_{\mathcal{S}_{T,\phi}}$ for some $e' \in T^*$ such that $\forall e'' \in B^{e'}_{\mathcal{S}_{T,\phi}}$: $e'' \models_{\mathcal{S}_{T,\phi}} \phi$. Hence, $e \models_{\mathcal{S}_{T,\phi}} \phi$. □

The opposite inclusion does not hold in general, as shown in the following example.

Example 4.21 *Consider the PTS* $\mathcal{S}$



where s_0 *satisfies the atomic proposition* p*. Consider the search* $\{t_{00}\}$*. Let* $\phi = \Box p$*. Then the PTS* $\mathcal{S}_{T,\phi}$ *can be depicted by*



Hence, ${t_{00}}^{\omega} \models_{\mathcal{S}_{T,\phi}} \phi$*. By construction, the state* $s_\perp$ *does not satisfy* p*. So there is no prefix of* ${t_{00}}^{\omega}$ *for which all possible extensions satisfy* ϕ*. Therefore,* ${t_{00}}^{\omega} \notin \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}$.

However, we will show that the set $\{\, e \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \,\} \setminus \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)$ has measure zero. In the proof, we will use the following proposition.

Proposition 4.22 *Let* T *be a search of the PTS* $\mathcal{S}$ *and let* ϕ *be a unipolar PNF formula. Assume that* T *has not found a violation of* ϕ*. Then for all* $e \in T^{\omega} \cap \mathrm{Exec}_{\mathcal{S}_{T,\phi}}$*,* $e \models_{\mathcal{S}_{T,\phi}} \phi$*.*

Proof Let $e \in T^{\omega} \cap \mathrm{Exec}_{\mathcal{S}_{T,\phi}}$. Since T has not found a violation of ϕ, by definition there exists a PTS $\mathcal{S}'$ that extends T of $\mathcal{S}$ such that $e' \models_{\mathcal{S}'} \phi$ for all $e' \in \mathrm{Exec}_{\mathcal{S}'}$. Then $e \in \mathrm{Exec}_{\mathcal{S}'} \cap T^{\omega}$ by Proposition 2.38(b), because $\mathcal{S}'$ and $\mathcal{S}_{T,\phi}$ both extend T. Hence, $e \models_{\mathcal{S}'} \phi$. Therefore, from Proposition 2.40 we can conclude that $e \models_{\mathcal{S}_{T,\phi}} \phi$. □

Proposition 4.23 *Let* T *be a search of the PTS* $\mathcal{S}$ *and let* ϕ *be a unipolar PNF formula. If* T *has not found a violation of* ϕ *then*

$$\mu_{\mathcal{S}_{T,\phi}}(\{\, e \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \,\} \setminus \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) = 0.$$

Proof To avoid clutter, we denote the set $\{ e \in \text{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \} \setminus \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)$ by Z.

First, we show that $Z \subseteq T^{\omega}$. Assume that $e \in Z$. Towards a contradiction, suppose that $e \notin T^{\omega}$. From the construction of $\mathcal{S}_{T,\phi}$ we can deduce that $e = e' t_s {t_{\perp}}^{\omega}$ for some $e' \in T^*$, where s is the last state of e'. Let $\text{trace}_{\mathcal{S}_{T,\phi}}(e') = \sigma$, and $L = \text{neg}(\phi)$. Then $\text{trace}_{\mathcal{S}_{T,\phi}}(e) = \sigma L^{\omega}$. Since $e \in Z$, we have that $e \models_{\mathcal{S}_{T,\phi}} \phi$ and, hence, $\sigma L^{\omega} \models \phi$. By Proposition 4.6, $\forall \rho \in (2^{AP})^{\omega} : \sigma\rho \models \phi$. Hence, $\forall e'' \in B^{e'}_{\mathcal{S}_{T,\phi}} : e'' \models_{\mathcal{S}_{T,\phi}} \phi$. Since $e \in B^{e'}_{\mathcal{S}_{T,\phi}}$, we have that $e \in \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)$, which contradicts our assumption that $e \in Z$.

Next, we show that each state in $\{ \text{target}_{\mathcal{S}_{T,\phi}}(e) \mid e \in \text{pref}(Z) \}$ is transient (see Definition 2.46). By Proposition 2.49, it suffices to show that each state in $\{ \text{target}_{\mathcal{S}_{T,\phi}}(e) \mid e \in \text{pref}(Z) \}$ can reach the state $s_{\perp}$.

Since T has not found a violation of ϕ, we can conclude from Proposition 4.22 that $e \models_{\mathcal{S}_{T,\phi}} \phi$ for all $e \in T^{\omega}$. Hence, from the construction of $\mathcal{S}_{T,\phi}$ we can deduce that if $e \not\models_{\mathcal{S}_{T,\phi}} \phi$ then $e \notin T^{\omega}$ and, hence, e reaches $s_{\perp}$.

Let $e \in \text{pref}(Z)$. Hence, there exists $e' \in B^{e}_{\mathcal{S}_{T,\phi}}$ such that $e' \not\models_{\mathcal{S}_{T,\phi}} \phi$. Therefore, e' reaches $s_{\perp}$ and, hence, $\text{target}_{\mathcal{S}_{T,\phi}}(e)$ can reach $s_{\perp}$.

Since $Z \subseteq T^{\omega}$, the set $\{ \text{target}_{\mathcal{S}_{T,\phi}}(e) \mid e \in \text{pref}(Z) \}$ is finite. As stated in Proposition 2.50, the probability of remaining in a finite set of transient states is zero. As a consequence, the probability of remaining in the set $\{ \text{target}_{\mathcal{S}'}(e) \mid e \in \text{pref}(Z) \}$ is zero. Hence, we can conclude that $\mu_{\mathcal{S}_{T,\phi}}(Z) = 0$. □

From the above, we can derive the following result.

Theorem 4.24 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a unipolar PNF formula. If T has not found a violation of ϕ then*

$$\mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) = \mu_{\mathcal{S}_{T,\phi}}(\{ e \in \text{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \}).$$

Proof

$$\begin{aligned}
&\mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) \\
&\quad \leq \mu_{\mathcal{S}_{T,\phi}}(\{ e \in \text{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \}) \text{[Proposition 4.20 and } \mu_{\mathcal{S}_{T,\phi}} \text{ is monotone]} \\
&\quad = \mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) + \mu_{\mathcal{S}_{T,\phi}}(\{ e \in \text{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \} \setminus \mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) \\
&\qquad\qquad \text{[Proposition 4.20 and } \mu_{\mathcal{S}_{T,\phi}} \text{ is additive]} \\
&\quad = \mu_{\mathcal{S}_{T,\phi}}(\mathcal{B}^{\phi}_{\mathcal{S}_{T,\phi}}(T)) \text{[Proposition 4.23]}
\end{aligned}$$

□

Combining Theorem 4.19 and 4.24, we obtain the following characterization of the progress measure.

Corollary 4.25 *Let T be a search of the PTS $\mathcal{S}$ and let ϕ be a unipolar PNF formula. If T has not found a violation of ϕ then*

$$\mathrm{prog}_{\mathcal{S}}(T, \phi) = \mu_{\mathcal{S}_{T,\phi}}(\{\, e \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \,\}).$$

Proof Immediate consequence of Theorem 4.19 and 4.24. □

How to compute $\mu_{\mathcal{S}_{T,\phi}}(\{\, e \in \mathrm{Exec}_{\mathcal{S}_{T,\phi}} \mid e \models_{\mathcal{S}_{T,\phi}} \phi \,\})$ can be found, for example, in [11, Section 3.1]. Computing this measure is exponential in the size of ϕ and polynomial in the size of T. However, in general the size of the LTL formula is small, whereas the size of the search is huge. Hence, we expect our algorithm to be useful.

5 A Lower Bound on Progress

The algorithm developed in Chapter 4 to compute $\text{prog}_{\mathcal{S}}(T, \phi)$ for unipolar LTL formulas is exponential in the size of ϕ. In this section, we trade precision for efficiency. We present an algorithm that does not compute $\text{prog}_{\mathcal{S}}(T, \phi)$, but only provides a lower bound in polynomial time. This lower bound is tight for invariants. However, we also show an example in which the lower bound does not provide us any information.

Definition 5.1 *Let T be a search of the PTS $\mathcal{S}$. The PTS $\mathcal{S}_T$ is defined by*

- $S_{\mathcal{S}_T} = S_{\mathcal{S}}^T \cup \{s_\perp\}$,
- $T_{\mathcal{S}_T} = T \cup \{\, t_s \mid s \in S_{\mathcal{S}}^T \wedge \text{out}_{\mathcal{S}}(s) < 1 \,\} \cup \{t_\perp\}$,
- $AP_{\mathcal{S}_T} = AP_{\mathcal{S}}$
- $\text{source}_{\mathcal{S}_T}(t) = \begin{cases} \text{source}_{\mathcal{S}}(t) & \text{if } t \in T \\ s & \text{if } t = t_s \\ s_\perp & \text{if } t = t_\perp \end{cases}$
- $\text{target}_{\mathcal{S}_T}(t) = \begin{cases} \text{target}_{\mathcal{S}}(t) & \text{if } t \in T \\ s_\perp & \text{if } t = t_\perp \text{ or } t = t_s \end{cases}$
- $\text{prob}_{\mathcal{S}_T}(t) = \begin{cases} \text{prob}_{\mathcal{S}}(t) & \text{if } t \in T \\ 1 - \text{out}_{\mathcal{S}}(s) & \text{if } t = t_s \\ 1 & \text{if } t = t_\perp \end{cases}$
- $\text{label}_{\mathcal{S}_T}(s) = \begin{cases} \emptyset & \text{if } s = s_\perp \\ \text{label}_{\mathcal{S}}(s) & \text{otherwise} \end{cases}$

The above is similar to Definition 4.9. However, in this case the sink state has no labels, and hence $\mathcal{S}_T$ is the same regardless of the LTL formula being verified.

Next, we prove various properties of the system defined above, to show how it can be used to compute a lower bound for the progress measure. We use the concept of a transient state as defined in Definition 2.46.

First we show that if an execution prefix e formed by transitions of a search ends in a transient state, then the basic cylinder set formed by e in $\mathcal{S}_T$ is a superset of that formed by e in any extension of the search.

Proposition 5.2 *Let T be a search of the PTS $\mathcal{S}$. Let the PTS $\mathcal{S}'$ extend T. For all $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$, if $\text{target}_{\mathcal{S}_T}(e)$ is non-transient, then $B^e_{\mathcal{S}'} \subseteq B^e_{\mathcal{S}_T}$.*

Proof Let $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$. Assume that $\text{target}_{\mathcal{S}_T}(e)$ is non-transient. Let $e' \in B^e_{\mathcal{S}'}$. Toward a contradiction, assume that $e' \notin B^e_{\mathcal{S}_T}$. Let $e'[i]$ be the longest prefix of e' in $\text{pref}(\text{Exec}_{\mathcal{S}_T})$. Note that e is a prefix of $e'[i]$. Let $s = \text{target}_{\mathcal{S}}(e'[i])$. Then $\text{out}_{\mathcal{S}}(s) < 1$. Hence, $e'[i]t_s t_\perp^\omega \in \text{Exec}_{\mathcal{S}_T}$. Therefore, $\text{target}_{\mathcal{S}_T}(e)$ can reach $s_\perp$. By Proposition 2.49, we can conclude that $\text{target}_{\mathcal{S}_T}(e)$ is transient, which contradicts our assumption. □

Next we prove that if an execution prefix in $\mathcal{S}_T$, e, only contains transitions in a search T and ends in a transient state, then all execution paths in the basic cylinder set formed by e in any extension of the search contain only transitions in T.

Proposition 5.3 *Let T be a search of the PTS $\mathcal{S}$. For all PTSs $\mathcal{S}'$ that extend T, and for all $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$, if $\text{target}_{\mathcal{S}_T}(e)$ is non-transient, then $B^e_{\mathcal{S}'} \subseteq \text{Exec}_{\mathcal{S}'} \cap T^\omega$.*

Proof Since $T_{\mathcal{S}_T} = T \cup \{\, t_s \mid s \in S^T_{\mathcal{S}} \text{ and } \text{out}_{\mathcal{S}}(s) < 1 \,\} \cup \{t_\perp\}$ and $\text{target}_{\mathcal{S}_T}(t_s) = s_\perp$ and $\text{target}_{\mathcal{S}_T}(t_\perp) = s_\perp$, we can conclude that for all $e' \in \text{Exec}_{\mathcal{S}_T}$, if e' does not reach $s_\perp$, then $e' \in T^\omega$.

Let $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$. If $\text{target}_{\mathcal{S}_T}(e)$ is non-transient, then by Proposition 2.49 it cannot reach $s_\perp$. Hence, for all $e' \in B^e_{\mathcal{S}_T}$, e' cannot reach $s_\perp$ and, therefore, $e' \in \text{Exec}_{\mathcal{S}_T} \cap T^\omega$.

Let $e' \in B^e_{\mathcal{S}'}$. By Proposition 5.2, $e' \in B^e_{\mathcal{S}_T}$. Hence, $e' \in \text{Exec}_{\mathcal{S}_T} \cap T^\omega$. By Proposition 2.38, $e' \in \text{Exec}_{\mathcal{S}'} \cap T^\omega$. □

Now, we show that the set of execution paths in $\mathcal{S}_T$ that consist only of transitions in a search T has a measure equal to that of the union of basic cylinder sets formed by prefixes in T^* which end in non-transient states.

Proposition 5.4 *Let T be a search of the PTS $\mathcal{S}$. Then* $\mu_{\mathcal{S}_T}(\text{Exec}_{\mathcal{S}_T} \cap T^\omega)$ $= \mu_{\mathcal{S}_T}\left(\bigcup\{B^e_{\mathcal{S}_T} \mid e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^* \wedge \text{target}_{\mathcal{S}_T}(e) \textit{ is non-transient}\}\right)$.

Proof Let $Y = \bigcup\{B^e_{\mathcal{S}_T} \mid e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^* \wedge \text{target}_{\mathcal{S}_T}(e) \text{ is non-transient}\}$. Let $Z = (\text{Exec}_{\mathcal{S}_T} \cap T^\omega) \setminus Y$.

From Proposition 5.3, we can conclude that $Y \subseteq \text{Exec}_{\mathcal{S}_T} \cap T^\omega$. Since $\mu_{\mathcal{S}_T}$ is monotone (Proposition 2.9), $\mu_{\mathcal{S}_T}(Y) \leq \mu_{\mathcal{S}_T}(\text{Exec}_{\mathcal{S}_T} \cap T^\omega)$. So it suffices to prove that $\mu_{\mathcal{S}_T}(Z) = 0$.

Let $e \in \text{pref}(Z)$. Toward a contradiction, assume that $\text{target}_{\mathcal{S}_T}(e)$ is non-transient. Then $B^e_{\mathcal{S}_T} \subseteq Y$, which contradicts $e \in \text{pref}(Z)$. Hence, for all $e \in \text{pref}(Z)$, $\text{target}_{\mathcal{S}_T}(e)$ is transient. Since the set T is finite, $\{\text{target}_{\mathcal{S}_T}(e) \mid e \in \text{pref}(Z)\}$ is a finite set of transient states. As described in Proposition 2.50, the probability of remaining in a finite set of transient states forever is 0. Therefore, $\mu_{\mathcal{S}_T}(Z) = 0$. □

Below, we show that certain basic cylinder sets have the same measure in $\mathcal{S}_T$ and in extensions of T.

Proposition 5.5 *Let $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. For all $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$, if $\text{target}_{\mathcal{S}_T}(e)$ is non-transient, then $B^e_{\mathcal{S}_T} = B^e_{\mathcal{S}'}$.*

Proof From Proposition 5.3, we can conclude that $B^e_{\mathcal{S}_T} \subseteq T^\omega$ and $B^e_{\mathcal{S}'} \subseteq T^\omega$. From Proposition 2.38, we can deduce that $\text{Exec}_{\mathcal{S}_T} \cap T^\omega = \text{Exec}_{\mathcal{S}'} \cap T^\omega$. Hence, $B^e_{\mathcal{S}_T} = B^e_{\mathcal{S}'}$. □

Now we prove that if an execution prefix e in $\mathcal{S}_T$ contains only transitions in a search T and ends in a transient state, and T has not found a violation of the linear-time property ϕ, then all execution paths in the basic cylinder set formed by e in any extension of the search satisfy ϕ.

Proposition 5.6 *Let $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Assume the search has not found a violation of the linear-time property ϕ. For all $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$, if $\text{target}_{\mathcal{S}_T}(e)$ is non-transient, then $e' \models_{\mathcal{S}'} \phi$ for all $e' \in B^e_{\mathcal{S}'}$.*

Proof Let $e \in \text{pref}(\text{Exec}_{\mathcal{S}_T}) \cap T^*$. Assume that $\text{target}_{\mathcal{S}_T}(e)$ is non-transient. Toward a contradiction, assume that $e \not\models_{\mathcal{S}'} \phi$ for some $e' \in B^e_{\mathcal{S}'}$. Let $\mathcal{S}''$ be an arbitrary PTS that extends T. By Proposition 5.5, $e' \in B^e_{\mathcal{S}'} = B^e_{\mathcal{S}_T} = B^e_{\mathcal{S}''}$. By Proposition 5.3, $e' \in T^\omega \cap \text{Exec}_{\mathcal{S}''}$. By Proposition 2.40, $e' \not\models_{\mathcal{S}''} \phi$. Since $\mathcal{S}''$ was chosen arbitrarily, this contradicts our assumption that T has not found a violation of ϕ. □

Next, we use the properties defined above to show that the measure of execution paths in $\mathcal{S}_T$ that contain only transitions explored during a search is a lower bound on the progress of that search.

Theorem 5.7 *Assume the search T of the PTS $\mathcal{S}$ has not found a violation of the linear-time property ϕ. Then $\mu_{\mathcal{S}_T}(\mathrm{Exec}_{\mathcal{S}_T} \cap T^\omega) \leq \mathrm{prog}_{\mathcal{S}}(T, \phi)$.*

Proof Let $\mathcal{S}'$ be an arbitrary PTS that extends the search T of the PTS $\mathcal{S}$. Then

$$\begin{aligned}
&\mu_{\mathcal{S}_T}(\mathrm{Exec}_{\mathcal{S}_T} \cap T^\omega) \\
&\quad = \mu_{\mathcal{S}_T}\left(\bigcup\{\, B^e_{\mathcal{S}_T} \mid e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}_T}) \cap T^* \wedge \mathrm{target}_{\mathcal{S}_T}(e) \text{ is non-transient} \,\}\right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[Proposition 5.4]} \\
&\quad = \mu_{\mathcal{S}'}\left(\bigcup\{\, B^e_{\mathcal{S}'} \mid e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}_T}) \cap T^* \wedge \mathrm{target}_{\mathcal{S}_T}(e) \text{ is non-transient} \,\}\right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[Proposition 4.17 and 5.3]} \\
&\quad \leq \mu_{\mathcal{S}'}\left(\bigcup\{\, B^e_{\mathcal{S}'} \mid e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}_T}) \cap T^* \wedge \forall e' \in B^e_{\mathcal{S}'} : e' \models_{\mathcal{S}'} \phi \,\}\right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[Proposition 5.6 and 2.9]} \\
&\quad = \mu_{\mathcal{S}'}\left(\bigcup\{\, B^e_{\mathcal{S}'} \mid e \in \mathrm{pref}(\mathrm{Exec}_{\mathcal{S}'}) \cap T^* \wedge \forall e' \in B^e_{\mathcal{S}'} : e' \models_{\mathcal{S}'} \phi \,\}\right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{[Proposition 2.38]} \\
&\quad = \mu_{\mathcal{S}'}(\mathcal{B}^\phi_{\mathcal{S}'}(T))
\end{aligned}$$

Since $\mathcal{S}'$ is arbitrary,

$$\mu_{\mathcal{S}_T}(\mathrm{Exec}_{\mathcal{S}_T} \cap T^\omega) \leq \inf\{\, \mu_{\mathcal{S}'}(\mathcal{B}^\phi_{\mathcal{S}'}(T)) \mid \mathcal{S}' \text{ extends } T \text{ of } \mathcal{S} \,\} = \mathrm{prog}_{\mathcal{S}}(T, \phi).$$

□

From the construction of $\mathcal{S}_T$ we can conclude that $\mu_{\mathcal{S}_T}(T^\omega \cap \mathrm{Exec}_{\mathcal{S}_T})$ is the same as $\mu_{\mathcal{S}_T}(\{\, e \in \mathrm{Exec}_{\mathcal{S}_T} \mid e \text{ does not reach } s_\perp \,\})$, which is the same as $1 - \mu_{\mathcal{S}_T}(\{\, e \in \mathrm{Exec}_{\mathcal{S}_T} \mid e \text{ reaches } s_\perp \,\})$. The latter can be computed in polynomial time using, for example, Gaussian elimination (see, for example, [2, Section 10.1.1]).

This algorithm has been implemented and incorporated into an extension of the model checker JPF [29]. While JPF is model checking sequential Java code which contains probabilistic choices, our extension also keeps track of the underlying PTS. The amount of memory needed to store this PTS is in general only a small fraction of the total amount of memory needed. Once our extension of JPF runs almost out of memory, it can usually free enough memory so that the lower bound can be computed from the stored PTS.

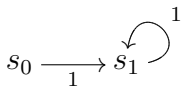
As was shown in [30, Theorem 4], the above bound is tight for invariants.

Proposition 5.8 *If the search T of the PTS $\mathcal{S}$ has not found a violation of the invariant ϕ, then*

$$\mu_{\mathcal{S}_T}(T^\omega \cap \mathrm{Exec}_{\mathcal{S}_T}) = \mathrm{prog}_{\mathcal{S}}(T, \phi).$$

In the example below, we present a search of a PTS for an LTL formula of which the progress is one whereas the lower bound is zero. In this case, the bound does not provide any information.

Example 5.9 *Consider the PTS*



Assume that the state s_1 satisfies the atomic proposition p. Consider the linear-time property $\bigcirc p$ and the search $\{t_{01}\}$. In this case, we have that $\mathrm{prog}_{\mathcal{S}}(\{t_{01}\}, \bigcirc p) = 1$ *but* $\mu_{\mathcal{S}_{\{t_{01}\}}}(\{t_{01}\}^{\omega} \cap \mathrm{Exec}_{\mathcal{S}_{\{t_{01}\}}}) = \mu_{\mathcal{S}_{\{t_{01}\}}}(\emptyset) = 0$.

6 Progress for LTL Until Formulas

Here, we present an algorithm to calculate the progress for properties of the form $p \,\mathcal{U}\, q$, where p and q are atomic propositions. The method works whenever the search has not found a violation of $p \,\mathcal{U}\, q$. This method also applies to formulas of the form $\Diamond p$, which are defined as $\text{true} \,\mathcal{U}\, p$.

This algorithm reduces the progress calculation to a simple reachability problem which, as discussed earlier, can be solved in polynomial time. The algorithm presented here is therefore more efficient than the method for unipolar LTL presented in Chapter 4, which can also be used to calculate the progress of $p \,\mathcal{U}\, q$.

To calculate the progress of a search T of a PTS $\mathcal{S}$ toward verifying $p \,\mathcal{U}\, q$, we first create a new system. We take the set of all states in $S_{\mathcal{S}}$ visited by T. We copy the outgoing transitions of states that do not satisfy q from $\mathcal{S}$, and add a probability-one self-loop to each state that satisfies q. Then, we take each state whose outgoing transition probabilities now sum to less than 1, and add to it a new transition that directs its missing transition probability to a sink state with no labels. This creates the system $\mathcal{S}_q$, as defined below.

Definition 6.1 *Let T be a search of the PTS $\mathcal{S}$, and let $q \in AP_{\mathcal{S}}$. The PTS $\mathcal{S}_q$ is defined as follows:*

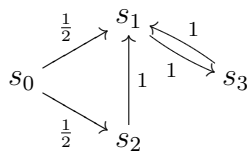
- $S_{\mathcal{S}_q} = S_{\mathcal{S}}^T \cup \{s_\perp\}$
- $T_{\mathcal{S}_q} = T_{\neg q} \cup T_{s_\perp} \cup T_q \cup \{t_\perp\}$ *where*
  - $T_{\neg q} = \{\, t \in T \mid q \notin \text{label}_{\mathcal{S}}(\text{source}_{\mathcal{S}}(t)) \,\}$
  - $T_q = \{\, t_s \mid s \in S_{\mathcal{S}}^T \text{ and } q \in \text{label}_{\mathcal{S}}(s) \,\}$
  - $T_{s_\perp} = \{\, t_s \mid s \in S_{\mathcal{S}}^T, q \notin \text{label}_{\mathcal{S}}(s) \text{ and } \text{out}_{\mathcal{S}}(s) < 1 \,\}$
- $AP_{\mathcal{S}_q} = AP_{\mathcal{S}}$
- $\text{source}_{\mathcal{S}_q}(t) = \begin{cases} \text{source}_{\mathcal{S}}(t) & \text{if } t \in T_{\neg q} \\ s & \text{if } t = t_s \\ s_\perp & \text{if } t = t_\perp \end{cases}$

- $\text{target}_{\mathcal{S}_q}(t) = \begin{cases} \text{target}_{\mathcal{S}}(t) & \text{if } t \in T_{\neg q} \\ s & \text{if } t = t_s \in T_q \\ s_\perp & \text{if } t \in T_{s_\perp} \text{ or } t = t_\perp \end{cases}$

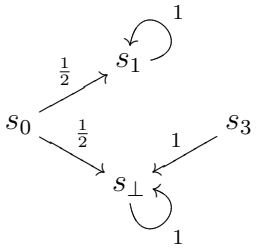
- $\text{prob}_{\mathcal{S}_q}(t) = \begin{cases} \text{prob}_{\mathcal{S}}(t) & \text{if } t \in T_{\neg q} \\ 1 & \text{if } t \in T_q \text{ or } t = t_\perp \\ 1 - \text{out}_{\mathcal{S}}(s) & \text{if } t = t_s \in T_{s_\perp} \end{cases}$

- $\text{label}_{\mathcal{S}_q}(s) = \begin{cases} \emptyset & \text{if } s = s_\perp \\ \text{label}_{\mathcal{S}}(s) & \text{otherwise} \end{cases}$

Example 6.2 *Consider the PTS $\mathcal{S}$ depicted by*



Consider the search $\{t_{01}, t_{13}\}$. Assume that only the state s_1 satisfies the atomic proposition q. Then the PTS $\mathcal{S}_q$ can be depicted by



Note that $\mathcal{S}_q$ does not extend the search T of $\mathcal{S}$, as specified in Definition 2.36. Hence, results on extensions from previous chapters cannot be applied directly to $\mathcal{S}_q$.

$\mathcal{S}_q$ contains a different set of transitions than the PTS $\mathcal{S}$, so below we confirm that $\mathcal{S}_q$ is also a PTS.

Proposition 6.3 *If $\mathcal{S}$ is a PTS, then $\mathcal{S}_q$ is a PTS.*

Proof For all $s \in S_{\mathcal{S}_q}$ such that $q \in \text{label}_{\mathcal{S}_q}(s)$, by definition there is only one transition t such that $\text{source}_{\mathcal{S}_q}(t) = s$, and its probability is 1.

For all $s \in S_{\mathcal{S}_q}$ such that $q \notin \text{label}_{\mathcal{S}_q}(s)$ and $\text{out}_{\mathcal{S}}(s) < 1$, the original transitions that make up $\text{out}_{\mathcal{S}}(s)$ are kept, and a new transition t_s is added from s to $s_\perp$ with $\text{prob}_{\mathcal{S}_q}(t_s) = 1 - \text{out}_{\mathcal{S}}(s)$. So the total probability of outgoing transitions is: $\text{prob}_{\mathcal{S}_q}(t_s) + \text{out}_{\mathcal{S}}(s) = (1 - \text{out}_{\mathcal{S}}(s)) + \text{out}_{\mathcal{S}}(s) = 1$.

For all $s \in \mathcal{S}_q$ such that $q \notin \text{label}_{\mathcal{S}_q}(s)$ and $\text{out}_{\mathcal{S}}(s) = 1$, the outgoing transitions are the same as in $\mathcal{S}$, so their sum is 1.

Thus, for all $s \in S_{\mathcal{S}_q}$, $\sum_{\text{source}_{\mathcal{S}_q}(t)=s} \text{prob}_{\mathcal{S}_q}(t) = 1$. Therefore, $\mathcal{S}_q$ is a PTS. □

Below, we show the relationships between execution prefixes whose extensions satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_q$, and in extensions of a search. These relationships are key to our ultimate goal of showing that $\mathcal{S}_q$ can be used to measure progress for $p\,\mathcal{U}\,q$.

Proposition 6.4 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then*

$$E_{\mathcal{S}_q}^{p\mathcal{U}q}(T_{\neg q}) \subseteq E_{\mathcal{S}'}^{p\mathcal{U}q}(T_{\neg q}) \subseteq E_{\mathcal{S}'}^{p\mathcal{U}q}(T).$$

Proof Because $T_{\neg q} \subseteq T$, it follows that $E_{\mathcal{S}'}^{p\mathcal{U}q}(T_{\neg q}) \subseteq E_{\mathcal{S}'}^{p\mathcal{U}q}(T)$.

To prove the other inclusion, we distinguish two cases. In the first case, the initial state satisfies q. Then for the first transition t of any execution, $q \in \text{label}_{\mathcal{S}_q}(\text{source}_{\mathcal{S}_q}(t))$, so $t \notin T_{\neg q}$. Thus ${T_{\neg q}}^* \cap \text{pref}(\text{Exec}_{\mathcal{S}_q}) = \emptyset$, and therefore $E_{\mathcal{S}_q}^{p\mathcal{U}q}(T_{\neg q}) = \emptyset \subseteq E_{\mathcal{S}'}^{p\mathcal{U}q}(T_{\neg q})$.

In the second case, the initial state does not satisfy q. Let $e \in E_{\mathcal{S}_q}^{p\mathcal{U}q}(T_{\neg q})$, and $e' \in B_{\mathcal{S}_q}^e$. By definition, $e' \models_{\mathcal{S}_q} p\,\mathcal{U}\,q$.

Because of the construction of $\mathcal{S}_q$, e' reaches only one state that satisfies q, say s_q, which has a self-loop of probability 1. Let $t_1 ... t_i$ be the transitions of e' up to and including the first transition with target s_q. Because $e' \models_{\mathcal{S}_q} p\,\mathcal{U}\,q$, for all $1 \leq j \leq i : p \in \text{label}_{\mathcal{S}_q}(\text{source}_{\mathcal{S}_q}(t_j))$. Also, because t_i is the first transition with a target that satisfies q, for all $1 \leq j \leq i : q \notin \text{label}_{\mathcal{S}_q}(\text{source}_{\mathcal{S}_q}(t_j))$. So $t_1 ... t_i \in {T_{\neg q}}^*$.

By the construction of $\mathcal{S}_q$, the transitions in $T_{\neg q}$, the states they reach, and the atomic propositions they satisfy are identical in $\mathcal{S}_q$ and $\mathcal{S}'$. So in $\mathcal{S}'$, $t_1 ... t_i$ also reaches states satisfying p followed by one state that satisfies q. Hence, all extensions of $t_1 ... t_i$ in $\mathcal{S}'$ also satisfy $p\,\mathcal{U}\,q$. Because each extension of e is prefixed by such a $t_1 ... t_i$, it must satisfy $p\,\mathcal{U}\,q$. Therefore for all $e'' \in B_{\mathcal{S}'}^e : e'' \models_{\mathcal{S}'} p\,\mathcal{U}\,q$. Thus, $e \in E_{\mathcal{S}'}^{p\mathcal{U}q}(T_{\neg q})$.

□

Another property of $\mathcal{S}_q$ is that its basic cylinder sets have the same measures as those in extensions of the search.

Proposition 6.5 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. For all $e \in E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$,*

$$\mu_{\mathcal{S}_q}(B^e_{\mathcal{S}_q}) = \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'}).$$

Proof Let $e = t_1...t_n$.

$$\begin{aligned}\mu_{\mathcal{S}_q}(B^e_{\mathcal{S}_q}) &= \prod_{1\leq i\leq n} \text{prob}_{\mathcal{S}_q}(t_i) \\ &= \prod_{1\leq i\leq n} \text{prob}_{\mathcal{S}'}(t_i) \quad [\text{Proposition 6.4, and } \forall i : t_i \in T_{\neg q}] \\ &= \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'})\end{aligned}$$

□

Proposition 6.6 to 6.13 below are presented for the purpose of proving Proposition 6.14, which states that for a search T of a PTS $\mathcal{S}$, the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ is less than or equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)$, for any extension $\mathcal{S}'$ of the search. Alone, Proposition 6.14 shows that the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ is less than or equal to the progress of the search T of the PTS $\mathcal{S}$. We later show that it is equal.

Proposition 6.6 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. For all $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$, there exists exactly one i such that $0 \leq i \leq |e|$ and $e[i] \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$.*

Proof Let $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$.

As shown in Proposition 6.4, $e \in E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q})$. Thus by definition, there exists some prefix $e[i] \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$.

There is only one such $e[i] \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$. As proof, suppose there were two such prefixes, $e[i]$ and $e[j]$, such that $i \neq j$. Then either $e[i] \sqsubset e[j]$, or $e[j] \sqsubset e[i]$, which contradicts the definition of a set of minimal elements. □

As a result of Proposition 6.6 above, the following function can be defined.

Definition 6.7 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. The function $f : \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \to \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$ is defined by*

$$\forall e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) : f(e) \sqsubseteq e.$$

For each $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$, $f(e)$ is the single element of $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$ that is a prefix of e. $f(e)$ is unique due to Proposition 6.6. The set of these elements is the image of f, defined below.

Definition 6.8 *Let T be a search of the PTS $\mathcal{S}$. The set* $\mathrm{Im}(f)$*, the image of f, is defined by*

$$\mathrm{Im}(f) = \{f(e) \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\}.$$

Below, we examine the basic cylinder sets whose prefixes are in $f^{-1}(e)$.

Proposition 6.9 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then for all $e' \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$,*

$$\sum_{e \in f^{-1}(e')} \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'}) = \mu_{\mathcal{S}'}\left(\bigcup_{e \in f^{-1}(e')} B^e_{\mathcal{S}'}\right).$$

Proof Since $f^{-1}(e') \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$, all elements of $f^{-1}(e')$ are incomparable, so by Proposition 2.23, the sets in $\{B^e_{\mathcal{S}'} \mid e \in f^{-1}(e')\}$ are disjoint. □

The function f could map more than one element of $\min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$ to the same element of $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$. Below, we show that the union of basic cylinder sets whose prefixes are mapped to a common prefix $e' \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$ is a subset of the basic cylinder set formed by e'.

Proposition 6.10 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then for all $e' \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$,*

$$\bigcup_{e \in f^{-1}(e')} B^e_{\mathcal{S}'} \subseteq B^{e'}_{\mathcal{S}'}.$$

Proof If $e \in f^{-1}(e')$, then e' is a prefix of e, and hence $B^e_{\mathcal{S}'} \subseteq B^{e'}_{\mathcal{S}'}$. □

Now, we show that union of all basic cylinder sets whose prefixes are mapped by f to a common element e' has a measure less than or equal to that of the basic cylinder set formed by e'.

Proposition 6.11 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. For all $e' \in \mathrm{Im}(f)$,*

$$\mu_{\mathcal{S}_q}\left(\bigcup_{e \in f^{-1}(e')} B^e_{\mathcal{S}_q}\right) \leq \mu_{\mathcal{S}'}(B^{e'}_{\mathcal{S}'}).$$

Proof

$$
\begin{aligned}
&\mu_{\mathcal{S}_q}\left(\bigcup_{e\in f^{-1}(e')} B^e_{\mathcal{S}_q}\right)\\
&\quad= \sum_{e\in f^{-1}(e')} \mu_{\mathcal{S}_q}(B^e_{\mathcal{S}_q}) \quad \text{[Proposition 4.14, and } f^{-1}(e') \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\text{]}\\
&\quad= \sum_{e\in f^{-1}(e')} \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'}) \quad \text{[Proposition 6.5]}\\
&\quad= \mu_{\mathcal{S}'}\left(\bigcup_{e\in f^{-1}(e')} B^e_{\mathcal{S}'}\right) \quad \text{[Proposition 6.9]}\\
&\quad\leq \mu_{\mathcal{S}'}(B^{e'}_{\mathcal{S}'}) \quad \text{[Proposition 6.10 and Proposition 2.9]}
\end{aligned}
$$

□

Next, we show that a union of basic cylinder sets whose prefixes map to a common element of $\mathrm{Im}(f)$ is disjoint from any union of basic cylinder sets with prefixes that map to a different element in $\mathrm{Im}(f)$.

Proposition 6.12 *Let* ${e_1}', {e_2}' \in \mathrm{Im}(f)$. *If* ${e_1}' \neq {e_2}'$, *then*

$$\left(\bigcup_{e_1\in f^{-1}({e_1}')} B^{e_1}_{\mathcal{S}_q}\right) \cap \left(\bigcup_{e_2\in f^{-1}({e_2}')} B^{e_2}_{\mathcal{S}_q}\right) = \emptyset.$$

Proof Let ${e_1}', {e_2}' \in \mathrm{Im}(f)$ and assume that ${e_1}' \neq {e_2}'$. It suffices to prove that for all $e_1 \in f^{-1}({e_1}')$ and $e_2 \in f^{-1}({e_2}')$, $B^{e_1}_{\mathcal{S}_q} \cap B^{e_2}_{\mathcal{S}_q} = \emptyset$.

Since ${e_1}', {e_2}' \in \mathrm{Im}(f) \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$ and ${e_1}' \neq {e_2}'$, ${e_1}'$ and ${e_2}'$ are incomparable. Because $e_1 \in f^{-1}({e_1}')$ and $e_2 \in f^{-1}({e_2}')$, ${e_1}'$ and ${e_2}'$ are prefixes of e_1 and e_2, respectively. Hence, e_1 and e_2 are incomparable and by Proposition 2.23, $B^{e_1}_{\mathcal{S}_q} \cap B^{e_2}_{\mathcal{S}_q} = \emptyset$. □

We now use the properties of the function f described above to prove that the union of all basic cylinder sets formed by elements of $\min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$ has a measure less than or equal to that of the union of basic cylinder sets formed by elements of $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$.

Proposition 6.13 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\}\right) \leq \mu_{\mathcal{S}'}\left(\bigcup\{B^e_{\mathcal{S}'} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))\}\right).$$

Proof

$$\begin{aligned}
&\mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\}\right) \\
&\quad= \mu_{\mathcal{S}_q}\left(\bigcup_{e' \in \mathrm{Im}(f)}\left(\bigcup_{e \in f^{-1}(e')} B^e_{\mathcal{S}_q}\right)\right) \\
&\quad= \sum_{e' \in \mathrm{Im}(f)} \mu_{\mathcal{S}_q}\left(\bigcup_{e \in f^{-1}(e')} B^e_{\mathcal{S}_q}\right) \quad \text{[Proposition 6.12]} \\
&\quad\leq \sum_{e' \in \mathrm{Im}(f)} \mu_{\mathcal{S}'}(B^{e'}_{\mathcal{S}'}) \quad \text{[Proposition 6.11]} \\
&\quad\leq \sum_{e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))} \mu_{\mathcal{S}'}(B^e_{\mathcal{S}'}) \quad [\mathrm{Im}(f) \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))] \\
&\quad= \mu_{\mathcal{S}'}\left(\bigcup\{B^e_{\mathcal{S}'} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))\}\right) \quad \text{[Proposition 4.14]}
\end{aligned}$$

□

Finally, we show that the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ is less than or equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)$

Proposition 6.14 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \leq \mu_{\mathcal{S}'}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)).$$

Proof

$$
\begin{aligned}
&\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \\
&\quad = \mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})\}\right) \\
&\quad = \mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\}\right) \quad \text{[Proposition 4.13]} \\
&\quad \leq \mu_{\mathcal{S}'}\left(\bigcup\{B^e_{\mathcal{S}'} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))\}\right) \quad \text{[Proposition 6.13]} \\
&\quad = \mu_{\mathcal{S}'}\left(\bigcup\{B^e_{\mathcal{S}'} \mid e \in E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q})\}\right) \quad \text{[Proposition 4.13]} \\
&\quad \leq \mu_{\mathcal{S}'}\left(\bigcup\{B^e_{\mathcal{S}'} \mid e \in E^{p\mathcal{U}q}_{\mathcal{S}'}(T)\}\right) \quad \text{[Proposition 6.4 and Proposition 2.9]} \\
&\quad = \mu_{\mathcal{S}'}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T))
\end{aligned}
$$

□

In the following proofs we use $\mathcal{S}_T$, as defined in Definition 5.1, as a specific instance of an extension of a search. The overall goal of Proposition 6.15 to 6.17 is to prove Proposition 6.18, which states that the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ is equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)$ for some extension $\mathcal{S}'$.

Proposition 6.15 *Let the PTS $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. Then*

$$\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T)) = \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q})).$$

Proof

We distinguish two cases. In the first case, $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T))$ contains a prefix of length 0, which we call ϵ. This occurs when every execution path of the system satisfies $p\,\mathcal{U}\,q$. In this case, $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T)) = \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q})) = \{\epsilon\}$.

Now, we assume that all prefixes in $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T))$ have length greater than zero. Because of Proposition 6.4, $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T)) \supseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$.

Next, we prove that $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T))$ implies $e \in {T_{\neg q}}^*$. Toward a contradiction, suppose e contains a transition whose source satisfies q. Let t_q be the first transition in e such that $q \in \text{label}_{\mathcal{S}'}(\text{source}_{\mathcal{S}'}(t_q))$, let $s_q = \text{source}_{\mathcal{S}'}(t_q))$, and let $e[i]$ be the prefix of e up to and including the first transition with target s_q. Since t_q has source s_q, the final transition of $e[i]$ cannot be t_q, so $e[i] \neq e$.

For any extension e' of e, $e' \models_{\mathcal{S}'} p\mathcal{U}q$. Therefore, $e[i]$ reaches only states that satisfy p before it reaches s_q. Therefore, all extensions of $e[i]$ satisfy $p\,\mathcal{U}\,q$. Thus, $e[i] \in$

$E^{p\mathcal{U}q}_{\mathcal{S}'}(T)$. Since $e[i] \neq e$, this contradicts the assumption that $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T))$. So e does not contain any transition t_q such that $q \in \text{label}_{\mathcal{S}'}(\text{source}_{\mathcal{S}'}(t_q))$, thus $e \in {T_{\neg q}}^*$. Therefore, $\min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T)) \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$. □

Now, we show that the set of minimal prefixes in $T_{\neg q}$ whose extensions satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_T$ is a subset of those whose extensions satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_q$.

Proposition 6.16 *Let T be a search of the PTS $\mathcal{S}$. Then*

$$\min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T_{\neg q})) \subseteq \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})).$$

Proof Let $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T_{\neg q}))$, and let $e' \in B^e_{\mathcal{S}_T}$.

Because $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T_{\neg q}))$, $e' \models_{\mathcal{S}_T} p\mathcal{U}q$. So e' must have some prefix $e'[i]$ such that $q \in \text{label}_{\mathcal{S}_T}(\text{target}_{\mathcal{S}_T}(e'[i]))$, and for all $t \in e'[i]$, $q \notin \text{label}_{\mathcal{S}_T}(\text{source}_{\mathcal{S}_T}(t))$ and $p \in \text{label}_{\mathcal{S}_T}(\text{source}_{\mathcal{S}_T}(t))$. The state $s_\perp$ satisfies neither p nor q, so $e'[i]$ does not include a transition to $s_\perp$. All transitions of $\mathcal{S}_T$ that do not lead to $s_\perp$ are in T, so $e'[i] \in T^*$. And since for all $t \in e'[i]$, $q \notin \text{label}_{\mathcal{S}_T}(\text{source}_{\mathcal{S}_T}(t))$, $e'[i] \in {T_{\neg q}}^*$.

Since the final transition of $e'[i]$ has a target that satisfies q, $e'[i+1] \notin {T_{\neg q}}^*$. Since $e \in {T_{\neg q}}^*$, e cannot be longer than $e'[i]$, therefore $e \sqsubseteq e'[i]$.

Thus, every extension e' of e in $\mathcal{S}_T$ includes some path e_2 that leads to a state that satisfies q, and whose intermediate states satisfy p, such that $e'[i] = ee_2$ and $ee_2 \in {T_{\neg q}}^*$.

Since the transitions in $T_{\neg q}$ are the same in $\mathcal{S}_T$ and $\mathcal{S}_q$, $e, ee_2 \in \text{pref}(\text{Exec}_{\mathcal{S}_q})$. And since $\text{trace}_{\mathcal{S}_T}(ee_2) = \text{trace}_{\mathcal{S}_q}(ee_2)$, all extensions of ee_2 in $\mathcal{S}_q$ also satisfy $p\,\mathcal{U}\,q$.

So in $\mathcal{S}_q$, every extension of e also includes a path e_2 leading to a state that satisfies q, and whose intermediate states satisfy p. Thus, all extensions of e in $\mathcal{S}_q$ satisfy $p\,\mathcal{U}\,q$. Therefore $e \in E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$.

Furthermore, $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$. As proof, suppose the contrary. Because $e \in E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, e must have some prefix $e[i] \neq e$ such that $e[i] \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$. As shown in Proposition 6.4, $e[i] \in E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q})$, which contradicts the assumption that $e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}'}(T_{\neg q}))$. □

Next, we show that the measure of basic cylinder sets with prefixes in T whose execution paths satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_T$ is less than or equal to the measure of basic cylinder sets with prefixes in $T_{\neg q}$ which satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_q$.

Proposition 6.17 *Let T be a search of the PTS $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}_T}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_T}(T)) \leq \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})).$$

Proof

$$\begin{aligned}
&\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \\
&\quad = \mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})\}\right) \\
&\quad = \mu_{\mathcal{S}_q}\left(\bigcup\{B^e_{\mathcal{S}_q} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\}\right) \text{ [Proposition 4.13]} \\
&\quad = \sum_{e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))} \mu_{\mathcal{S}_q}(B^e_{\mathcal{S}_q}) \text{ [Proposition 4.14]} \\
&\quad = \sum_{e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))} \mu_{\mathcal{S}_T}(B^e_{\mathcal{S}_T}) \text{ [Proposition 6.5]} \\
&\quad \geq \sum_{e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T_{\neg q}))} \mu_{\mathcal{S}_T}(B^e_{\mathcal{S}_T}) \text{ [Proposition 6.16]} \\
&\quad = \sum_{e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T))} \mu_{\mathcal{S}_T}(B^e_{\mathcal{S}_T}) \quad \text{[Proposition 6.15]} \\
&\quad = \mu_{\mathcal{S}_T}\left(\bigcup\{B^e_{\mathcal{S}_T} \mid e \in \min(E^{p\mathcal{U}q}_{\mathcal{S}_T}(T))\}\right) \quad \text{[Proposition 4.14]} \\
&\quad = \mu_{\mathcal{S}_T}\left(\bigcup\{B^e_{\mathcal{S}_T} \mid e \in E^{p\mathcal{U}q}_{\mathcal{S}_T}(T)\}\right) \quad \text{[Proposition 4.13]} \\
&\quad = \mu_{\mathcal{S}_T}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_T}(T))
\end{aligned}$$

□

Using the propositions above, we are now able to prove that the measure of basic cylinder sets with prefixes in $T_{\neg q}$ whose executions satisfy $p\,\mathcal{U}\,q$ in $\mathcal{S}_q$ is the same as the measure of basic cylinder sets with prefixes in T whose executions satisfy $p\,\mathcal{U}\,q$ in some extension of the search.

Proposition 6.18 *There exists an extension $\mathcal{S}'$ of the search T of the PTS $\mathcal{S}$ such that*

$$\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) = \mu_{\mathcal{S}'}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)).$$

Proof Let $\mathcal{S}' = \mathcal{S}_T$. This is then a direct result of Proposition 6.14 and Proposition 6.17. □

Consider a search T of a PTS $\mathcal{S}$, and consider the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$. We now have Proposition 6.14, which shows that this measure is less than or equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)$ for any extension $\mathcal{S}'$, and Proposition 6.18, which shows that this measure is equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}'}(T)$ for at least one extension. Together with the definition of the progress measure (Definition 2.43), these propositions give rise to the following theorem, which states that the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ is equal to the progress of the search toward verifying $p\,\mathcal{U}\,q$.

Theorem 6.19 *Let T be a search of the PTS $\mathcal{S}$. Then*

$$\mathrm{prog}_{\mathcal{S}}(T, p\,\mathcal{U}\,q) = \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})).$$

Proof This is an immediate consequence of Proposition 6.14 and Proposition 6.18. □

Now that we have shown that $\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$ is equal to the progress of the search toward verifying $p\,\mathcal{U}\,q$, we show that it is also equal to the reachability probability of the state $s_\perp$ in $\mathcal{S}_q$, provided that the search has not found a violation of $p\,\mathcal{U}\,q$.

By the construction of $\mathcal{S}_q$, any execution path that reaches $s_\perp$ contains $t_\perp$. So, Proposition 6.20 to 6.26 show that the measure discussed above, $\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$, is equal to the measure of executions of $\mathcal{S}_q$ that do not include $t_\perp$. We begin by showing that no execution in $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ contains $t_\perp$.

Proposition 6.20 *Let T be a search of the PTS $\mathcal{S}$. If $e \in \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, then $t_\perp \notin e$.*

Proof If $|e| = 0$, clearly $t_\perp \notin e$. Otherwise, because $e \in \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, it must reach a state that satisfies q. By the construction of $\mathcal{S}_q$, all states that satisfy q have probability-one self-loops, so no execution that reaches them can later reach $s_\perp$. Therefore, e cannot contain $t_\perp$. □

Corollary 6.21 *Let T be a search of the PTS $\mathcal{S}$. Then*

$$\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \leq \mu_{\mathcal{S}_q}(\{e \in \mathrm{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}).$$

Proof The set $\bigcup\{B^e_{\mathcal{S}_q} \mid t_\perp \in e\}$ is a countable union of measurable sets and hence is measurable. Therefore, its complement $\{e \in \mathrm{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}$ is measurable as well.

As shown in Proposition 6.20, if $e \in \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, then $t_\perp \notin e$. So, since $\mu_{\mathcal{S}_q}$ is monotone (Proposition 2.9), $\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) \leq \mu_{\mathcal{S}_q}(\{e \in \mathrm{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\})$. □

Below, we present two situations in which properties of $\mathcal{S}_q$ indicate that the search has found a violation of $p\,\mathcal{U}\,q$. Firstly, we prove that the search has found a violation if $\mathcal{S}_q$ has an execution path in ${T_{\neg q}}^\omega$.

Proposition 6.22 *Let T be a search of the PTS $\mathcal{S}$. If there exists $e \in {T_{\neg q}}^\omega \cap \mathrm{Exec}_{\mathcal{S}_q}$, then the search has found a violation of $p\,\mathcal{U}\,q$.*

Proof Let $e \in {T_{\neg q}}^\omega \cap \mathrm{Exec}_{\mathcal{S}_q}$, and let $\mathcal{S}'$ extend the search T of the PTS $\mathcal{S}$. The transitions of $T_{\neg q}$, as well as the states they visit and their labels, are identical in every PTS that extends the search. Therefore, $e \in \mathrm{Exec}_{\mathcal{S}'}$. Because no transition in $T_{\neg q}$ has a source that satisfies q, an execution in ${T_{\neg q}}^\omega$ has not visited any state that satisfies q. Thus, $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$. Since $\mathrm{trace}_{\mathcal{S}_q}(e) = \mathrm{trace}_{\mathcal{S}'}(e)$, $e \not\models_{\mathcal{S}'} p\,\mathcal{U}\,q$.

Therefore, every extension $\mathcal{S}'$ contains this execution e that does not satisfy $p\,\mathcal{U}\,q$. By definition, this constitutes a violation of $p\,\mathcal{U}\,q$ within the search T of $\mathcal{S}$. □

Secondly, we show that the search has found a violation if any execution path in $\mathcal{S}_q$ does not satisfy $p\,\mathcal{U}\,q$ and does not contain $t_\perp$.

Proposition 6.23 *Let T be a search of the PTS $\mathcal{S}$. If there exists $e \in \mathrm{Exec}_{\mathcal{S}_q}$ such that $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$ and $t_\perp \notin e$, then the search has found a violation of $p\,\mathcal{U}\,q$.*

Proof Let $e \in \mathrm{Exec}_{\mathcal{S}_q}$ such that $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$ and $t_\perp \notin e$.

$t_\perp \notin e$, so e can contain only transitions in $T_{\neg q}$, and probability-one self-loops on states that satisfy q, because these are the only transitions in $\mathcal{S}_q$ that do not lead to $s_\perp$ and $t_\perp$.

If e contains no probability-one self-loops on states that satisfy q, then $e \in {T_{\neg q}}^\omega$, and the proof is the same as in Proposition 6.22.

Suppose e does contain probability-one self-loops on states that satisfy q. Because these self-loops have probability 1, e can only include one such loop, and therefore reaches only one state that satisfies q.

Because $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$, e must have reached some state that does not satisfy p before reaching the state that satisfies q. Therefore, e has some prefix $e[i]$ that consists

of all transitions up to and including the first transition to a state that does not satisfy p. Clearly, any possible extension of $e[i]$ will violate $p\,\mathcal{U}\,q$. Furthermore, because $e[i]$ occurs before e reaches the state that satisfies q, $e[i]$ itself does not contain any states that satisfy q. And, by assumption $e[i]$ does not contain $t_\perp$. Therefore, $e[i] \in {T_{\neg q}}^*$.

For any extension $\mathcal{S}'$, $T_{\neg q}$, all the states these transitions visit, and the labels of these states, are identical in $\mathcal{S}_q$ and $\mathcal{S}'$. Therefore, $e[i] \in \text{pref}(\text{Exec}_{\mathcal{S}'})$. And because any extension of $e[i]$ violates $p\,\mathcal{U}\,q$, every $\mathcal{S}'$ contains some execution with prefix $e[i]$ that violates $p\,\mathcal{U}\,q$. By definition, this constitutes a violation of $p\,\mathcal{U}\,q$ within the search T of $\mathcal{S}$. □

Next we show that in $\mathcal{S}_q$, only execution paths in $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$ satisfy $p\,\mathcal{U}\,q$.

Proposition 6.24 *Let T be a search of the PTS $\mathcal{S}$. For all $e \in \text{Exec}_{\mathcal{S}_q}$, if $e \notin \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, then $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$.*

Proof Let $e \in \text{Exec}_{\mathcal{S}_q}$ and $e \notin \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$.

Toward a contradiction, suppose $e \models_{\mathcal{S}_q} p\,\mathcal{U}\,q$. If the initial state satisfied q, there would be no executions not in $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$. Therefore, the initial state does not satisfy q. So e must have some prefix that reaches only states that satisfy p, followed by a state that satisfies q. Let $e[i]$ be the prefix of e up to and including the first transition whose target satisfies q. Clearly, $\forall e' \in B^{e[i]}_{\mathcal{S}_q} : e' \models_{\mathcal{S}_q} p\,\mathcal{U}\,q$.

Since $s_\perp$ has a probability-one self-loop and does not satisfy q, $t_\perp \notin e[i]$. And since the final transition of $e[i]$ is the first whose target satisfies q, for all $t \in e[i]$: $q \notin \text{label}_{\mathcal{S}_q}(\text{source}_{\mathcal{S}_q}(t))$. Thus, $e[i] \in {T_{\neg q}}^*$.

So, $e \in B^{e[i]}_{\mathcal{S}_q} \in \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$. This contradicts the assumption that $e \notin \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$. Therefore, $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$. □

We now show the set of execution paths in $\mathcal{S}_q$ that do not contain $t_\perp$ has a measure less than or equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_\neg q)$.

Proposition 6.25 *If the search T of the PTS $\mathcal{S}$ has not found a violation of $p\mathcal{U}\,q$, then*

$$\mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}) \leq \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})).$$

Proof Let $\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\} = E_{\not\mathcal{B}} \cup E_{\mathcal{B}}$, where $E_{\mathcal{B}} = \{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e \text{ and } e \in \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})\}$, and $E_{\not\mathcal{B}} = \{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e \text{ and } e \notin \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})\}$. Clearly $E_{\mathcal{B}} \cap E_{\not\mathcal{B}} = \emptyset$, and $\mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}) = \mu_{\mathcal{S}_q}(E_{\mathcal{B}}) + \mu_{\mathcal{S}_q}(E_{\not\mathcal{B}})$.

By definition, $E_{\mathcal{B}} \subseteq \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, so $\mu_{\mathcal{S}_q}(E_{\mathcal{B}}) \leq \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))$.

$E_{\not\mathcal{B}}$ is empty. Toward a contradiction, assume that $e \in E_{\not\mathcal{B}}$. Because $e \notin \mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})$, according to Proposition 6.24, $e \not\models_{\mathcal{S}_q} p\,\mathcal{U}\,q$. Furthermore, $t_\perp \notin e$ by definition. Therefore, according to Proposition 6.23, there is a violation of $p\,\mathcal{U}\,q$ in the search T. By assumption no violation has been found, which is a contradiction.

Therefore,

$$\begin{aligned}\mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}) &= \mu_{\mathcal{S}_q}(E_{\mathcal{B}}) + \mu_{\mathcal{S}_q}(E_{\not\mathcal{B}}) \\ &= \mu_{\mathcal{S}_q}(E_{\mathcal{B}}) \\ &\leq \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q}))\end{aligned}$$

□

Finally, we improve the result above by showing that the measure of execution paths in $\mathcal{S}_q$ that do not contain $t_\perp$ is in fact equal to the measure of $\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_\neg q)$.

Proposition 6.26 *If the search T of the PTS $\mathcal{S}$ has not found a violation of $p\mathcal{U}\,q$, then*

$$\mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\}) = \mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})).$$

Proof Direct result of Corollary 6.21 and Proposition 6.25. □

The two corollaries below help clarify the relationship between the probability of reaching $s_\perp$, and the progress toward verifying $p\,\mathcal{U}\,q$.

Corollary 6.27 *If the search T of the PTS $\mathcal{S}$ has not found a violation of $p\,\mathcal{U}\,q$, then*

$$\mu_{\mathcal{S}_q}(\mathcal{B}^{p\mathcal{U}q}_{\mathcal{S}_q}(T_{\neg q})) = 1 - \mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid e \textit{ contains } t_\perp\}).$$

Proof Result of Proposition 6.26, and $1 - \mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid e \text{ contains } t_\perp\}) = \mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid t_\perp \notin e\})$. □

Corollary 6.28 *If the search T of the PTS $\mathcal{S}$ has not found a violation of $p \,\mathcal{U}\, q$, then*

$$\text{prog}_{\mathcal{S}}(T, p \,\mathcal{U}\, q) = 1 - \mu_{\mathcal{S}_q}(\{e \in \text{Exec}_{\mathcal{S}_q} \mid e \textit{ contains } t_\perp\})$$

Proof This is a result of Theorem 6.19 and Corollary 6.27. □

Due to the construction of $\mathcal{S}_q$, the measure of execution paths that contain $t_\perp$ is equal to the measure of execution paths that reach $s_\perp$. Therefore, calculating the progress for $p \,\mathcal{U}\, q$ when the search T of the PTS $\mathcal{S}$ has not found a violation can be reduced to calculating the reachability probability of $s_\perp$ in $\mathcal{S}_q$. As discussed in Chapter 5, this can be done by many well-known algorithms in polynomial time.

7 Computing Reachability Probabilities on a GPU

In this chapter, we consider the problem of calculating reachability probabilities, which can be used to calculate progress for invariants [30], progress for properties of the form $p \,\mathcal{U}\, q$ (Chapter 6), and the lower bound of progress (Chapter 5). We consider two methods to calculate these probabilities: the Jacobi method and the biconjugate gradient stabilized (BiCGStab) method. For each of the two methods, we present both a sequential and GPU implementation.

In Chapter 8, we will compare the performances of the algorithms presented here on randomized data, and on actual probabilistic model checking data.

7.1 Reachability Probabilities of a PTS

Below, we review the well-known problem of computing the reachability probabilities of a PTS. Our presentation is based on [2, Section 10.1].

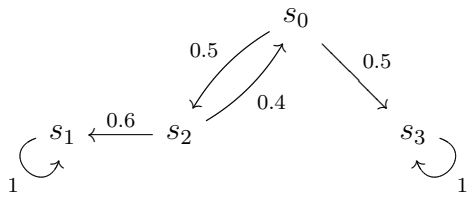
The transition probability function of a PTS $\mathcal{S}$ can be represented as a matrix $\mathbf{P}_{\mathcal{S}}$.

Definition 7.1 *Let $\mathcal{S}$ be a PTS. Let n be the number of states in $\mathcal{S}$. A* transition probability matrix $\mathbf{P}_{\mathcal{S}}$ *can be created from $\mathcal{S}$ as follows:*

- $\mathbf{P}_{\mathcal{S}}$ *is an $n \times n$ matrix.*
- *The states of $\mathcal{S}$ are numbered from 0 to $n - 1$.*
- *Each entry $\mathbf{P}_{\mathcal{S}i,j}$ represents the probability of taking a single transition from state i to state j in $\mathcal{S}$, calculated as follows:*

$$\mathbf{P}_{\mathcal{S}i,j} = \sum_{t \in T_{\mathcal{S}} \,|\, \mathrm{source}_{\mathcal{S}}(t) = i \,\wedge\, \mathrm{target}_{\mathcal{S}}(t) = j} \mathrm{prob}_{\mathcal{S}}(t)$$

Example 7.2 *The probability transition function of the PTS $\mathcal{S}$ depicted by*



can be represented by the matrix

$$\mathbf{P}_{\mathcal{S}} = \begin{bmatrix} 0 & 0 & 0.5 & 0.5 \\ 0 & 1 & 0 & 0 \\ 0.4 & 0.6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The probability of going from state i to state j, which is equal to the matrix entry $\mathbf{P}_{\mathcal{S}i,j}$, can also be expressed as $\mathbf{P}_{\mathcal{S}}(i,j)$.

In the following, we are interested in two particular events. First of all, given a PTS $\mathcal{S}$, a state $s \in S_{\mathcal{S}}$, and a set of states $GS \subseteq S_{\mathcal{S}}$, known as the *goal states*, we are interested in the probability of reaching a state in GS in one transition when starting in s. We denote the set of execution paths starting in s that reach a state in GS in one transition by $\{\, e \in \text{Exec}^s_{\mathcal{S}} \mid e \models_{\mathcal{S}} \bigcirc GS \,\}$. This set is measurable, that is, it belongs to the σ-algebra $\Sigma_{\mathcal{S}}$ [25].

Secondly, given a PTS $\mathcal{S}$, a state $s \in S_{\mathcal{S}}$, and a set of goal states $GS \subseteq S_{\mathcal{S}}$, we are interested in the probability of reaching a state in GS in zero or more transitions when starting in s. We denote the set of execution paths starting in s that reach a state in GS in zero or more transitions by $\{\, e \in \text{Exec}^s_{\mathcal{S}} \mid e \models_{\mathcal{S}} \Diamond GS \,\}$. This set is measurable [25].

The problem we are examining in this chapter is the following: Given a PTS $\mathcal{S}$, and set of goal states GS, what is the probability to reach a state in GS in zero or more transitions from the initial state? In other words, what is the probability of states in GS eventually being reached? That is, we want to compute $\mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \Diamond GS\})$. This is what is referred to as computing *reachability probabilities* in [2, Section 10.1.1]. This probability could be computed with an algorithm like that in [11, Section 3.1] discussed in Chapter 4. However, as explained in [2, Section 10.1], this measure can be calculated more efficiently using matrix operations on $\mathbf{P}_{\mathcal{S}}$.

To compute the reachability probabilities, one usually first partitions the set of states of $\mathcal{S}$ into three parts, based on their probability to reach the set GS of goal states.

Definition 7.3 *Let $\mathcal{S}$ be a PTS, and $GS \subseteq S_{\mathcal{S}}$ a set of goal states. Then:*

- $S_{=1} = \{\, s \in S_{\mathcal{S}} \mid \mu_{\mathcal{S}}(\{e \in Exec^s_{\mathcal{S}} \models_{\mathcal{S}} \Diamond GS\}) = 1 \,\}$
- $S_{=0} = \{\, s \in S_{\mathcal{S}} \mid \mu_{\mathcal{S}}(\{e \in Exec^s_{\mathcal{S}} \models_{\mathcal{S}} \Diamond GS\}) = 0 \,\}$
- $S_? = S_{\mathcal{S}} \setminus (S_{=1} \cup S_{=0})$

The partitioning of the set of states can be done easily by considering the *underlying digraph* of $\mathcal{S}$. The vertices of this digraph are the states of $\mathcal{S}$. There is an edge from state s to state s' if and only if $\mathbf{P}_{\mathcal{S}}(s, s') > 0$. Using graph algorithms, such as depth-first-search or breadth-first-search, the set of states can be partitioned as follows:

- To find $S_{=0}$, determine the set of all states that can reach GS (including the states in GS). The complement of this set is $S_{=0}$.
- To find $S_{=1}$, determine the set of all states that can reach $S_{=0}$. The complement of this set is $S_{=1}$.
- To find $S_?$, simply take the complement of $S_{=1} \cup S_{=0}$.

Example 7.4 *Consider the PTS $\mathcal{S}$ of Example 7.2 and let s_3 be the only goal state. Then $S_{=0} = \{s_1\}$, $S_{=1} = \{s_3\}$ and $S_? = \{s_0, s_2\}$.*

To compute the reachability probabilities, one must determine the probability of the initial state leading to a state in GS. That is, one has to compute $\mu_{\mathcal{S}}(\{e \in \text{Exec}_{\mathcal{S}} \mid e \models_{\mathcal{S}} \Diamond GS\})$. To do so, one must also determine the probabilities of reaching a state in GS from other states.

For each state $s \in S_{\mathcal{S}}$ we compute x_s, which is the probability of reaching GS from s, that is, $x_s = \mu_{\mathcal{S}}(\{e \in \text{Exec}^s_{\mathcal{S}} \mid e \models_{\mathcal{S}} \Diamond GS\})$. The values of x_s can be expressed as a vector, $\mathbf{x}$. For any state $s \in S_{=1}$, by definition $x_s = 1$. Similarly, for each $s \in S_{=0}$, $x_s = 0$. So once the states have been partitioned into the three sets of Definition 7.3, the only values of $\mathbf{x}$ that need to be calculated are $\{\, x_s \mid s \in S_? \,\}$. These values satisfy the following equation:

$$x_s = \sum_{s' \in S \setminus GS} \mathbf{P}_{\mathcal{S}}(s, s') \cdot x_{s'} + \sum_{s' \in GS} \mathbf{P}_{\mathcal{S}}(s, s').$$

So, we will create a matrix $\mathbf{M}$, which includes only the transition probabilities between states in $S_?$. For each s, $s' \in S_?$, $\mathbf{M}_{s,s'} = \mathbf{P}_{\mathcal{S}}(s, s')$.

To aid in calculations, we will also create a vector $\mathbf{b}$. For each $s \in S_?$, $b_s = \mu_{\mathcal{S}}(\{e \in \text{Exec}^s_{\mathcal{S}} \mid e \models_{\mathcal{S}} \bigcirc GS\}) = \sum_{s' \in GS} \mathbf{P}_{\mathcal{S}}(s, s')$, that is, the probability of a state in GS being reached from s in one transition.

Example 7.5 *Consider the PTS of Example 7.2 and let s_3 be the only goal state. Then states s_1 and s_3 are excluded because they do not belong to $S_?$, and*

$$\mathbf{M} = \begin{bmatrix} 0 & 0.5 \\ 0.4 & 0 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix}$$

For brevity, we will refer to the vector $\{ x_s \mid s \in S_? \}$ as $\mathbf{x}$. The equation for $\mathbf{x}$ can be written as $\mathbf{x} = \mathbf{M} \cdot \mathbf{x} + \mathbf{b}$. Rearranged, this becomes $(\mathbf{I} - \mathbf{M}) \cdot \mathbf{x} = \mathbf{b}$, where $\mathbf{I}$ is the identity matrix. $\mathbf{M}$ and $\mathbf{b}$ are already known from $\mathbf{P}_{\mathcal{S}}$. So, $\mathbf{x}$ can be found by solving the linear equation $(\mathbf{I} - \mathbf{M}) \cdot \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$.

We refer to the matrix $\mathbf{I} - \mathbf{M}$ as $\mathbf{A}$ for brevity. There are several ways to solve the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. Most obviously, one can find $\mathbf{A}^{-1}$. However, methods to find matrix inverses tend to have high computational complexity and, hence, for large matrices this becomes infeasible. For instance, Gauss-Jordan elimination has time complexity $O(n^3)$ (see, for example, [22, Section 2]).

Iterative approximation methods find solutions that are within a specified margin of error of the exact solutions, and can work much more quickly. The methods used in this thesis are in this category, and will be discussed in more detail next.

7.2 Iterative Methods

Solving the linear equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$ can be very time-consuming, especially when $\mathbf{A}$ is large. To solve this equation, there are numerous iterative methods. These methods compute successive approximations to obtain a more accurate solution to the linear system at each iteration.

These iterative methods can be classified into two groups: the stationary methods and the nonstationary ones. In stationary methods, the same information is used in each iteration. As a consequence, these methods are usually easier to understand and implement. However, convergence of these methods may be slow. In this thesis, we consider one stationary linear method, namely the Jacobi method. We chose this method because it is very well-known, and commonly used in model checking as well as other areas.

In nonstationary methods, the information used may change per iteration. These methods are usually more intricate and harder to implement, but often give rise to faster convergence. In this thesis we focus on one particular nonstationary linear method, namely the biconjugate gradient stabilized (BiCGStab) method. BiCGStab was chosen because it can be used on non-symmetric matrices, such as those found in model checking. It is also designed to minimize the impact of rounding errors, making it appropriate for computerized calculations.

Given a matrix $\mathbf{A}$ and a vector $\mathbf{b}$, the Jacobi method returns an approximate solution for $\mathbf{x}$ in the linear equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$. This method is a simplified version of an algorithm developed in 1846 by Carl Jacobi. It is derived in a fairly straightforward manner from the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, by considering the equation for each individual element of $\mathbf{x}$ while assuming that all other elements of $\mathbf{x}$ are fixed. The derivation is described more precisely in [4, Section 2.2.1]. Below, we present pseudocode for the Jacobi method.

```
1 Jacobi(A,b):
2 x := arbitrary vector
3 repeat
4     x' := x
5     for all i = 1...n do
6         x_i := 1/A_{i,i} · (b_i − Σ_{j≠i} A_{i,j} · x'_j)
7 until x is accurate enough
8 return x
```

To know when $\mathbf{x}$ is accurate enough to terminate, we calculate the difference between the current value of $\mathbf{x}$, and its value from the previous iteration $\mathbf{x}'$. As the Jacobi algorithm approaches the correct solution, $\mathbf{x}$ changes more gradually. So when the change in $\mathbf{x}$ is less than some predetermined amount, we accept that $\mathbf{x}$ is close enough to the exact solution for our purposes. The larger the amount chosen, the less accurate $\mathbf{x}$ will be, but the fewer iterations will be needed. With implementations such as ours that use finite-precision data types, too small an error value, such as zero, may not be achievable.

Like the Jacobi method, the BiCGStab method returns an approximate solution to $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ for a given matrix $\mathbf{A}$ and vector $\mathbf{b}$. The BiCGStab method was developed by Van der Vorst [24]. It is a type of Krylov subspace method. This category of linear methods includes several other algorithms, such as the Conjugate Gradients Squared and Bi-Conjugate Gradients methods from which BiCGStab was developed. The word "stabilized" in its name refers to the fact that rounding errors cause less irregular behavior than they do in related methods, making the BiCGStab algorithm appropriate for use with a computer's finite-precision data types. Unlike most Kyrlov subspace methods, the BiCGStab method can solve non-symmetric linear systems, which is necessary when working with probabilistic model checking data.

If exact arithmetic is used, the BiCGStab method will terminate in at most n iterations for an $n \times n$ matrix [24, page 636]. In practice, it often requires fewer iterations to find an approximate solution.

The un-preconditioned version of this method was used, as the sequential and

GPU performance of the preconditioning method would be a separate issue. Buchholz [8] did a limited comparison between a preconditioned and un-preconditioned version of the BiCGStab method specifically on matrices that represent stochastic processes, and it does not show a large performance difference.

A precise explanation of how the BiCGStab method operates requires numerous concepts otherwise unrelated to our work, and is thus beyond the scope of this thesis. Hence, we present only the pseudocode. For more details, we refer the reader to the highly cited paper [24] in which Van der Vorst introduces the method.

```
1  BiCGStab(A, b):
2  x := arbitrary vector
3  r := b − A · x
4  q := arbitrary vector such that q · r ≠ 0
5  y := 1; a := 1; w := 1; v := 0; p := 0
6  repeat
7      y′ := y
8      y := q · r
9      p := r + (y·a)/(y′·w) · (p − w · v)
10     v := A · p
11     a := y/(q·v)
12     s := r − a · v
13     t := A · s
14     w := (t·s)/(t·t)
15     x := x + a · p + w · s
16     r := s − w · t
17 until x is accurate enough
18 return x
```

For this algorithm, we know when $\mathbf{x}$ is accurate enough to terminate based on the vector $\mathbf{s}$, which is called the *residual*. For reasons discussed in [24], the residual approaches 0 as $\mathbf{x}$ becomes more accurate. Thus, some predetermined maximum value of $\mathbf{s}$ is used to decide when to terminate the algorithm. As for the Jacobi method, a maximum value of zero might not be possible due to rounding errors.

7.3 Binary Reachability

Before applying an iterative algorithm to an actual transition probability matrix, reachability properties need to be determined to divide the states into $S_{=1}$, $S_{=0}$ and $S_?$ sets, as described in Section 7.1. We refer to this as binary reachability, since the property of interest is whether or not each state can reach some subset of

states.

This calculation reduces the size of the matrix that needs to be solved and thus makes finding the solution faster. It is also necessary because the matrix $\mathbf{A}$ is often singular, and may not have specific properties required for iterative methods to solve equations containing it. For instance, the Jacobi method can only solve an equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ when $\mathbf{A}$ has no zero diagonal entries. The transition probability matrix of a system with a probability-one self-loop will have a one on the diagonal, and when subtracted from $\mathbf{I}$ will result in a matrix with a zero diagonal element. The Jacobi method will not be able to solve this matrix. However, since any state with a probability-one self-loop will reach the goal with either probability 0 or 1 (0 if it is not part of the goal set, 1 if it is), it will be in the $S_{=1}$ or $S_{=0}$ set, and can be eliminated before the Jacobi algorithm is applied.

To reduce the matrix to only $S_?$ states, first the $S_{=0}$ states are found using reachability algorithms, and then the $S_{=1}$ states, as explained in Section 7.1. Both these calculations can be done using the same basic reachability algorithm.

Below, we represent the sequential binary reachability pseudocode. The variable *canReach* is a boolean vector of length n. Each vector entry indicates whether or not the corresponding state can reach the goal states. Initially, only the entries corresponding to goal states are set to true, and all others to false.

```
1 Reachability(canReach):
2 boolean changed
3 repeat
4         changed := false
5         for each non-zero element A_{i,j}
6                 changed |= canReach_j & !canReach_i
7                 canReach_j |= canReach_i
8 until !changed
9 return canReach
```

7.4 General Purpose Graphics Processing Units (GPGPU)

Graphics processing units (GPUs) were primarily developed to improve the performance of graphics-intensive programs. The market driving their production has traditionally been video game players. They are a throughput-oriented technology, optimized for data-intensive calculations in which many identical operations can be done in parallel on different data. Unlike most commercially available multi-core central processing units (CPUs), which normally run up to eight threads in parallel, GPUs are designed to run hundreds of threads in parallel. They have many

more cores than CPUs, but these cores are generally less powerful. GPUs sacrifice latency and single-thread performance to achieve higher data throughput [15].

Originally, GPU APIs were designed exclusively for graphics use, and their underlying parallel capabilities were difficult to access for other types of computation. This changed in 2006 when NVIDIA released CUDA, the first architecture and API designed to allow GPUs to be used for a wider variety of applications [18, Section 1].

All parallel algorithms discussed in this thesis were implemented using CUDA. Different generations of CUDA-enabled graphics cards have different compute capability levels, which indicate the sets of features that they support. The graphics card used here is the NVIDIA GTX260, which has compute capability 1.3. Cards with higher compute capability have been released by NVIDIA since the GTX260 was developed.

The GTX260 has 896MB global on-card memory. This GPU supports double-precision floating-point operations, which is required for BiCGStab. Lower compute capabilities (1.2 and below) do not support this. The GTX260 does not support atomic floating-point operations. This limits the way operations that require dot-products and other row sums can be separated into threads. Since these sums require all elements of a row to be summed to a global variable, and atomic addition can not be used to protect the integrity of the sum, the structure of the threads must do so. Here, we have simply created one thread per row and done these additions sequentially. The next generation of NVIDIA GPUs (compute capability 2.0 and higher) support atomic floating-point addition, though only on single-precision numbers.

Current GPUs have limited memory to store matrix data. For all but the smallest matrices, some sort of compression is necessary so that they can be transferred to the GPU in their entirety.

For this work, data is stored using a compressed row storage, as described in [4, page 57]. In this format, only the non-zero elements of the matrix are recorded. The size of an $n \times n$ matrix with m non-zero elements compressed in this manner is $O(n+m)$, whereas uncompressed it is $O(n^2)$. This representation saves significant space when used to store sparse matrices.

Definition 7.6 *Given an $n \times n$ matrix with m non-zero elements, its* compressed row storage *representation consists of three vectors:*

- rstart *contains integers and has length $n+1$, where $rstart_i$ is the total number of non-zero elements in the first i rows.*
- col *also contains integers and has length m. It stores the column position of each non-zero element.*

- nonzero *contains floating points and has length m. It stores the non-zero elements of the matrix.*

Example 7.7 *The matrix*

$$\begin{bmatrix} 0.4 & 0 & 0 \\ 0 & 0 & 0.5 \\ 0 & 0 & 0.6 \end{bmatrix}$$

is represented by the vectors

$$\begin{aligned} \textit{rstart} &: [0, 1, 2, 3] \\ \textit{col} &: [0, 2, 2] \\ \textit{nonzero} &: [0.4, 0.5, 0.6] \end{aligned}$$

CUDA code consists of C code executed on a CPU, which in turn can execute portions of code on the GPU. We refer to methods that are executed on the GPU as *kernels*. Code in a kernel function is executed in parallel by each GPU core. Our GPU, like most at this point in time, requires each core to perform the same operations at the same time. When the kernel contains instructions that cause some threads to execute different operations than other threads (e.g., if statements), threads following one set of instructions must wait until threads following a different set are finished. This is called *divergence*. Divergence negatively impacts performance, and GPU algorithms are designed to minimize it whenever possible.

7.5 Parallel Implementations in CUDA

Next, we present parallel versions of the Jacobi, BiCGStab and binary reachability algorithms. These parallel versions are implemented in CUDA.

Our CUDA implementation of the Jacobi method is based on the one described by Bosnacki et al. in [6, 7]. Given an $n \times n$-matrix $\mathbf{A}$ and an n-vector $\mathbf{b}$, in the parallel implementation of the Jacobi method n threads are created. For each thread i, where $0 \leq i < n$, the following algorithm is used. Essentially, one thread traverses each row of the matrix $\mathbf{A}$ to compute the corresponding element of $\mathbf{x}$. Thus, each element of $\mathbf{x}$ is computed in parallel.

In the parallel implementation of the Jacobi method, each GPU thread performs the following operations:

```
1  i := thread id
2  if i = 0 then
3      terminate := true
4  if i < n then
5      d := b_i
6      l := rstart_i
7      h := rstart_{i+1} − 1
8      for all j = l...h do
9          d := d − nonzero_j · x_{col_j}
10     x_i := d / A_{i,i}
11     if x_i is not accurate enough then
12       terminate = false
```

The variable *terminate* is shared by all the threads to determine when the iteration can stop. Note that line 5–10 of the code above corresponds to line 5–6 of the Jacobi method in Section 7.2. Thus, this algorithm represents one iteration of the Jacobi method. The main program repeatedly launches the corresponding code on the GPU, waiting between executions for all threads to complete, until $\mathbf{x}$ reaches the desired accuracy.

When we started this research, we were aware of the paper by Gaikwad and Toke [14] that mentions a CUDA implementation of the BiCGStab method. Since we did not have access to their code, we implemented the BiCGStab method in CUDA ourselves.

Each iteration of the BiCGStab method consists of several matrix and vector multiplications, that each result in a vector. Most of these steps depend on completion of the previous step, and thus need to be done in sequence. Therefore, the steps were parallelized individually by creating one thread for each element of the resulting vectors.

For instance, the operation in line 15 is split between n threads, so for each $0 \leq i < n$ a thread does the following:

$$x_i = x_i + a \cdot p_i + w \cdot s_i$$

And thus each element of the vector is calculated in parallel. The matrix operation in line 10 is done as follows for each of the n threads:

$$v_i = \mathbf{A}_i \cdot \mathbf{p}$$

where $\mathbf{A}_i$ denotes the i^{th} row of the matrix $\mathbf{A}$. Dot products are done sequentially. It would be possible to increase parallelism by splitting these into multiple sums

done in parallel. This might require too much overhead to result in a significant performance gain.

Currently, CUDA requires all threads running on a GPU to execute the same code. There are some separate steps of the BiCGStab method which could be executed in parallel, but this is not possible with a single GPU of the type used here. This may be possible using the next generation of GPUs, or multiple GPUs. However, as described earlier, most steps of the algorithm must be done in sequence.

Below is an abbreviated version of the CUDA code used to implement BiCGStab. To save space, non-essential code has been removed, and some steps are summarized in square brackets. Kernel numbering corresponds to the line numbers of the algorithm in Section 7.2. Kernels for subsequent steps are combined when they require the same number of threads. Sequential steps are done on the GPU with a single thread, since this avoids time-consuming data transfers between the GPU and host computer.

The first portion of the code, below, executes on the CPU. Pointers prefixed by *d_* indicate data stored on the GPU, and n is the dimension of the matrix. The matrix $\mathbf{A}$ is represented on the GPU using compressed row storage as *d_rstart, d_col* and *d_nonzero*.

```
int terminate = 0;
for(int i = 0; i <= max_iterations && !terminate; i++)
     [d_yprime = d_y, switch pointers without moving data];
     step8Kernel<<<1,1>>>(d_B, d_y, d_yprime, d_a, d_w, d_q, d_r, n);
     step9Kernel<<<grid,block>>> (d_p, d_r, d_B, d_w, d_v, n, blocksz);
     matrixVectorMult<<<grid,block>>>
              (d_col, d_rstart, d_nonzero, d_p, d_v, n, blocksz);
     step11Kernel<<<1,1>>> (d_a, d_y, d_q, d_v, n);
     step12Kernel<<<grid,block>>> (d_s, d_r, d_a, d_v, n, blocksz);
     matrixVectorMult<<<grid,block>>>
              (d_col, d_rstart, d_nonzero, d_s, d_t, n, blocksz);
     step14Kernel<<<1,1>>> (d_w, d_t, d_s, n);
     step15_16Kernel<<<grid,block>>>
              (d_x, d_a, d_p, d_w, d_s, d_r, d_t, n, blocksz, d_terminate);
     [terminate = d_terminate, transferred from GPU to host machine];
```

The kernel methods below execute on the GPU. The kernels for steps 12 are 14 are very similar to previous steps, and are excluded. The `matrixVectorMult` method, not shown, is a generic parallel matrix-vector multiplication kernel.

```
__global__ static void step8Kernel(double *d_B, double *d_y, double *d_yprime,
 double *d_a, double *d_w, double *d_q, double *d_r, int n)
     double result = 0;
```

```
     for(int i = 0; i < n; i++) result += d_q[i] * d_r[i];
     *d_y = result;
     *d_B = (*d_y / *d_yprime) * (*d_a / *d_w); //prepare scalar for step 9


__global__ static void step9Kernel(double *d_p, double *d_r, double *d_B,
double *d_w, double *d_v, int n, int blocksz)
     int i = blockIdx.x * blocksz + threadIdx.x; //thread index
     if(i < n) d_p[i] = d_r[i] + *d_B * ( d_p[i] - *d_w * d_v[i]);


__global__ static void step11Kernel(double *d_a, double *d_y, double *d_q,
 double *d_v, int n)
     double dot_q_v = 0; //dot product of q,v
     for(int i = 0; i < n; i++) dot_q_v += d_q[i] * d_v[i];
     *d_a = *d_y / dot_q_v;


__global__ static void step15_16Kernel(double *d_x, double *d_a, double *d_p,
 double *d_w, double *d_s, double *d_r, double *d_t, int n, int blocksz,
 int* d_terminate)
     int i = blockIdx.x * blocksz + threadIdx.x; //thread index
     if(i == 0) *d_terminate = 1;  //one thread sets d_terminate
     if(i < n)
          d_x[i] = d_x[i] + *d_a * d_p[i] + *d_w * d_s[i]; //step 15
          double diff = abs(d_s[i]);
          if(diff > ERROR) *d_terminate = 0; //stop when residual is near zero
          d_r[i] = d_s[i] - *d_w * d_t[i]; //step 16
```

Binary reachability calculations are not usually a significant factor in model checking performance. They take very little time compared to the iterative matrix solvers, even when implemented sequentially. However, in certain situations there is potential for some time gain by implementing these on a GPU.

In our parallel implementation of the binary reachability algorithm, each row i of $\mathbf{A}$ is traversed independently by one thread. Each such thread contains the following local boolean variables:

- *changedNow* indicates whether the element currently being examined by this thread has caused a change.
- *changedEver* expresses whether any element seen by this thread has caused a change.

The threads also share one global boolean value:

- *changedGlobal* indicates whether any change has occurred in any thread during the current kernel execution, and is set to *false* at beginning of each kernel launch.

Each thread of the parallel implementation performs the following operations:

```
1  changedNow := false
2  changedEver := false
3  i := thread number
4
5  for each non-zero element A_{i,j} in row i of A
6          changedNow = canReach_j & !canReach_i
7          changedEver |= changedNow
8          if changedNow
9                  canReach_i := true
10
11 if changedEver
12         changedGlobal := true
```

Kernels are launched repeatedly until *changedGlobal* remains false after all threads finish.

The GPU algorithm considers all edges adjacent to each vertex in a single step, instead of considering each edge individually as in the sequential algorithm. Thus the average-case time complexity is lower.

The reason that line 8 of the pseudocode uses an if statement, which can cause divergence, instead of something such as $changedGlobal \mid = changedEver$, which would allow all threads to use an identical instruction, is the following:

If multiple threads with different values of *changedEver* try to simultaneously update *changedGlobal*, it can cause incorrect results. For example, suppose *changedGlobal* is false. One thread i has its local value $changedEver_i = true$, and another thread j has $changedEver_j = false$. At $changedGlobal \mid = changedEver$, each thread fetches the current value of *changedGlobal*, false. Thread i determines that $changedGlobal \mid changedEver_i = \text{true}$, the other that $changedGlobal \mid changedEver_j = \text{false}$. So thread i attempts to set *changedGlobal* to true, and thread j tries to set it to false. *changedGlobal* should be set to true if any thread has caused a change, so if it is set to false by thread j then the algorithm could produce incorrect results. Our approach works since threads only attempt to update the global value to true, so it cannot mistakenly be changed back to false.

Atomic operations were attempted, but resulted in slower performance since they cause each thread to perform the operation in sequence.

8 Performance of GPU Reachability Probability Calculations

In this chapter, we present the results of our experiments on iterative methods. We observe that which of the four algorithms performs best varies, depending on the size and density of the matrix being solved. BiCGStab is best for larger, denser matrices, and Jacobi for smaller, sparser ones. CUDA generally improves the performance of the Jacobi method (though it was slightly slower for random quick sort). CUDA improves BiCGStab performance on larger, denser matrices, but reduces performance on smaller, sparser ones.

For reachability probability calculations in model checking, our results show that CUDA Jacobi is a useful algorithm. CUDA BiCGStab was not very efficient for our model checking calculations, but we show that it could perform very well for problems involving denser matrices.

We also observed that GPU implementation could be beneficial for binary reachability calculations on dense matrices, or graphs with high vertex degree.

8.1 Performance on Randomly Generated Matrices

For these tests, random matrices were generated. The entries were random positive integers, placed in random locations. The matrices were then modified by adding the non-diagonal elements to have non-zero diagonal entries and be diagonally-dominant, which ensured that both the Jacobi and BiCGStab methods were able to solve equations containing them. For each matrix $\mathbf{A}$, a vector $\mathbf{b}$ of random integers was also created. This formed the equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, to be solved for $\mathbf{x}$. Each trial used a newly-generated matrix and vector.

Figure 8.1 shows the performances of the four implementations on random matrices of several sizes, and varying densities. The matrix densities are similar to those encountered in our probabilistic model checking examples. These graphs indicate that the implementations' relative performances change their order as the density increases. Furthermore, as the size of the matrix increases, the density at which the performance order changes becomes lower. Thus, which implementation

performs best depends on the size and density of the matrix it is being used on.

Generally, the graphs in Figure 8.1 show that the BiCGStab method is superior to the Jacobi method for denser matrices, but the Jacobi method performs best for very sparse matrices. They also demonstrate a consistent performance benefit from using CUDA to implement the Jacobi method. However, the third graph shows that the CUDA version of the BiCGStab method is only beneficial for larger, denser matrices. For other matrices, the sequential version of the BiCGStab method outperforms the CUDA version.

To confirm this, Figure 8.2 and Figure 8.3 show the implementations' performances on random matrices with 10% density. As the sizes of these relatively dense matrices increase, the CUDA versions of both implementations increasingly outperform their sequential counterparts, and the BiCGStab method significantly outperforms the Jacobi method.

It is easier to determine the relationship between matrix density and GPU performance gain for the binary reachability algorithm. As discussed earlier, the GPU algorithm examines all edges adjacent to a vertex in a single step, whereas the sequential algorithm examines each edge in series. So, a dense matrix will show the most benefit from GPU implementation. Similarly, the GPU algorithm will perform best on a graph with high vertex degree, even if the matrix is sparse overall.

For these tests, the density was approximately one non-zero entry for every ten zero entries, or 10%. The results, shown in Figure 8.4, predictably show significant performance gains from using the GPU. However, if the matrix had 1 entry or fewer per row, no benefit would be expected.

8.2 Performance on Probabilistic Model Checking Data

For these tests, the implementations of the iterative methods were tested on matrices representing PTSs based on actual randomized algorithms. These matrices were then reduced to only $S_?$ states and subtracted from the identity matrix, as discussed in Section 7.1.

JPF was used on two randomized algorithms to create this data. The biased die algorithm simulates a biased dice roll by means of biased coin flips, and random quick sort is a randomized version of the quick sort algorithm. Both algorithms were coded in Java by Zhang, who also created the JPF extension that outputs transition probability matrices of PTSs that correspond to the code being checked [28].

The JPF search strategies used were chosen to create the largest $S_?$ matrices relative to the size of the searched space. JPF's built-in depth first search was used

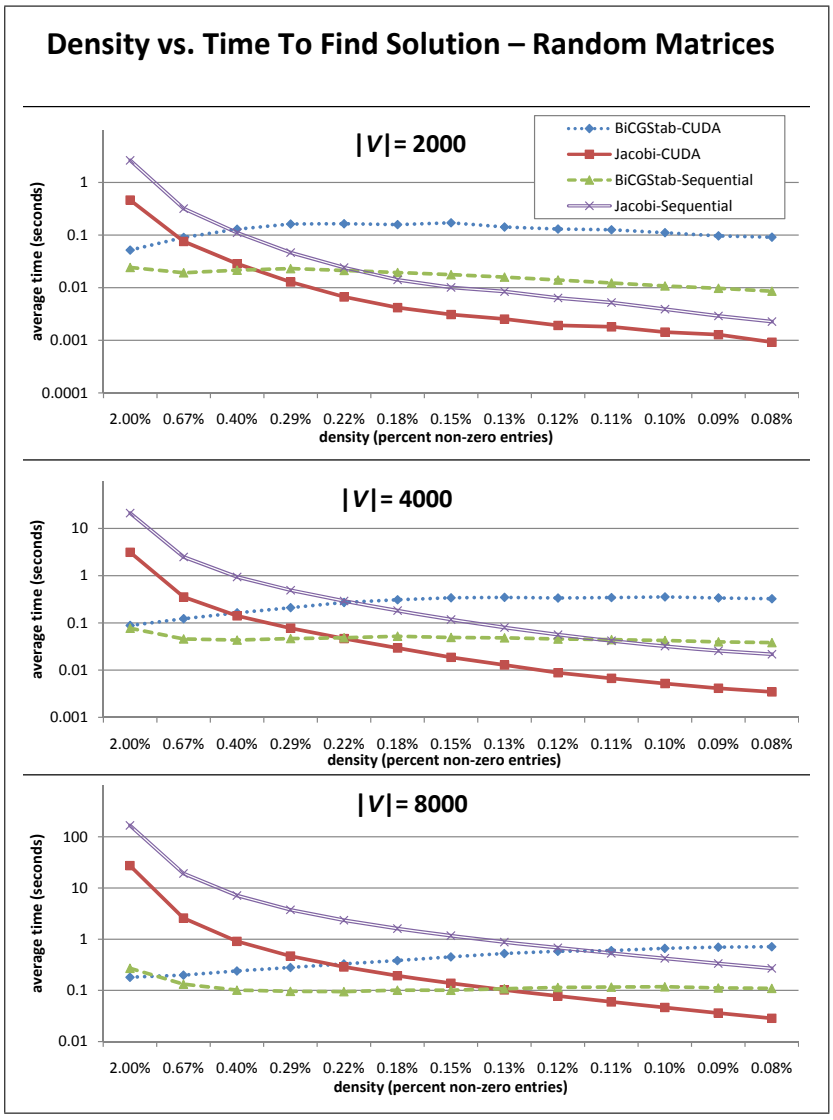


Figure 8.1: The effect of varying density on performance, for three matrix sizes. A logarithmic scale is used on the y-axis. Average times are based on 20 trials. The standard deviations of the times measured are too small to be shown.

for random quick sort, and a strategy called probability first search that prioritizes high-probability transitions, also written by Zhang [28], was used for the biased die example.

The results of the random quick sort tests are shown in Figure 8.5, and the results of the biased die tests in Figure 8.6. Error bars representing standard deviations of each data point are too small to be visible in the graphs. The matrix

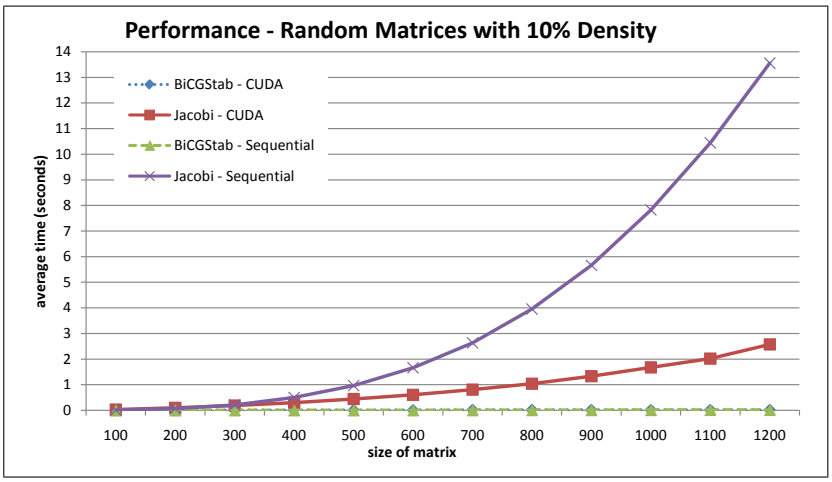


Figure 8.2: Performance on randomized matrices of varying sizes, all with 10% density.

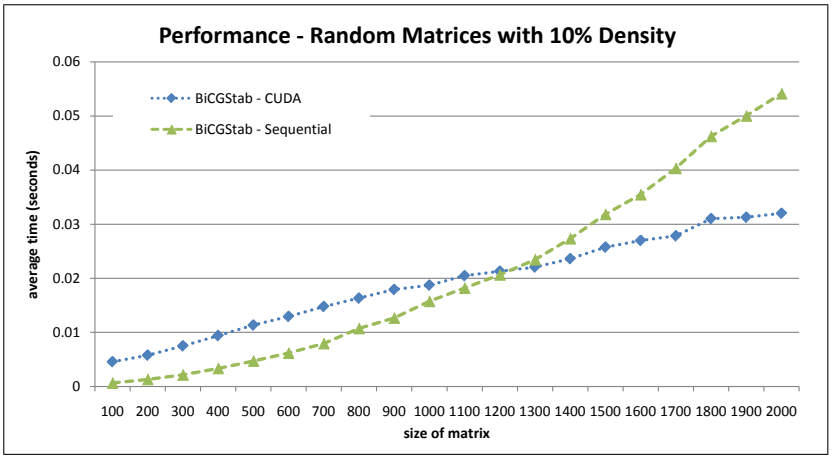


Figure 8.3: BiCGStab performance on randomized matrices of varying sizes, all with 10% density.

sizes in the random quick sort tests are much smaller than those in the biased die tests because, while the matrices for random quick sort are initially the same size as or larger than those produced for biased die, much fewer states belong to the $S_?$ set. Furthermore, note that the densities of these matrices decrease as their sizes increase. The sizes and densities of the matrices used in these tests are shown in Table 8.1.

The relative performances of the sequential and CUDA implementations of the Jacobi method, and the sequential implementation of the BiCGStab method, are

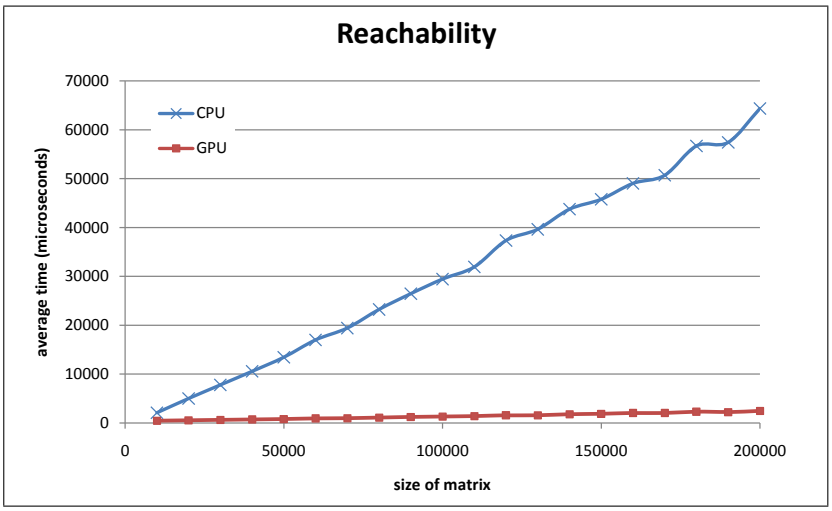


Figure 8.4: Results of GPU reachability algorithm compared to the same algorithm on the CPU. Averages are based on 10 trials. Matrices have 10% non-zero entries.

different for each algorithm. The best performance on the random quick sort data was from the sequential implementation of the Jacobi method, while the best performance for biased die was from the CUDA implementation of the Jacobi method. The CUDA implementation of the BiCGStab method performs worst in both cases.

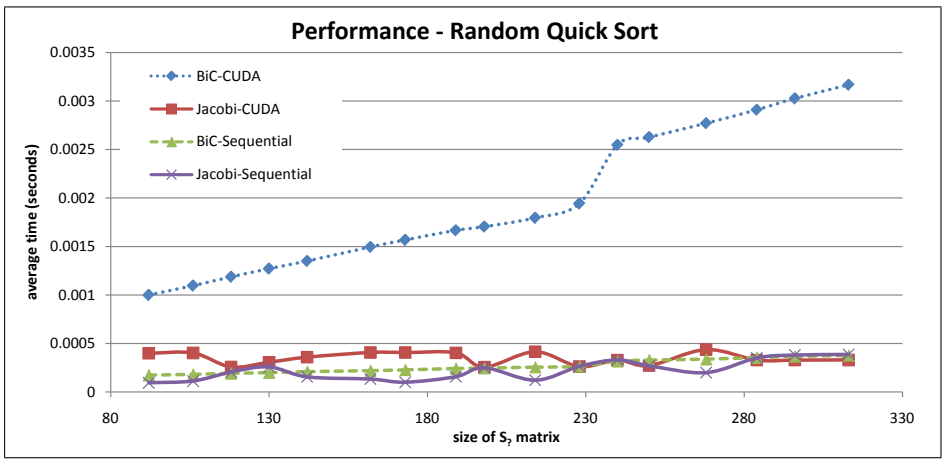


Figure 8.5: Performance on matrices output by JPF, while checking the random quick sort algorithm.

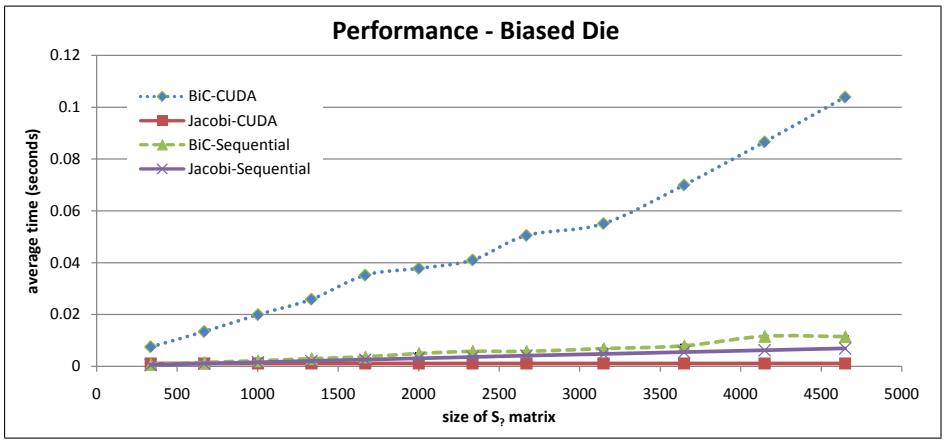


Figure 8.6: Performance on matrices output by JPF, while checking the biased die algorithm.

8.3 Comparing Probabilistic Model Checking Data with Random Matrices

In this section, we compare the performance of the iterative methods on model checking data and random matrices. For these tests, random matrices were generated with the same densities and sizes as those produced by JPF (as in Table 8.1). The JPF matrices are reduced transition probability matrices, subtracted from the identity matrix. So, they have entries in the interval [-1, 1] with locations based on the structure of a PTS. In contrast, entries in the randomized matrices are randomly-located positive integers, as described in Section 8.1. Unlike in the JPF matrix tests, each trial uses a different matrix and vector. However, the standard deviations of the times measured are still too small to be shown on the graphs.

Performance results for these matrices are shown in Figure 8.7 and Figure 8.8. It is apparent that the ordering of the different implementations' performances is the same as it was for matrices of the same sizes and densities generated by JPF. This suggests that size and density are the main determinants of which implementation performs best on probabilistic model checking data, and whether CUDA will be beneficial, rather than other properties unique to the matrices found in probabilistic model checking.

Our results indicate that for the particular types of matrix encountered during probabilistic model checking, implementing the BiCGStab method in CUDA does not improve performance. CUDA does, however, improve the performance of the Jacobi method.

In [6], the authors conjecture that the Jacobi method would be more suitable

Sizes and Densities of $S_?$ Matrices					
Random Quick Sort			Biased Die		
n	m	density	n	m	density
92	211	0.025	667	1333	0.003
118	263	0.019	1333	2665	0.001
142	312	0.015	2000	3999	0.001
173	379	0.013	2668	5335	0.001
198	430	0.011	3647	7293	0.001
228	491	0.009	4647	9293	0.000
250	536	0.009	5647	11293	0.000
284	606	0.008	6647	13293	0.000
313	669	0.007	7647	15293	0.000

Table 8.1: Sizes and densities of the matrices produced by JPF for the two algorithms. n is the matrix dimension, and m is the number of non-zero entries.

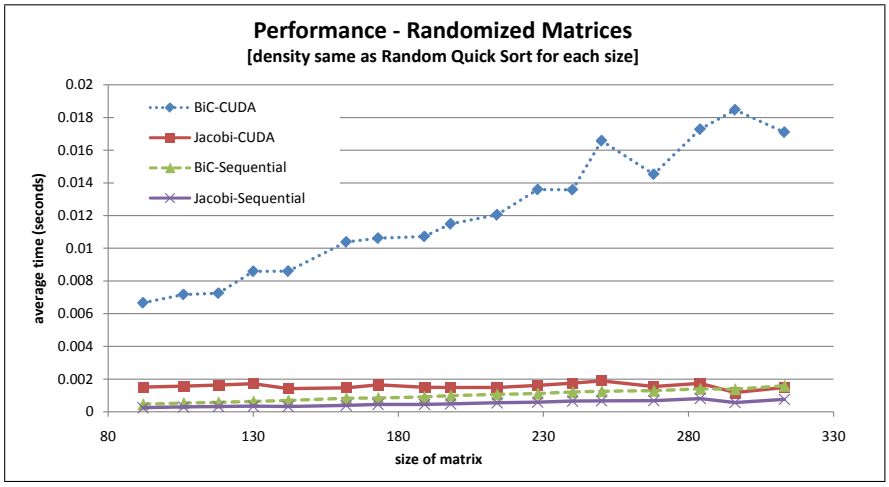


Figure 8.7: Performance on randomized matrices, with the same sizes and densities as the matrices produced by JPF when checking the random quick sort algorithm. Averages times are based on 40 matrices.

for probabilistic model checking using CUDA than Krylov subspace methods such as the BiCGStab method. This seems to be true. However, the results in Section 8.1 suggest that this is due to the superior performance of the Jacobi method on sparse matrices in general, rather than BiCGStab's higher memory requirements as proposed in [6]. For the probabilistic model checking data, the matrix density decreases as the size increases, so the conditions in which the CUDA BiCGStab implementation performs best (larger, denser matrices, as in Figure 8.2) are not

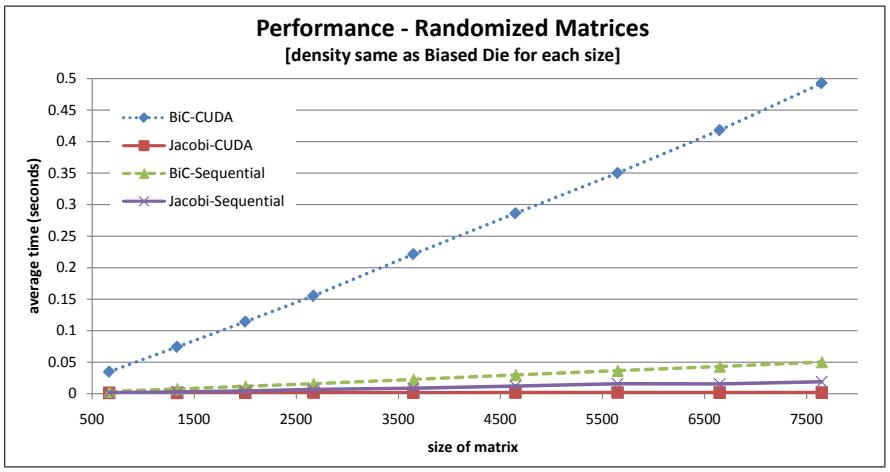


Figure 8.8: Performance on randomized matrices, with the same sizes and densities as the matrices produced by JPF when checking the biased die algorithm. Average times are based on 40 matrices.

encountered.

9 Conclusions

9.1 Summary

In this thesis, we continued work on the progress measure introduced in [30]. We showed how it could be calculated for linear-time properties expressed as unipolar LTL formulas, and we developed a more efficient algorithm to calculate progress for properties of the form $p \,\mathcal{U}\, q$. We also examined the relationships between the progress of properties and their negations, and determined how to calculate a lower bound of progress in polynomial time. Thus, the main contribution of this thesis is to vastly increase the number of situations in which the progress measure can be used.

Furthermore, as progress is calculated as a reachability equation for invariants [30] and $p \,\mathcal{U}\, q$-type properties (Chapter 6), we examined the performance benefits of computing reachability on a GPU. We compared iterative and sequential implementations of the Jacobi and BiCGStab algorithms. In some cases, but not all, the GPU implementation was beneficial. We observed that which algorithm gives the best performance is related to the density of the transition probability matrix of the system under verification. We also demonstrated how binary reachability algorithms on dense graphs could be accelerated using a GPU.

9.2 Related Work

Our work on the progress measure is based on the paper by Zhang and Van Breugel [30], and the thesis of Zhang [28]. These works first introduced the progress measure. They proved that the progress measure is less than or equal to the actual measure of executions in the system under verification that satisfy the linear-time property being verified (Theorem 2.45), and showed how to calculate progress for invariants. They also created the probabilistic extension of JPF which we used to generate test data for Chapter 7.

The work by Pavese, Braberman and Uchitel [20] is also related. They aim to measure the probability that a run of the system reaches a state that has not

been visited by the model checker. Also the work by Della Penna et al. [21] seems related. They show how, given a Markov chain and an integer i, the probability of reaching a particular state s within i transitions can be computed.

Our work on calculating reachability probabilities in CUDA is based on that of Bosnacki et al. [6], which tests a CUDA implementation of the Jacobi method on data obtained by another probabilistic model checker, PRISM. They used different probabilistic algorithms in their probabilistic model checker, and for each algorithm their results show a benefit from GPU usage. In a later expansion of their research [7], they test a CUDA implementation of the Jacobi method that uses a backward segmented scan, and one using two GPUs, which further improve performance. In their second paper they also compare the performance on 32- and 64-bit systems, and find that for one of the three algorithms they model check, the CPU algorithm on the 64-bit system outperforms the CUDA implementation. In another r5 elated paper, Barnat et al. [3] improve the performance of the maximal accepting predecessors algorithm for LTL model checking by implementing it using CUDA.

9.3 Future Work

As we have seen, there seems to be a trade off between efficiency and accuracy when it comes to computing progress. Our most general algorithm to compute progress, the algorithm for unipolar LTL in Chapter 4, is exponential in the size of the unipolar LTL formula and polynomial in the size of the search. We conjecture, and leave to future work to prove, that the problem of computing progress is PSPACE-hard.

The approach to handle the unipolar fragment of LTL seems not to apply to all of LTL. We believe a different approach is needed to create an algorithm that applies to LTL in general. When an atomic proposition and its negation are both included in a formula, situations arise where one must take into account the probabilities of individual paths leading to the unexplored part of the system, and the atomic propositions encountered along those paths, in order to measure progress. So, we speculate that a modified version of an algorithm like the one to measure executions that satisfy a property in [11, Section 3.1] might be useful. We leave the development of a general progress measure algorithm to future research.

Regarding the lower bound algorithm, we hypothesize that Theorem 4.24 is true without the precondition that no violation of the LTL property ϕ has been found. However, proving this theorem without the assumption seems much more difficult, and could be a topic for future work. Furthermore, as we have shown, the lower bound is tight for at least one important class of properties, the invariants. Since the lower bound can be calculated in polynomial time, determining the precise class

of properties for which the bound is tight would be useful, and is another topic for further research.

We also hypothesize that the algorithm to calculate progress for formulas of the form $p \,\mathcal{U}\, q$ in Chapter 6 could be expanded to include formulas with a series of atomic proposition connected by until operators, such as $p_1 \mathcal{U}\, p_2 \mathcal{U} \ldots \mathcal{U}\, q$. However, we have not proven this, and leave it to future research.

Future work on using the GPU for reachability calculations could include experiments on model checking data generated from the probabilistic algorithms used with the model checker PRISM in [6], to allow closer comparison with that work. Another possibility is to implement the CUDA BiCGStab algorithm using multiple GPUs, so that different steps can be run in parallel.

In [19], Kwiatkowska et al. suggest considering the strongly connected components of the underlying digraph of the PTS in reverse topological order. Since the sizes and densities of the matrices corresponding to these strongly connected components may be quite different from the size and density of the matrix corresponding to the PTS, we are interested in seeing whether this approach will favor different implementations.

Bibliography

[1] Robert B. Ash. *Basic Probability Theory*. John Wiley & Sons, 1970.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] Jiri Barnat, Lubos Brim, Milan Ceska, and Tomas Lamr. CUDA accelerated LTL model checking. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems*, pages 34–41. IEEE, December 2009.

[4] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994.

[5] Vladimir I. Bogachev. *Measure Theory*, volume I. Springer, November 2006.

[6] Dragan Bošnački, Stefan Edelkamp, and Damian Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *Proceedings of the 16th SPIN Workshop*, pages 32–49. Springer-Verlag, June 2009.

[7] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Technology Transfer*, 13(1):21–35, August 2011.

[8] Peter Buchholz. Structured analysis techniques for large Markov chains. In *Proceeding of the 2006 Workshop on Tools for Solving Structured Markov Chains*, page 2. ACM, October 2006.

[9] Elise Cormie-Bowins. A comparison of sequential and GPU implementations of iterative methods to compute reachability probabilities. In *Proceedings of the 1st Workshop on Graph Inspection and Traversal Engineering*, April 2012.

[10] Elise Cormie-Bowins and Franck van Breugel. Measuring progress of probabilistic LTL model checking. In *Proceedings of the 10th Workshop on Quantitative Aspects of Programming Languages*. EPTCS, April 2012.

[11] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, July 1995.

[12] Brian A. Davey and Hillary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[13] D. H. Fremlin. *Measure Theory*, volume 1. Torres Fremlin, 2010.

[14] Abhijeet Gaikwad and Ioane M. Toke. Parallel iterative linear solvers on GPU: A financial engineering case. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 607–614. IEEE, February 2010.

[15] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, November 2010.

[16] Boudewijn R. Havertkort. *Performance of Computer Communication Systems*. John Wiley & Sons, 1998.

[17] I. N. Herstein and Irving Kaplansky. *Matters Mathematical*. Chelsea Publishing Co., 2nd edition, 1978.

[18] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

[19] Marta Z. Kwiatkowska, David Parker, and Hongyang Qu. Incremental quantitative verification for Markov decision processes. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 359–370. IEEE, June 2011.

[20] Esteban Pavese, Victor Braberman, and Sebastian Uchitel. My model checker died!: how well did it do? In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, pages 33–40. ACM, May 2010.

[21] Giuseppe Della Penna, Benedetto Intrigila, Igor Melatti, Enrico Tronci, and Marisa Venturini Zilli. Finite horizon analysis of Markov chains with the Murφ verifier. *International Journal on Software Tools for Technology*, 8(4/5):397–409, August 2006.

[22] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2003.

[23] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, pages 429–528. Springer-Verlag, September 1996.

[24] Henk A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, March 1992.

[25] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, pages 327–338. IEEE, October 1985.

[26] Daniel J. Velleman. *How to Prove It: A Structured Approach*. Cambridge University Press, 2nd edition, 2006.

[27] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[28] Xin Zhang. Measuring progress of model checking randomized algorithms. Master's thesis, York University, Toronto, 2010.

[29] Xin Zhang and Franck van Breugel. Model checking randomized algorithms with Java PathFinder. In *Proceedings of 7th International Conference on Quantitative Evaluation of Systems*, pages 157–158. IEEE, September 2010.

[30] Xin Zhang and Franck van Breugel. A progress measure for explicit-state probabilistic model-checkers. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming*, pages 283–294. Springer-Verlag, July 2011.