Efficient Model Checking for Probabilistic Timed Automata

Robert Jørgensgaard Engdahl   Arild Martin Møller Haugstad

12th June 2008

The Faculty of Engineering and Science
Aalborg University

TITLE:
Efficient Model Checking for Probabilistic Timed Automata

SUBJECT:
Formal systems

PROJECT PERIOD:
Dat5, autumn 2007

PROJECT GROUP:
d604a (in 1.1.01)

GROUP MEMBERS:
Robert Jørgensgaard Engdahl
Arild Martin Møller Haugstad

SUPERVISOR:
Kim Guldstrand Larsen

COPIES: 6

PAGES: 99

COMPLETED ON:
12th June 2008

Abstract:
In this thesis, we focus on the modeling formalism networks of probabilistic timed automata. Networks of probabilistic timed automata may be used to model systems where time, i.e. delays and timeouts, concurrency and synchronisation as well as probabilistic behaviour, either from the system or its environment, are all important properties in the model.

We introduce a forward state space exploration and reduction algorithm to partition the state space of probabilistic and normal timed automata according to time-abstract equivalences.

We demonstrate how this partitioning can be used to compute probabilistic reachabilities.

A prototype tool, Uppaal Prob, has been implemented as an extension to the Uppaal Prob tool. This includes the state space exploration and reduction algorithm, refinement according to probabilistic reachability formulae and computation of maximal reachability probabilities.

Preface

The present report is our masters thesis, documenting results of our work during the spring semester 2008, leading to the degree of M.Sc. in computer science at the Department of Computer Science at Aalborg University. Parts of this report is based on our work during the autumn semester 2007. This can be found in [EH07].

The work presented in this report assumes basic knowledge of probability theory, timed automata, operational semantics, discrete mathematics and the real time systems verification tool UPPAAL.

We would like to thank our supervisor professor Kim Guldstrand Larsen for allowing us to work on the UPPAAL code base, for asking relevant and enlighting questions, for providing valuable input and references, and for recommending our project as a project in the Bang & Olufsen Scholarship Programme. We would also like to thank associate professor Alexandre David for always being willing to help us figure out the existing UPPAAL code, and for suggesting various implementation specific design choices.

Additionally we would also like to thank Bang & Olufsen A/S for granting us scholarships, so that we may concentrate fully on our master project. We would especially like to thank Bodil Hviid Steengaard and Karsten Langhoff Sørensen at Bang & Olufsen A/S for showing their interrest in our work, and for recommending our project as a project in the Bang & Olufsen Scholarship Programme.

Robert Jørgensgaard Engdahl

Arild Martin Møller Haugstad

Aalborg, 12th June 2008

Contents

1 Introduction

Real time systems have been modelled in timed automata for about the last two decades.

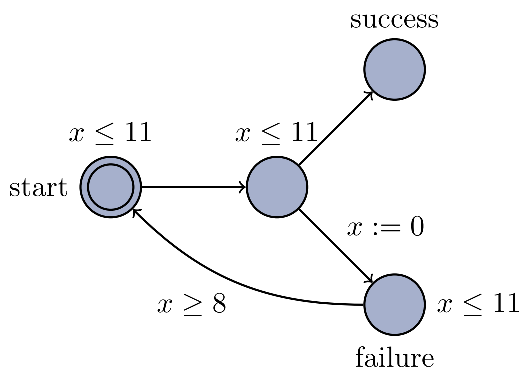


Figure 1.1: The timed automaton $\mathcal{A}$.

In Figure 1.1, the timed automaton $\mathcal{A}$ is given. From the start location, $\mathcal{A}$ can take a transition to the anonymous location, where it can choose nondeterministically between a transition leading to the success state, and one leading to the failure state. When taking the transition leading to the failure state, the clock x is reset, and only when $x \geq 8$ can $\mathcal{A}$ take a transition back to start, where it may try again. In all states except success, time is only allowed to pass until $x = 11$ at which point a transition must be taken, if not sooner.

A timed automaton like $\mathcal{A}$ has no reason ever to fail, since it may always simply choose the shortest path to the success state. However, it is possible never to have success either; $\mathcal{A}$ may loop between start and failure forever.

The choice between failure and success is normally not really a choice: It is more likely a chance of success or a risk of failure, where both the word *chance* and the word *risk* are the key. We would like to express such choices as chance or risk in timed automata.

In Figure 1.2, an analogue of the timed automaton $\mathcal{A}$ is given, namely the probabilistic timed automaton $\mathcal{A}'$. From the start location, $\mathcal{A}'$ can take a transition that with probability $\frac{3}{4}$ will lead to success, and with probability $\frac{1}{4}$, the clock x is reset and the transition will lead to the failure state. While in the failure state, $\mathcal{A}'$ must delay for 8 time units until it can take a transition back to start, where it may try again; just like $\mathcal{A}$. In all states except success, time is only allowed to pass until $x = 11$ at which point a transition must be taken, if not sooner.

A probabilistic timed automaton like $\mathcal{A}'$ will, unlike the timed automaton $\mathcal{A}$, have reasons both to fail and to succeed. The property from $\mathcal{A}$, about it being possible never

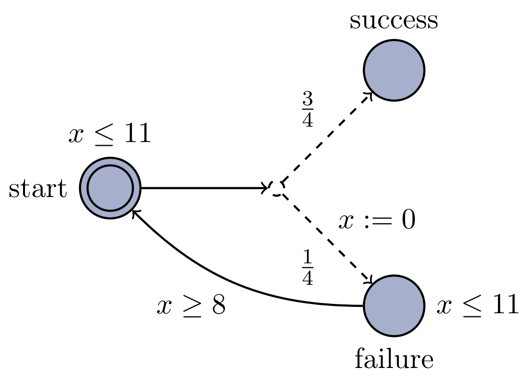


Figure 1.2: The probabilistic timed automaton $\mathcal{A}'$.

to have success, has a slightly more informative analogue in $\mathcal{A}'$, namely it being possible, but highly unlikely, never to have success; the event "never to have success" involves looping from start to failure back to start for ever, and the probability of that is

$$\Pr(\text{“never to have success”}) = \lim_{n \to \infty} \frac{1}{4^n} = 0$$

We call such a property a soft property of a real time system.

Real time model checkers, such as Uppaal, are used to check hard properties such as "is it possible never to have success" and "is it possible to always have success" for timed automata like $\mathcal{A}$.

In this project, we extend Uppaal so that it is able to check soft properties such as "how likely is it eventually to have success" for probabilistic timed automata like $\mathcal{A}'$. We call this extension Uppaal Prob.

Analysis of timed automata is normally done using convex zones of clock valuations. An example of a convex zone of clock valuations is $\{u \mid 11 \geq u(x) \geq 8\}$, where u are functions from clocks into $\mathbb{R}$ called clock valuations. This symbolic representation is necessary since there are uncountable many clock valuations.

When analysing probabilistic timed automata, computational problems arise that were not an issue for timed automata. In particular, there may be two different paths leading to some goal state, and we need to know where these overlap to know how their probabilities contribute. In other words, we will need a forward stable partitioning of relevant parts of the state space. The partitions that represent overlaps of convex zones, i.e. intersections, are themselves convex zones, but those remaining, i.e. subtractions, need not to be convex.

In this project, we show how those remainder sets of clock valuations may be represented using federations — unions of convex zones — and how to construct a relevant forward stable partitioning of the state space of a given network of probabilistic timed automata.

We also show how to calculate the maximum probability of reaching a given class of symbolic states in such a partitioning. In particular, we have implemented this functionality in Uppaal Prob.

1.1 Related Work

Time abstract bisimulation[1] is often used in the analysis of timed automata. A well known, easy to get, bisimulation quotient is the region graph. The region graph is a bit stronger than all other time abstract bisimulations in that all configurations in an augmented region (a symbolic state) in the region graph satisfy the same PTCTL[2]

Region graphs are always exponential in the number of components in a given network of (probabilistic) timed automata, so analysis on region graphs quickly becomes intractable. Nevertheless, Kwiatkowska et al. gives an algorithm for model checking probabilistic timed automata against PTCTL formulae using region graphs in [KNSS02]. We believe that similar algorithms may be applied on bisimulation quotients, say ρ, given bisimulation classes are either fully satisfying or fully violating each atomic proposition being part of the given PTCTL formula. This is why time abstract bisimulation quotient construction is interesting.

In [BFHR92], Bouajjani et al. gives a minimal graph generation algorithm that generates a bisimulation quotient ρ of the reachable state space from an initial state space partitioning ρ_0; this algorithm works on any transition system. The bisimulation quotient ρ is minimal with respect to ρ_0. The real contribution of [BFHR92] is bisimulation quotient generation on the fly while doing forward exploration.

In [ACH$^+$92], Alur et al. adapts the minimal graph generation algorithm by [BFHR92] for timed automata, at the expense of minimality; nevertheless, the output is a time abstract bisimulation quotient which is potentially much coarser than the region graph. As in [BFHR92], the time abstract bisimulation quotient of [ACH$^+$92] is generated on the fly while doing forward exploration.

In [EH07], we made the time abstract bisimulation quotient of [ACH$^+$92] minimal again, with respect to certain initial state space partitionings, namely those that do not make abstractions of the discrete part of the state space. As in [BFHR92] the time abstract bisimulation quotient of [EH07] is generated on the fly while doing forward exploration.

[1]A bisimulation where the length of delays are ignored. For a probabilistic analogue see Definition 5.1.2

[2]Probabilistic Timed Computation Tree Logic. There are a few variants of this logic in the literature. But the main idea seem to be some kind of CTL with atomic propositions involving clocks and formulae involving probability bounds, not to forget maximum and minimum probability formulae.

Part I

Preliminaries

In this part, we establish the preliminary theory about probabilities, probabilistic reachability and probabilistic timed automata. Many of these definitions and theorems may also be found in [EH07].

2 Probabilities

2.1 The Axioms of Probability

In Section 2.2 we will need at least the basics of probability theory. In this subsection we introduce probability theory in terms of the axioms of probability. For a more thorough introduction to probability theory, we can recommend [Olo05].

Definition 2.1.1 (Events and Sample Spaces)
Let S be a sample space of a random experiment. An event of S is a subset of S. The set of all events of S is written $\mathcal{P}(S)$. □

Definition 2.1.2 (Axioms of Probability)
Let S be a sample space of a random experiment. A probability measure is a function $\Pr\colon \mathcal{P}(S) \to [0;1]$ that maps events into probabilities such that

$$\Pr(S) = 1$$

and for pair wise disjoint events $A_1, A_2, \ldots \subseteq S$ we have

$$\Pr\left(\bigcup_{k=1}^{\infty} A_k\right) = \sum_{k=1}^{\infty} \Pr(A_k)$$ □

2.2 Random Variables

In the following we define random variables so that we may later specify the semantics of probabilistic timed automata in terms of these[1].

Definition 2.2.1 (Random Variable)
Let S be a sample space. A random variable is a function $X\colon S \to \mathcal{X}$.
The set of all random variables with codomain $\mathcal{X}$ is denoted by $V(\mathcal{X})$. □

Let X be a random variable and S be a sample space. When an event $s \in \mathcal{P}(S)$ is unimportant in the given context we write X as notation for $X(s)$.

Definition 2.2.2 (The Distribution of a Random Variable)
Let X be a random variable with codomain $\mathcal{X}$. The distribution of X is a function $p : \mathcal{X} \to [0;1]$, such that $p(x) = \Pr(X = x)$. □

[1]When specifying the semantics of stochastic systems, two different formalisms can be considered; σ-algebra and random variables. These are equivalent, but due to more reader friendly syntax, we use random variables

Let X be a random variable. If p is the distribution of X we write $X \sim p$.

To give a random variable $X \sim p$ with codomain $\mathcal{X}$ of size n explicitly, we write

$$\{x_{1_{p_1}}, \ldots, x_{n_{p_n}}\},$$

where $x_1, \ldots, x_n \in \mathcal{X}$ and $p_i = p(x_i)$ for $i = 1, \ldots, n$.

Definition 2.2.3 (Probability Distribution)
Let $\mathcal{X}$ be a set. A probability distribution on $\mathcal{X}$ is a function $p\colon \mathcal{X} \to [0;1]$, where

$$\sum_{x \in \mathcal{X}} p(x) = 1.$$

The set of all probability distributions on finite subsets of $\mathcal{X}$ is written $P(\mathcal{X})$. □

2.3 Labelled Markov Transition Systems

In the following we define labelled Markov transition systems so that we may later specify the semantics of probabilistic timed automata in terms of these[2].

Recall that a partial function $f\colon A \rightharpoonup B$ is a function except that is not well defined on all of it's domain A.

Definition 2.3.1 (Labelled Markov Transition System)
A labelled Markov transition system is a tuple $(\Gamma, A, \to)$, where Γ is a set of configurations and A is a set of labels (or actions) and $\to\colon \Gamma \times A \rightharpoonup V(\Gamma)$ is the transition function. □

Given a labelled Markov transition system $(\Gamma, A, \to)$, we shall write $x \xrightarrow{a} X$ for

$$\to (x, a) = X,$$

where $x \in \Gamma$, $X \in V(\Gamma)$ and $a \in A$, and $\to (x, a)$ is well defined.

[2]In computer science, we prefer discrete Markov chains with the property that the probability distribution of the next configuration (the target of a probabilistic transition) is a function of the current configuration, but not the number of transitions taken so far. Such discrete Markov chains are called time homogeneous discrete Markov chains. A labelled Markov transition system is a hybrid of time homogeneous discrete Markov chains and labelled transition systems.

3 Probabilistic Reachability for Labelled Markov Transition Systems

3.1 Adversary

Definition 3.1.1 (Path)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system. A path ω of $\mathcal{M}$ is a tuple $\omega = (x_0, \{(a_n, x_n)\}_{n\geq 1})$, where $\{(a_n, x_n)\}_{n\geq 1}$ is a sequence of actions $a_n \in A$ and configurations $x_n \in \Gamma$, such that $x_n \xrightarrow{a_{n+1}} X$, where $x_{n+1} \in \operatorname{img}(X)$ for $n \geq 0$.
The set of all paths of $\mathcal{M}$ is denoted $\Omega(\mathcal{M})$. □

Definition 3.1.2 (Adversary)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system. An adversary of $\mathcal{M}$ is a partial function $f\colon \Omega \rightharpoonup A$, where $\rightarrow$ is well-defined on $(x_k, f(\omega))$ for all paths $\omega = (x_0, \{(a_n, x_n)\}_{k\geq n\geq 1})$ where $f(\omega)$ is well defined.
We write $\mathcal{F}(\mathcal{M})$ for the set of all adversaries of $\mathcal{M}$. □

Definition 3.1.3 (Memoryless Adversary)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system, and let f be an adversary of $\mathcal{M}$. If there exists a partial function $g\colon \Gamma \rightharpoonup A$ such that

$$g(x_k) = f(x_0, \{(a_n, x_n)\}_{k\geq n\geq 1}) \quad \text{for all } k \text{ where } f \text{ is well defined,}$$

for all paths $(x_0, \{(a_n, x_n)\}_{k\geq n\geq 1})$ in $\Omega(\mathcal{M})$, we say that f is memoryless. □

If f is a memoryless adversary, we may write $f(x_k)$ instead of $f(x_0, \{(a_n, x_n)\}_{k\geq n\geq 1})$.

Definition 3.1.4 (Run)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system, let f be an adversary of $\mathcal{M}$, and let $\omega = (x_0, \{(a_n, x_n)\}_{n\geq 1})$ be a path of $\mathcal{M}$. We say that ω is a run of f if

$$a_{k+1} = f(x_0, \{(a_m, x_m)\}_{k\geq m\geq 1}),$$

for each $k \geq 0$. □

3.2 Stochastic Process of an Adversary

Definition 3.2.1 (Stochastic Process)
A stochastic process is a sequence $\{X_n\}_{n\geq 1}$ of random variables. □

Definition 3.2.2 (The Stochastic Process of an Adversary)
Let $\mathcal{M} = (\mathcal{X}, A, \to)$ be a labelled Markov transition system, and let f be an adversary of $\mathcal{M}$. A stochastic process of f and $x_0 \in \mathcal{X}$ is a sequence $\{f_n\}_{n>0}$ of random variables with codomain $(A \times \mathcal{X})$ with

$$f_n = (a_n, X_n) \quad \text{where } a_n = f(x_0, \{f_k\}_{k<n}),\ X_{n-1} \xrightarrow{a_n} X_n,\ X_0 = x_0. \qquad \square$$

Definition 3.2.3 (Markov Chain)
Let $\{X_n\}_{n\geq 1}$ be a stochastic process. If the Markov property

$$\Pr\left(X_n = x_n \mid X_{n-1} = x_{n-1}\right) = \Pr\left(X_n = x_n \mid X_{n-1} = x_{n-1}, \ldots, X_1 = x_1\right) \tag{3.1}$$

holds for all $n > 1$ then $\{X_n\}_{n\geq 1}$ is called Markov chain. $\square$

Theorem 3.2.4
If f is a memoryless adversary, then the stochastic process $\{f_n\}_{n\geq 1}$ is a Markov chain.

Proof
Proof by induction over the length of $\{f_n\}_{n\geq 1}$.

Ad $\{f_n\}_{n=1}$ is a Markov chain: By Definition 3.2.3 all sequences of random variables with one element is a Markov chain

Ad $\{f_n\}_{2\geq n\geq 1}$ is a Markov chain: The Markov property (3.1) is trivially true for $n = 2$, and by $\{f_n\}_{n=1}$ is a Markov chain we have that $\{f_n\}_{2\geq n\geq 1}$ is a Markov chain.

Ad $\{f_n\}_{k\geq n\geq 1}$ is a Markov chain: Assume by induction that $\{f_n\}_{k'\geq n\geq 1}$ is a Markov chain for all $k' < k$. It is enough to show (3.1) holds for $n = k$.

For any sequence of events $\{f_i = (a_i, x_i)\}_{k-1\geq i\geq 1}$ it follows from Definition 3.2.2 and

Definition 3.1.3 that

$$\begin{aligned}
&\Pr\left(f_k = (a_k, x_k) \,\middle|\, \begin{array}{c} f_{k-1} = (a_{k-1}, x_{k-1}) \\ \vdots \\ f_1 = (a_1, x_1) \end{array} \right) \\
&= \Pr\left(f(x_0, \{a_i, x_i\}_{k-1 \geq i \geq 1}) = (a_k, x_k) \,\middle|\, \begin{array}{c} f_{k-1} = (a_{k-1}, x_{k-1}) \\ \vdots \\ f_1 = (a_1, x_1) \end{array} \right) \\
&= \Pr\left(f(x_{k-1}) = (a_k, x_k) \,\middle|\, \begin{array}{c} f_{k-1} = (a_{k-1}, x_{k-1}) \\ \vdots \\ f_1 = (a_1, x_1) \end{array} \right) \\
&= \Pr\left(f(x_{k-1}) = (a_k, x_k) \,\middle|\, \begin{array}{c} f_{k-1} = (a_{k-1}, x_{k-1}) \\ f_{k-2} = (a'_{k-2}, x'_{k-2}) \\ \vdots \\ f_1 = (a'_1, x'_1) \end{array} \right) \quad \begin{array}{l} \text{for all sequences} \\ \{f_i = (a'_i, x'_i)\}_{k-2 \geq i \geq 1} \\ \text{of events.} \end{array} \\
&= \Pr\left(f(x_{k-1}) = (a_k, x_k) \mid f_{k-1} = (a_{k-1}, x_{k-1}) \right) \\
&= \Pr\left(f_k = (a_k, x_k) \mid f_{k-1} = (a_{k-1}, x_{k-1}) \right),
\end{aligned}$$

which is (3.1) for $n = k$. ■

3.3 Probabilistic Reachability for Labelled Markov Transition Systems

Definition 3.3.1 (Atomic Proposition)
Let $\{X_n\}_{n\geq 1}$ be a stochastic process where $\text{img}(X_n) \subseteq \mathcal{X}$ for all n. Any subset $B \subseteq \mathcal{X}$ is an atomic proposition of $\{X_n\}_{n\geq 1}$. □

Example 3.3.2
Let $\mathcal{M} = (\mathcal{X}, A, \rightarrow)$ be a labelled Markov transition system, let x_0 be an initial configuration for $\mathcal{M}$, and let f be an adversary of $\mathcal{M}$. By Definition 3.3.1 it follows that $B \subseteq A \times \mathcal{X}$ is an atomic proposition of f_n, where f_n is the stochastic process of f and x_0. □

With the same definitions as in Example 3.3.2 we will write $B \subseteq \mathcal{X}$ as notation for $A \times B \subseteq A \times \mathcal{X}$, when we are only interested in configurations.

Definition 3.3.3
Let $A \subseteq \mathcal{X}$ be a set of states of a stochastic process $\{X_n\}_{n\geq 1}$, where $\mathrm{img}(X_n) \subseteq \mathcal{X}$ for $n \geq 1$. The *probability of* $\{X_n\}_{n\geq 1}$ *ever reaching* A is written

$$\Pr\left(\bigcup_{n>1}\{X_n \in A \mid X_{n-1}, \ldots, X_1 \notin A\}\right). \qquad \square$$

Remark 3.3.4
By event inclusion we have that

$$\bigcup_{n>1}\{X_n \in A \mid X_{n-1}, \ldots, X_1 \notin A\} = \bigcup_{n>1}\{X_n \in A\}. \qquad \square$$

In what follows we will need to range over different stochastic processes instantiated by a given labelled Markov transition system, an initial configuration and an adversary. So we define some syntax to make this more readable:

Definition 3.3.5
Let $\mathcal{M} = (\mathcal{X}, A, \rightarrow)$ be a labelled Markov transition system, let x_0 be an initial configuration for $\mathcal{M}$, let f be an adversary of $\mathcal{M}$, and let $B \subseteq A \times X$ be an atomic proposition. We write

$$\{\mathcal{M}, x_0 \models f\colon f_n \in B\}$$

for the event

$$\bigcup_{n>1}\{f_n \in B\}$$

where f_n is the stochastic process of f and x_0 $\square$

Corollary 3.3.6
Let $\mathcal{M} = (\mathcal{X}, A, \rightarrow)$ be a labelled Markov transition system, let x_0 be an initial configuration for $\mathcal{M}$, let f be an adversary of $\mathcal{M}$, and let f_n be the stochastic process of f and x_0. Clearly $\mathrm{img}(f_n) \subseteq A \times \mathcal{X}$ for all $n \geq 1$. Therefore any subset $B \subseteq A \times \mathcal{X}$ is a valid atomic proposition for f_n. $\square$

Definition 3.3.7 (Probability of the Best Adversary)
Let $\mathcal{M} = (\mathcal{X}, A, \rightarrow)$ be a labelled Markov transition system, let x_0 be an initial configuration of $\mathcal{M}$ and let $B \subseteq A \times \mathcal{X}$ be an atomic proposition. We define the probability of the best adversary for reaching B as

$$\max_{f\in\mathcal{F}(\mathcal{M})} \Pr\left(\mathcal{M}, x_0 \models f\colon f_n \in B\right), \tag{3.2}$$

where the adversary f that maximises (3.2) is called the best adversary for reaching B. $\square$

Definition 3.3.8 (Probability of the Worst Adversary)
Let $\mathcal{M} = (\mathcal{X}, A, \to)$ be a labelled Markov transition system, let x_0 be an initial configuration of $\mathcal{M}$ and let $B \subseteq A \times \mathcal{X}$ be an atomic proposition. We define the probability of the worst adversary for reaching B as

$$\min_{f \in \mathcal{F}(\mathcal{M})} \Pr(\mathcal{M}, x_0 \models f \colon f_n \in B), \tag{3.3}$$

where the adversary f that minimises (3.3) is called the worst adversary for reaching B.□

The following theorem states that a memoryless greedy adversary is the best.

Theorem 3.3.9
Let $\mathcal{M} = (\mathcal{X}, A, \to)$ be a labelled Markov transition system, let x_0 be an initial configuration of $\mathcal{M}$ and let $B \subseteq A \times \mathcal{X}$ be an atomic proposition for all f_n, where $f \in \mathcal{F}(\mathcal{M})$.

Let g be the memoryless adversary defined by $g(x) = a$, where $x \xrightarrow{a} X$ such that the expected supreme reachability probability

$$\sum_{x_0' \in \operatorname{img}(X)} \sup_{h \in \mathcal{F}(\mathcal{M})} \Pr\left(\{\mathcal{M}, x_0' \models h \colon h_n \in B\}, \{X = x_0'\}\right), \tag{3.4}$$

is maximal. Then

$$\Pr(\mathcal{M}, x_0 \models g \colon g_n \in B) = \max_{f' \in \mathcal{F}(\mathcal{M})} \Pr\left(\mathcal{M}, x_0 \models f' \colon f_n' \in B\right). \tag{3.5}$$

Proof
We have that

$$\Pr(\mathcal{M}, x_0 \models g \colon g_n \in B) = \sum_{x_0' \in \operatorname{img}(X)} \sup_{h \in \mathcal{F}(\mathcal{M})} \Pr\left(\{\mathcal{M}, x_0' \models h \colon h_n \in B\}, \{X = x_0'\}\right),$$

where $x_0 \xrightarrow{g(x_0)} X$. Since we can construct an adversary h' such that $x_0 \xrightarrow{h'(x_0)} X$ and h' behaves like the supreme h for each $x_0' \in \operatorname{img}(X)$, we can safely set h' to start from x_0 and marginalise X inside the supremum expression.

$$\begin{aligned} \Pr(\mathcal{M}, x_0 \models g \colon g_n \in B) &= \sup_{h' \in \mathcal{F}(\mathcal{M})} \sum_{x_0' \in \operatorname{img}(X)} \Pr\left(\{\mathcal{M}, x_0 \models h' \colon h_n' \in B\}, \{X = x_0'\}\right) \\ &= \sup_{h' \in \mathcal{F}(\mathcal{M})} \Pr\left(\mathcal{M}, x_0 \models h' \colon h_n' \in B\right) \end{aligned} \tag{3.6}$$

Since there exists an adversary having the reachability probability of (3.6), (namely g), the supremum is a maximum, and (3.5) follows. ■

Corollary 3.3.10
By Theorem 3.3.9 it follows that (3.2) *is well defined.* □

Corollary 3.3.11
By a similar procedure as in Theorem 3.3.9 it can be shown that (3.3) *is also well defined.*□

4 Probabilistic Timed Automata

4.1 Probabilistic Timed Automata

Definition 4.1.1
Let C be a set of clocks. The clock constraints on C are defined as

$$B(C) \quad ::= \quad c_1 - c_2 \sim k \mid c_1 \sim k' \mid k' \sim c_1,$$

where $c_1, c_2 \in C$, $\sim \in \{<, \leq\}$, and $k \in \mathbb{Z}$, $k' \in \mathbb{Z}^+$. □

Definition 4.1.2 (Clock Valuation)
Let C be a set of clocks, then a function $u\colon C \to \mathbb{R}^+$ is called a clock valuation. The set of all clock valuations on C is written $\mathcal{U}(C)$. □

Definition 4.1.3 (Zone)
Let $Z \subseteq \mathcal{U}(C)$ be a set of clocks. If there exists a $z \in 2^{B(C)}$ such that for all $u \in \mathcal{U}(C)$ we have $u \in Z$ if and only if u satisfies all clock constraints of z we call Z a zone.

The set of all zones is denoted $\mathcal{Z}$. □

Definition 4.1.4 (Probabilistic Timed Automata)
A probabilistic timed automata is a tuple (L, l_0, C, A, E, I), where

i) L is a finite set of *locations*,

ii) $l_0 \in L$ is the *initial location*,

iii) C is a finite set of *clocks*,

iv) A is a finite set of *actions*, *co-actions* and the *internal* τ*-action*,

v) $E \subseteq L \times A \times \mathcal{Z} \times P(2^C \times L)$ is a finite set of *edges* from locations with an action, a guard and a probability distribution for the target set of clocks to be assigned and the target location.

vi) $I : L \to \mathcal{Z}$ assigns *invariants* to locations. □

Let (L, l_0, C, A, E, I) be a probabilistic timed automaton and let (l, a, g, p) be an edge in E. We always assume that if a clock valuation $u \in \mathcal{U}(C)$ satisfies g then the clock valuation $u' = [r \mapsto 0]u$ satisfies the invariant of l' for all (r, l') in the domain of p.

4.2 The Semantics of Probabilistic Timed Automata

Definition 4.2.1 (Configuration of Probabilistic Timed Automata)
Let $T = (L, l_0, C, A, E, I)$ be a probabilistic timed automaton. A configuration of T is a tuple (l, v) where $l \in L$ is a location and $v\colon C \to \mathbb{R}^+$ is a clock valuation. The initial configuration of T is (l_0, u_0), where $u_0(c) = 0$ for all $c \in C$. □

Definition 4.2.2 (Target Random Variable)
Let $T = (L, l_0, C, A, E, I)$ be a probabilistic timed automaton, let $p \in P(2^C \times L)$ be a probability distribution, and let $u \in \mathcal{U}(C)$ be a clock valuation. The (target) random variable of u and p is then defined as

$$X(u,p) = \left\{ (l', u')_\mu \;\middle|\; \mu = \sum_{r \mid u' = [r \mapsto 0]u} p(r, l'), \quad l' \in L, \quad u' \in \mathcal{U}(C) \right\}. \qquad \square$$

Definition 4.2.3 (Semantics of Probabilistic Timed Automata)
Let $T = (L, l_0, C, A, E, I)$ be a probabilistic timed automaton. The operational semantics is given by the labelled Markov transition system $(\Gamma, A', \to)$, where $\Gamma = L \times \mathcal{U}(C)$, $A' = A \cup \mathbb{R}^+$, and $\to$ is the least partial function from $\Gamma \times A'$ to $V(\Gamma)$ satisfying the following rules:

$$\frac{[u, d+u] \subseteq I(l)}{(l,u) \xrightarrow{d} \{(l, u+d)_1\}} \qquad \text{(DELAY)}$$

$$\frac{(l,a,g,p) \in E \qquad u \in g}{(l,u) \xrightarrow{a} X(u,p)} \qquad \text{(SWITCH)}$$

□

4.3 Networks of Probabilistic Timed Automata

Definition 4.3.1 (Network of Probabilistic Timed Automata)
A network of probabilistic timed automata is a sequence of probabilistic timed automata $\{A_i\}_{1 \le i \le n}$ where $A_i = (L_i, l_{0_i}, C, A, E_i, I_i)$ is a probabilistic timed automaton for each $1 \le i \le n$. □

4.4 The Semantics of Networks of Probabilistic Timed Automata

Definition 4.4.1 (Configuration of Networks of Probabilistic Timed Automata)
Let $\{A_i\}_{1 \le i \le n}$ be a network of probabilistic timed automata, with clock set C. A configuration of $\{A_i\}_{1 \le i \le n}$ is a tuple $(\bar{l}, v)$, where $l_i \in L_i$ for $i = 1, \ldots, n$ and $u \in \mathcal{U}(C)$ is a clock valuation. The initial configuration of $\{A_i\}_{1 \le i \le n}$ is $(\bar{l}_0, u_0)$ where each entry l_{0_i} in $\bar{l}_0$ is the initial location of A_i and $u_0(c) = 0$ for all $c \in C$. □

Definition 4.4.2
Let $\{A_i\}_{0\leq i\leq n}$ be a network of probabilistic timed automata where

$$A_i = (L_i, l_{0_i}, C, A, E_i, I_i) \quad \text{for i=1,\dots,n,}$$

and let $Y \sim p$ be a random variable with codomain $L_1 \times \cdots \times L_n \times \mathcal{U}(C)$. The invariant map of Y is then defined as

$$I(Y) = \left\{(\bar{l}, u) \in \text{img}(Y) \mid u \in I_i(l_i) \text{ for } i = 1, \dots, n\right\}. \qquad \square$$

Definition 4.4.3 (Semantics of Network of Probabilistic Timed Automata)
Let $\{A_i\}_{1\leq i\leq n}$ be a network of probabilistic timed automata. The operational semantics is given by the labelled Markov transition system $(\Gamma, A', \rightarrow)$, where $\Gamma = L_1 \times \cdots \times L_n \times \mathcal{U}(C)$ are the configurations, L_i is the location set of A_i, $A' = A \cup \mathbb{R}^+$ are the actions, and $\rightarrow$ is the least partial function from $\Gamma \times A'$ to $V(\Gamma)$ satisfying the following rules:

$$\frac{(l_1, u) \xrightarrow{d} \{(l_1, u+d)_1\} \qquad \cdots \qquad (l_n, u) \xrightarrow{d} \{(l_n, u+d)_1\}}{(\bar{l}, u) \xrightarrow{d} \{(\bar{l}, u+d)_1\}} \quad \text{where} \quad d \in \mathbb{R}^+ \qquad \text{(DELAY)}$$

$$\frac{(l_i, u) \xrightarrow{\tau} X \qquad \text{img}(Y) = I(Y)}{(\bar{l}, u) \xrightarrow{\tau} Y} \quad \text{where} \quad Y = \left\{\left(\bar{l}\{l/l_i\}, u\right)_\mu \middle| (l, u)_\mu \in X\right\} \qquad \text{(TAU)}$$

$$\frac{\begin{matrix}(l_i, u) \xrightarrow{c!} X \qquad (l_j, u) \xrightarrow{c?} X' \\ i \neq j \qquad \text{img}(Y) = I(Y)\end{matrix}}{(\bar{l}, u) \xrightarrow{c} Y} \quad \text{where} \quad \begin{aligned} Y &= \left\{\left(\bar{l}\{l/l_i, l'/l_j\}, u'\right)_{\mu\cdot\mu'} \middle| \begin{matrix}(l, u_i)_\mu \in X, \\ (l', u_j)_{\mu'} \in X'\end{matrix}\right\} \\ u'(c) &= \begin{cases} u(c) & \text{if } u_i(c) = u_j(c) = u(c) \\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

(SYNC)

$\square$

Part II

Algorithms

In this part, we give algorithms used for probabilistic reachability analysis.

As we show in Chapter 6, probabilistic reachability analysis can be reduced to linear programming. In order to do this, however, a forward stable partitioning of the state space must be computed.

Computing a forward stable state space for timed automata is a major task, previously engaged by Alur et al. in [ACH$^+$92], and later by the authors in [EH07]. In Chapter 5, we reengage this task.

5 State Space Exploration and Reduction for Probabilistic Timed Automata

In [ACH$^+$92], Alur et al. gave a forward stable state space generation algorithm for timed automata. The symbolic states of [ACH$^+$92] was given by augmented convex zones. In [EH07], we noted that the convexity of zones was only a requirement in the direction of time, and we thus introduced time convexity, and rewrote the forward stable state space algorithm to use time convex federations instead of convex zones, resulting in a possible coarser state space. In this chapter, we give an abstract state space exploration and reduction algorithm in sections 5.1, 5.2 and 5.3, along with some results about it — and in Section 5.5 and 5.6 we modify this algorithm slightly, while giving concrete details for implementation.

5.1 State Space Exploration and Reduction Algorithm I

In [EH07], we adapted the minimisation algorithm for finite transition systems by Bouajjani et al. [BFHR92] to reduce the state space of networks of probabilistic timed automata.

This has previously been done by Alur et al. in [ACH$^+$92]. That approach, however, featured symbolic states with single fully convex zones, and inherently the resulting state space was not minimal, and we showed this in [EH07]. Our approach is not minimal either, however, its at least as small as that of [ACH$^+$92], with equality as worst case.

Definition 5.1.1 (Equivalent Random Variables)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system, let $\equiv$ be an equivalence relation on Γ, and let $X, X' \in V(\Gamma)$. X and X' are equivalent with respect to $\equiv$, written $X \equiv X'$, if

$$\Pr\left(X \in S\right) = \Pr\left(X' \in S\right) \quad \text{for all } S \in \Gamma/\equiv$$

□

Definition 5.1.2 (Time Abstract Markov Equivalence)
Let $M = (\Gamma, A \cup \mathbb{R}^+, \rightarrow)$ be a labelled Markov transition system. An equivalence relation $\equiv$ on Γ is a time abstract Markov equivalence if for all $x, y \in \Gamma$ where $x \equiv y$ it holds that

i) If $x \xrightarrow{a} X$ then there exists $Y \in V(\Gamma)$, and $d \in \mathbb{R}^+$ such that $y \xrightarrow{d}\xrightarrow{a} Y$ and $X \equiv Y$

ii) If $x \xrightarrow{d} X$ then there exists $Y \in V(\Gamma)$ and $d' \in \mathbb{R}^+$ such that $y \xrightarrow{d'} Y$ and $X \equiv Y$.

for all $a \in A$ and all $d \in \mathbb{R}^+$. □

Definition 5.1.3 (Partitions)
Let Γ be a set. A subpartitioning α of Γ is a set of disjoint subsets of Γ. If the union

$$\bigcup_{S \in \alpha} S = \Gamma,$$

we call α a partitioning of Γ. The elements of α are called partitions. $\square$

We use α, σ and ρ to range over (sub)partitionings.

Definition 5.1.4 (Symbolic State)
Let $M = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system. A partitioning ρ of Γ is called a symbolic state space and each $S \in \rho$ is called a symbolic state. $\square$

Example 5.1.5
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system, and let $\equiv$ be a time abstract Markov equivalence on Γ. The quotient set $\Gamma/\equiv$ is a symbolic state space. $\square$

Definition 5.1.6 (Time Abstract Markov Transition)
Switch: There is a time abstract Markov switch transition from (s,u) to X if there exists a $d \in \mathbb{R}^+$ such that $(s,u) \xrightarrow{d} \{(s, u+d)_1\} \xrightarrow{a} X$ for some action a.

Delay: There is a time abstract delay transition from (s,u) to $\{(s,u+d)_1\}$, if $(s,u) \xrightarrow{d} \{(s,u+d)_1\}$. $\square$

Definition 5.1.7 (Symbolic Transition)
Let ρ be a symbolic state space for the labelled Markov transition system $\mathcal{M} = (\Gamma, A, \rightarrow)$. Let $x \in \Gamma$ be a configuration. We say that there is a symbolic transition from x to $\mathbf{X}$, where $\mathrm{img}(\mathbf{X}) \subseteq \rho$, if there exists a time abstract Markov transition from x to X, where $\{\mathbf{X} = S\} = \{X \in S\}$ for all $S \in \rho$. $\square$

Definition 5.1.8 (Split)
Let S be a symbolic state and let ρ be a symbolic state space. The split of S with respect to ρ is a minimal partitioning α of S, such that all elements of each symbolic state $S' \in \alpha$ can take the same symbolic transitions with respect to the partitioning ρ.
We denote the split of S with respect to ρ by `split(`S`,`ρ`)`. $\square$

Definition 5.1.9 (Pre-states)
Let S be a symbolic state and let ρ be a symbolic state space. The pre-states of S in ρ are the symbolic states in ρ from which there exists a symbolic transition to S.
We denote the pre-states of S in ρ by $\texttt{pre}_\rho\texttt{(}S\texttt{)}$. $\square$

Definition 5.1.10 (Post-states)
Let S be a symbolic state and let ρ be a symbolic state space. The post-states of S in ρ are the symbolic states in ρ to which there exists a symbolic transition from S.
We denote the post-states of S in ρ by $\texttt{post}_\rho\texttt{(}S\texttt{)}$. $\square$

Inspired by our minimisation algorithm of [EH07], our current approach, the state space reduction algorithm, is given in Algorithm 5.1.1.

Algorithm 5.1.1: State Space Exploration and Reduction Algorithm I

Input: A partitioning ρ_0, and an initial configuration s_0

Output: A quotient set, ρ, of a time abstract Markov equivalence relation.

```
1  $\rho := \rho_0$
2  $\alpha := \{[s_0]_\rho\}$
3  $\sigma := \emptyset$
4  while $\alpha \neq \sigma$ do
5      choose $S$ in $\alpha \backslash \sigma$
6      let $\alpha' = \texttt{split}(S,\rho)$
7      if $\alpha' = \{S\}$ then
8          $\sigma := \sigma \cup \{S\}$
9          $\alpha := \alpha \cup \texttt{post}_\rho(\texttt{S})$
10     else
11         $\alpha := \alpha \backslash \{S\}$
12         if $\exists S' \in \alpha'$ such that $s_0 \in S'$ then
13             $\alpha := \alpha \cup \{S'\}$
14         $\sigma := \sigma \backslash \texttt{pre}_\rho(\texttt{S})$
15         $\rho := (\rho \backslash \{S\}) \cup \alpha'$
```

5.2 State Space Exploration and Reduction Algorithm I Computability

In addition to the common set operations of intersection, union and subtraction, we will use the following operations on sets of clock valuations:

Definition 5.2.1 (Past and Future)
Let S be a set of clock valuations. A clock valuation u is in the past of S if there exists a real number $\delta \geq 0$ such that $u + \delta \in S$. The set of all clock valuations in the past of S is written $S_\swarrow$.

Similarly, a clock valuation u is in the future of S if there exists a real number $\delta \geq 0$ such that $u - \delta \in S$. The set of all clock valuations in the future of S is written $S_\nearrow$. □

Definition 5.2.2 (Up Relaxation, Down Relaxation)
Let S be a set of clock valuations. A clock valuation u is in the up relaxation of S if for all $\delta > 0$ there exists a $\delta' \in [0;\delta]$ and a $u' \in S$ such that $u' + \delta' = u$. The set of all clock valuations in the up relaxation of S is written $S_\sqsupset$.

Similarly, a clock valuation u is in the down relaxation of S if for all $\delta > 0$ there exists a $\delta' \in [0;\delta]$ and $u' \in S$ such that $u' - \delta' = u$. The set of all clock valuations in the down relaxation of S is written $S_\sqsubset$. □

Definition 5.2.3 (Direct Delay Predecessor)
Let S and S' be sets of clock valuations. $u \in S$ is called a direct delay predecessor of S'

in S if there exists a $\delta \geq 0$ such that $u + \delta \in S'$ and for all $\delta' \in [0; \delta]$, $u + \delta' \in S \cup S'$. The set of all direct delay predecessors of S' in S is written $S \Uparrow S'$. □

Definition 5.2.4 (Clock reset)
Let u be a clock valuation and r a set of clocks. Then the result of the reset of r from u, written $[r \mapsto 0]u$ is the clock valuation where, for each clock c,

$$([r \mapsto 0]u)(c) = \begin{cases} 0 & \text{if } c \in r, \\ u(c) & \text{otherwise.} \end{cases}$$

For S a set of clock valuations, $[r \mapsto 0]S$ is defined as $\{[r \mapsto 0]u \mid u \in S\}$. □

Definition 5.2.5 (Inverse clock reset)
Let S be a set of clock valuations and r a set of clocks. The result of the inverse reset of r from S, written $[r \mapsto 0]^{-1}S$ is the set of clock valuations

$$\{u \mid ([r \mapsto 0]u) \in S\}.$$ □

Definition 5.2.6
In the context of networks of probabilistic timed automata we write $\langle s, U\rangle$ as notation for the symbolic state $\{(s, u) \mid u \in U\}$, where s is a common vector of discrete values for configurations contained in the symbolic state. □

Remark 5.2.7
The symbolic states of Definition 5.2.6 does not range over multiple discrete vectors, so the symbolic states used here are in a subclass of those defined in Definition 5.1.4. □

Corollary 5.2.8 (Direct Time Abstract Delay Predecessor)
Let $\langle s, U\rangle, \langle s, U'\rangle$ be symbolic states. For all $u \in U$, there is a time abstract Markov delay transition from (s, u) to some $\{(s, u')_1\}$, such that $[u; u'] \subseteq U \cup U'$, where $u' \in U'$, if and only if $u \in U \Uparrow U'$.

Proof
This follows directly from Definition 5.2.3 and Definition 5.1.6. ■

Definition 5.2.9 (Symbolic Edge)
There exists a symbolic edge between s and s' guarded by f and with resets r, written $s \xrightarrow{f,r} s'$, if for some action a, and some $p \in (0; 1]$, f is the largest non-empty federation that satisfies

$$(s, u) \xrightarrow{a} X, \quad \text{where} \quad \Pr\big(X = (s', [r \mapsto 0]u)\big) = p$$

for all $u \in f$. □

Lemma 5.2.10 (Direct Time Abstract Markov Switch Transition Predecessor)
Let $\langle s, U\rangle$ and $\langle s', U'\rangle$ be symbolic states. There exists a $p \in (0;1]$, such that for all $u \in U$, there is a time abstract Markov switch transition from (s,u) to some X, such that

$$\Pr\big(X \in \langle s', U'\rangle\big) = p > 0 \quad \textit{and} \quad (s,u) \xrightarrow{d} \{(s, u+d)_1\} \xrightarrow{a} X,$$

where $[u; u+d] \subseteq U$, if and only if

$$u \in U \Uparrow (U \cap f \cap [r \mapsto 0]^{-1} U') \tag{5.1}$$

for some federation f and clock set r, for which there exists a symbolic edge between s and s' with f as guard and with r as resets.

Proof
Ad (5.1) *implies existence:* Assume (5.1). By Definition 5.2.3 there exists a $d \in \mathbb{R}^+$ such that $(s,u) \xrightarrow{d} \{(s, u+d)_1\}$, where $u+d \in (U \cap f \cap [r \mapsto 0]^{-1}U')$ and $[u; u+d] \subseteq U$, for all u. By assumption $u+d \in f$ and by Definition 5.2.9 it follows that $(s, u+d) \xrightarrow{a} X$, where $\Pr(X = (s', [r \mapsto 0]u+d)) = p > 0$, for all u and some $p \in (0;1]$. Since by assumption $u+d \in [r \mapsto 0]^{-1}U'$ it follows from Definition 5.2.5 that $[r \mapsto 0]u+d \in U'$, and thus $\Pr(X \in \langle s', U'\rangle) = p > 0$ for all u. The result now follows from Definition 5.1.6.

Ad existence implies (5.1)*:* Assume that there is a time abstract Markov switch transition from (s,u) to X, such that

$$\Pr\big(X \in \langle s', U'\rangle\big) > 0 \quad \text{and} \quad (s,u) \xrightarrow{d} \{(s, u+d)_1\} \xrightarrow{a} X,$$

where $[u; u+d] \subseteq U$. From $(s, u+d) \xrightarrow{a} X$ and $\Pr(X \in \langle s', U'\rangle) > 0$ it follows by Definition 5.2.9 that there exists a federation f and a clock set r such that there is a symbolic edge between s and s' with guard f and resets r, such that $u+d \in f$ and $\Pr(X = (s', [r \mapsto 0]u+d)) > 0$, where $[r \mapsto 0]u+d \in U'$. By Definition 5.2.5 it follows that $u+d \in [r \mapsto 0]^{-1}U'$. Since $u+d \in U \cap f \cap [r \mapsto 0]^{-1}U'$ it follows by Definition 5.2.3 that

$$u \in U \Uparrow (U \cap f \cap [r \mapsto 0]^{-1}U').$$

■

Definition 5.2.11 (Partitioning According to Intersection)
Let $U_1, \ldots, U_n$ be sets of clock valuations. The partitioning according to intersection of $U_1, \ldots, U_n$, written $U_1 \sqcup \cdots \sqcup U_n$, is defined as the minimum partitioning of $U_1 \cup \cdots \cup U_n$ such that for each $U \in U_1 \sqcup \cdots \sqcup U_n$ and for each $i = 1, \ldots, n$ either $U \subseteq U_i$ or $U \cap U_i = \emptyset$.□

Lemma 5.2.12
Let $\langle s, U\rangle$ be a symbolic state, let ρ be a symbolic state space and let $s' \neq s$. Then

$$\texttt{split(}\langle s,U\rangle\texttt{,}\rho\texttt{)} = \bigsqcup_{\langle s',U'\rangle\in\rho} \texttt{split(}\langle s,U\rangle\texttt{,}\langle s',U'\rangle\texttt{)}, \tag{5.2}$$

$$\texttt{split(}\langle s,U\rangle\texttt{,}\langle s',U'\rangle\texttt{)} = \langle s,U\rangle \sqcup \bigsqcup_{s\xrightarrow{f,r}s'} \langle s, U \Uparrow (U \cap f \cap [r \mapsto 0]^{-1}U')\rangle, \quad \textit{and} \tag{5.3}$$

$$\texttt{split(}\langle s,U\rangle\texttt{,}\langle s,U'\rangle\texttt{)} = \langle s,U\rangle \sqcup \langle s, U \Uparrow U'\rangle \sqcup \bigsqcup_{s\xrightarrow{f,r}s} \langle s, U \Uparrow (U \cap f \cap [r \mapsto 0]^{-1}U')\rangle. \tag{5.4}$$

Proof
By Definition 5.2.11 the right hand side of (5.2) is the least partitioning such that all configurations in each symbolic state

i) can take the same time abstract Markov delay transitions by Corollary 5.2.8 and the right hand side of (5.3), and

ii) can take the same time abstract Markov switch transitions by Lemma 5.2.10, the right hand side of (5.3) and the right hand side of (5.4),

which by Definition 5.1.8 was to be shown. ■

Definition 5.2.13 (Time Convexity)
A clock valuation set F is called time convex if for all $u \in F$ it holds that for all δ, where $0 < \delta < \delta_{\text{sup}}$, we have $u + \delta \in F$, where δ_{sup} is the supremum of $\{\delta' \in \mathbb{R} \mid u + \delta' \in F\}$. □

Corollary 5.2.14 (Time Convexity of $\nearrow$ and $\swarrow$)
If F is a set of clock valuations, then $F_{\nearrow}$ and $F_{\swarrow}$ are time convex. □

Corollary 5.2.15 (Time Convexity is Closed under $\sqsupset$ and $\sqsubset$)
If F is a time convex set of clock valuations, then $F_{\sqsupset}$ and $F_{\sqsubset}$ are time convex. □

Remark 5.2.16
Even if F and F' are time convex sets of clock valuations, $F \cup F'$, $[r \mapsto 0]^{-1}F$ and $F \backslash F'$ are not necessarily time convex. □

Lemma 5.2.17 (Time Convexity is Closed under $\cap$)
Let F and F' be time convex clock valuation sets. Then the intersection $F \cap F'$ is also a time convex clock valuation set.

Proof
Let z be in $F \cap F'$, let δ_{sup} be the supremum of $\{\delta' \in \mathbb{R} \mid z + \delta' \in F\}$, let δ'_{sup} be the supremum of $\{\delta' \in \mathbb{R} \mid z + \delta' \in F'\}$ and let δ''_{sup} be the supremum of $\{\delta' \in \mathbb{R} \mid z + \delta' \in F \cap F'\}$. Clearly $\delta''_{\text{sup}} \leq \delta_{\text{sup}}$ and $\delta''_{\text{sup}} \leq \delta'_{\text{sup}}$, thus for all $\delta \in [0, \delta''_{\text{sup}}]$, we have that $z + \delta \in F \cap F'$. ■

Lemma 5.2.18 (Time Convexity of $\Uparrow$)
Let F and F' be clock valuation sets, with F time convex. Then $F \Uparrow F'$ is a time convex clock valuation set.

Proof
Let z be in $F \Uparrow F'$. Choose δ_{sup} as the supremum of

$$\{\delta' \in \mathbb{R}^+ \mid z + \delta' \in F, \text{ and there exists a } \delta \text{ such that } \delta' < \delta, \text{ and } z + \delta \in F'\}, \tag{5.5}$$

where existence of δ follows from Definition 5.2.3. By the time convexity of F, it follows that $z+\delta' \in F$ for all $\delta' \in [0; \delta_{\text{sup}})$. By (5.5) and by $z \in F \Uparrow F'$, we get that $z+\delta' \in F \Uparrow F'$ for all $\delta' \in [0; \delta_{\text{sup}})$. ■

Lemma 5.2.19 (Time Convexity of $\setminus \Uparrow$)
Let F and F' be clock valuation sets, with F time convex. Then $F \setminus (F \Uparrow F')$ is a time convex clock valuation set.

Proof
Let $z \in F \setminus (F \Uparrow F')$. By the time convexity of F we have $z + \delta \in F$ for all $\delta \in [0; \delta_{\text{sup}})$, where δ_{sup} is the supremum of $\{\delta' \in \mathbb{R}^+ \mid z + \delta' \in F\}$.

Assume erroneously that $z + \delta'' \in F \Uparrow F'$, for some $\delta'' \in [0; \delta_{\text{sup}})$. But then, by the time convexity of F, it follows that $z \in F \Uparrow F'$. But the definition of z then contradicts the assumption, and the assumption must be wrong. Time convexity for $F \setminus (F \Uparrow F')$ now follows by the time convexity of F. ■

Lemma 5.2.20 (Time Convexity is Closed Under $[r \mapsto 0]$)
Let F be a time convex clock valuation set, and r be a set of clocks. Then $[r \mapsto 0]F$ is a time convex clock valuation set.

Proof
By Definition 5.2.4 $\delta_{\text{sup}} = 0$ for all $z \in [r \mapsto 0]F$, where δ_{sup} is the supremum of

$$\{\delta' \in \mathbb{R}^+ \mid z + \delta' \in [r \mapsto 0]F\},$$

which implies time convexity for $[r \mapsto 0]F$. ■

Theorem 5.2.21 (The Time Convexity Theorem [EH07])
Let U and U' be sets of clock valuations, where U is time convex. Then

$$U \Uparrow U' = ((U_{\sqsupset} \cap U') \cup (U \cap U'_{\sqsubset}))_{\swarrow} \cap U.$$

Proof
It is enough to show that $u \in U \Uparrow U'$ if and only if

$$u \in ((U_{\sqsupset} \cap U') \cup (U \cap U'_{\sqsubset}))_{\swarrow} \cap U. \tag{5.6}$$

Ad $u \in U \Uparrow U'$ *implies* (5.6)*:* Let $u \in U \Uparrow U'$. By definition of u there exists a real number $\delta \geq 0$ such that $u + \delta \in U'$. Let $\delta_{\mathtt{sup}}$ be the supremum of $\{\delta' \in [0; \delta] \mid \delta' + u \in U\}$. Clearly $u + \delta_{\ sup} \in (U_{\sqsupset} \cap U') \cup (U \cap U'_{\sqsubset})$ and thus $u \in ((U_{\sqsupset} \cap U') \cup (U \cap U'_{\sqsubset}))_{\swarrow}$ and by definition $u \in U$, which was to be shown.

Ad (5.6) *implies* $u \in U \Uparrow U'$*:* Let (5.6) be given.
By definition $u \in U$.
By (5.6) there exists a $\delta'' \geq 0$ such that $u + \delta'' \in (U_{\sqsupset} \cap U') \cup (U \cap U'_{\sqsubset})$. If $u + \delta'' \in U'$ choose $\delta = \delta''$ otherwise by Definition 5.2.2 there exists an $\epsilon > 0$ such that $u + \delta'' + \epsilon \in U'$ and in that case chose $\delta = \delta'' + \epsilon$. By the time convexity of U (Definition 5.2.13) and by Definition 5.2.2 it holds that for all $\delta' \in [0; \delta]$ we get $u + \delta' \in U \cup U'$, which was what to be shown. ■

Lemma 5.2.22 (Time Convexity is Closed under $\sqcup$)
If U is a time convex set of clock valuations then all

$$U' \in \bigsqcup_{i=1}^{n} U \Uparrow U_i \tag{5.7}$$

are time convex sets of clock valuations, where U_i is a set of clock valuations for all i.

Proof
Proof by induction over n.

Ad (5.7) *for* $n = 1$*:* This follows immediately from Lemma 5.2.18.

Ad (5.7) *for* $n = k > 1$*:* Assume (5.7) for $n = k - 1$. Let

$$V \in \bigsqcup_{i=1}^{k-1} U \Uparrow U_i. \tag{5.8}$$

By assumption V is a time convex set of clock valuations.
It is enough to show that $V \backslash U \Uparrow U_k$ and $V \cap U \Uparrow U_k$ are time convex sets of clock valuations.
By deduction $V \backslash U \Uparrow U_k = V \backslash V \Uparrow U_k$, since $V \subseteq U$. It follows by Lemma 5.2.19 that $V \backslash U \Uparrow U_k$ is time convex.
And since $V \cap U \Uparrow U_k$ is time convex by Lemma 5.2.17 and Lemma 5.2.18, (5.7) for $n = k$ follows. ■

Corollary 5.2.23 (`split` is computable)
Given algorithms for computing $U_{\swarrow}$, $U_{\nearrow}$, $U_{\sqsupset}$, $U_{\sqsubset}$, $[r \mapsto 0]U$, $[r \mapsto 0]^{-1}U$, $U \backslash U'$ and $U \cap U$, where U and U' are sets of clock valuations `split(`$\langle s, V \rangle, \rho$`)` *is computable, where V is a time convex set of clock valuations and s is a location vector.*

Proof
This follows by Lemma 5.2.12 and Lemma 5.2.22. ■

Corollary 5.2.24 (State Space Reduction is Computable)
Given algorithms for computing $U_{\swarrow}$, $U_{\nearrow}$, $U_{\sqsupset}$, $U_{\sqsubset}$, $[r \mapsto 0]U$, $[r \mapsto 0]^{-1}U$, $U \backslash U'$ and $U \cap U$, where U and U' are sets of clock valuations, it follows by Corollary 5.2.23 and Algorithm 5.1.1, that state space reduction for probabilistic timed automata is computable. □

5.3 State Space Exploration and Reduction Algorithm I Correctness

Definition 5.3.1 (Forward Stability)
Let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system, and let ρ be a partitioning of Γ. If all $x, y \in S$ are time abstract Markov equivalent for all $S \in \rho$, we call ρ forward stable. □

Theorem 5.3.2 (State Space Reduction Algorithm Correctness)
Let ρ be the result of Algorithm 5.1.1. Then ρ is forward stable.

Proof
By Definition 5.3.1 it is enough to show that if (s, u) and (s, u') are in the same symbolic state $\langle s, U \rangle \in \rho$, then (s, u) and (s, u') are time abstract Markov equivalent. And by Definition 5.1.1 it is enough to show that (s, u) and (s, v) can reach the same symbolic states in ρ with equal probability, by a single probabilistic time abstract transitions, that is

i) by the same probabilistic time abstract delay transitions, and

ii) by the same probabilistic time abstract switch transitions

Ad i): Assume $(s, u) \xrightarrow{d} \{(s, u + d)_1\}$ where $d \in \mathbb{R}^+$, and $\Pr(\{(s, u + d)_1 \in \langle s, U' \rangle\}) = 1$ for some symbolic state $\langle s, U' \rangle$ and zero probability for all others (this assumption is sound by Definition 2.1.2), then there exists $d_1, \ldots, d_n \in \mathbb{R}^+$ such that

$$\sum_{i=1}^{n} d_i = d,$$

$$[u; u + d_1] \subseteq U \cup U_1,$$

$$\left[u + \sum_{j=1}^{n-1} d_j \; ; \; u + d \right] \subseteq U_{n-1} \cup U', \quad \text{and}$$

$$\left[u + \sum_{j=1}^{i} d_j \; ; \; u + \sum_{j=1}^{i+1} d_j \right] \subseteq U_i \cup U_{i+1} \quad \text{for} \quad i = 1, \ldots, n-2,$$

for some symbolic states $\langle s, U_1 \rangle, \ldots, \langle s, U_{n-1} \rangle \in \rho$. By construction there exists

$$d'_1, \ldots, d'_n \in \mathbb{R}^+,$$

such that

$$[v; v + d'_1] \subseteq U \cup U_1,$$

$$\left[v + \sum_{j=1}^{n-1} d'_j \; ; \; v + \sum_{j=1}^{n} d'_j \right] \subseteq U_{n-1} \cup U' \quad \text{and}$$

$$\left[v + \sum_{j=1}^{i} d'_j \; ; \; v + \sum_{j=1}^{i+1} d'_j \right] \subseteq U_i \cup U_{i+1} \quad \text{for} \quad i = 1, \ldots, n-2.$$

Now let

$$d' = \sum_{i=1}^{n} d'_i.$$

Since $(s, v) \xrightarrow{d'_1} \cdots \xrightarrow{d'_n} \{(s, v + d')_1\}$ we have that $(s, v) \xrightarrow{d'} \{(s, v + d')_1\}$, where $\Pr(\{(s, v + d)_1\} \in \langle s, U' \rangle) = 1$ and zero for all others, by Definition 2.1.2.

Ad ii): Assume $(s, u) \xrightarrow{d} \{(s, u + d)_1\} \xrightarrow{a} X$ where $d \in \mathbb{R}^+$,

$$\Pr\big(\{(s, u + d)_1\} \in \langle s, U' \rangle\big) = 1,$$

and $\Pr(X \in \langle s', U'' \rangle) = p$. By i) there exists a $d' \in \mathbb{R}^+$ such that $(s, v) \xrightarrow{d'} \{(s, v + d')_1\}$, where $\Pr(\{(s, v + d')_1\} \in \langle s, U' \rangle) = 1$. By Lemma 5.2.10 it follows that there exists a random variable Y satisfying $\Pr(Y \in \langle s', U'' \rangle) = p$ such that

$$(s, v + d') \xrightarrow{d''} \{(s, v + d' + d'')_1\} \xrightarrow{a} Y.$$

■

5.4 State Space Exploration and Reduction Algorithm I Minimality

Definition 5.4.1 (Minimal Equivalence Quotient Modulo a Partitioning)
Let $\equiv \subseteq \Gamma \times \Gamma$ be an equivalence relation, and let ρ_0 be a partitioning of Γ, where Γ is a set of configurations. If

$$\rho = (\Gamma / \equiv) \sqcup \rho_0,$$

where $\Gamma / \equiv$ is the quotient set of $\equiv$, we call ρ a minimal equivalence quotient of $\equiv$ modulo ρ_0. □

Corollary 5.4.2 (Minimality)
Let $\mathcal{A} = \{A_i\}_{1 \leq i \leq n}$ be a network of probabilistic timed automata, with state space $L^n \times \mathcal{U}(C)$ and invariant map $I \colon L^n \to \mathcal{Z}$, where C is the set of clocks in $\mathcal{A}$.

Let

$$\rho_0 = \left\{ \langle s, U \rangle \middle| s \in L^n, \, U \in (I(s) \sqcup \mathcal{U}(C)) \right\}$$

be an initial state space partitioning.

Let ρ be the output of Algorithm 5.1.1 evaluated on ρ_0. Then ρ is the minimal equivalence quotient of time abstract Markov equivalence modulo ρ_0.

Proof

By the definition of ρ_0 we can use `split` as defined in Lemma 5.2.12. The result follows from definitions 5.1.8, 5.1.6, 5.1.7 and 5.1.2. ■

5.5 State Space Exploration and Reduction Algorithm II

Algorithm 5.5.1: State Space Exploration and Reduction Algorithm II

```
STATES := $\emptyset$
STABLE := $\emptyset$
STATES.prepare(initial.discretes)
WAITING := $\{S \in$ STATES $\mid S$.is_initial$\}$
while not WAITING.is_empty() do
    source := WAITING.pop()
    if not split_on_transitions(source) then
        if not split_on_delays(source) then
            STABLE.add(source)
```

To make the *exploration* part of the algorithm explicit, we choose to represent the state space being explored with STATES, WAITING and STABLE. STATES is the set of symbolic states for which we need in-memory representations. WAITING and STABLE are both subsets of STATES; STABLE contains symbolic states that are stable with respect to action- and delay-transitions to symbolic states in STATES; all configurations reachable in one such transition from a symbolic state in STABLE are represented by members of STATES. Furthermore, states known to be reachable from the initial configuration will be in either WAITING or STABLE.

We only reduce with respect to time — configurations with different location- or variable vectors cannot be part of the same symbolic state. We will use the term *discretes* for the combined location- and variable-vectors of symbolic states or configurations.

Initialisation; lines 1–4: We first initialise the in-memory state space and stable subset to empty sets.

The state space is then extended: STATES.`prepare` will ensure that all configurations with the given discretes are represented by some symbolic state in memory. Whenever this procedure is called, we are sure that either all configurations with these discretes are already represented, or none are. In the case where no such configurations are represented, one or two symbolic states must be added: The first representing the configurations fulfilling the appropriate invariants, the second representing the configurations *not* fulfilling the invariants. As either set may be empty, we may need only one symbolic state.

WAITING will then be assigned the set of symbolic states representing the initial configuration — in practice a singleton set.

Main loop; lines 5–9: WAITING will contain symbolic states that are accessible but not yet ensured to be stable with respect to transitions to other symbolic states; including those that are in memory and those that are not. When WAITING is empty, all accessible states are stable and we are done.

Otherwise, we take an element from WAITING, and try to split it; first according to action-transitions, then according to delay-transitions. These procedures may add symbolic states to WAITING, remove symbolic states from STABLE, extend STATES with STATES.`prepare` and — most important — they may replace their input symbolic state with some partitioning of it.

If the selected symbolic state is stable according to both action- and delay-transitions; i.e. neither `split_on_transitions` nor `split_on_delays` find reasons to split it, it is marked as stable.

Split on Transitions

Procedure `split_on_transitions(source)`

```
foreach transition in source.transitions do
    if source $\Uparrow$ (transition.guard $\cap$ source) $=$ source then
        target := transition.apply(source)
        STATES.prepare(target.discretes)
        T := $\{t \in$ STATES $\mid t \cap$ target $\neq \emptyset\}$
        if T.size() $> 1$ then
            foreach $t$ in T do
                $s$ := source.shrink(transition,$t$)
                if $s \neq$ source then
                    simple_split(source,$s$)
                    return true
        else
            target := T.pop()
            if target $\neq$ source then
                add_path(source,target)
    else if source $\Uparrow$ (transition.guard $\cap$ source) $\neq \emptyset$ then
        simple_split(source,source $\Uparrow$ (transition.guard $\cap$ source))
        return true
return false
```

Main loop; lines 1 and 19: We consider — and split according to — each transition from the input state source. Any actual split will prematurely return from the procedure; thus, if we reach line 19, we know that the input state has not been split and return **false**.

Check guards; lines 2 and 16: On line 2 we check which parts of the input symbolic state can reach configurations satisfying the guard of the transition through delays.

If all of the input state can delay to the guard, we go on to consider splits according to which symbolic states may be reached; if only part of it can delay to the guard, we need to split the symbolic state into the part that can and the part that cannot. If no part of the symbolic state can delay to the guard, then no action is necessary on behalf of this transition.

Splitting on transition targets; lines 3–15: When we can take the transition, we compute the result of actually doing so. The result may be outside the in-memory part of the state space; i.e. we may need to extend STATES by calling STATES.`prepare`.

After ensuring that relevant symbolic states are in STATES, we find those intersecting with the computed target. If there is only one, then source is stable with respect to this transition. In that case, source is saved as a predecessor of the target symbolic state, and the target symbolic state will be added to WAITING if it is not already in WAITING or STABLE.

Otherwise, source is unstable with respect to this transition to at least one of the target symbolic states; see Proposition B.1.1. We split source according to the first of these and return.

Splitting on guards; lines 17–18: We do a `simple_split` of the source with respect to the part that can delay its way to the guard. We then return **true** from the procedure, signifying that a split has taken place.

Split on Delay

Procedure `split_on_delays(source)`

```
1 foreach target in STATES.with_discretes(source) do
2     if target $\neq$ source then
3         if source $\Uparrow$ target = source then
4             add_path(source,target)
5             return false
6         else if source $\Uparrow$ target $\neq \emptyset$ then
7             simple_split(source,source $\Uparrow$ target)
8             return true
9 return false
```

Main loop; lines 1 and 9: We consider all symbolic states with the same discretes to be potential delay-successors. As in `split_on_transitions`, we return **false** if no split has taken place.

Examining the possible successor; lines 2, 3 and 6: When we find the input symbolic state, no action should be taken. Otherwise, we check what part of the input symbolic state may reach the target symbolic state; all of source, some of source or no part of source.

If all of source is a direct delay predecessor of the target, then, under the assumption that symbolic states are disjoint, source can have no other direct delay successors. We save source as a predecessor of the target symbolic state and add the target symbolic state to WAITING if it is not already in WAITING or STABLE.

If only some of source is a direct delay predecessor of the target, we do a `simple_split` of the source with respect to this and return.

Reconsider Predecessors

Predecessors of the input state that were considered stable are moved to WAITING to be checked again.

Procedure `reconsider_predecessors(source)`

```
1 foreach predecessor in source.predecessors do
2     if STABLE.contains(predecessor) then
3         STABLE.remove(predecessor)
4         WAITING.add(predecessor)
```

New State

When a new symbolic state is added to STATES, we will want to check whether it contains the initial configuration — if it does, that symbolic state should be added to WAITING.[1]

Procedure `new_state(`s`)`

```
1 STATES.add($s$)
2 if $s$.is_initial then
3     WAITING.add($s$)
```

Simple Split

The input state source is split according to intersecting with target.[2]Before removing source, we call `reconsider_predecessors`, as predecessors of source are not necessarily

[1]As an optimisation, the check can often be avoided. For example, it is unnecessary if the source is not initial before splitting it in `simple_split`.

[2]We assume that source and other are time convex, and that source $\cap$ other $\neq \emptyset$ and source$\setminus$other $\neq \emptyset$.

stable with respect to the new partitioning after source has been split. We then remove source from STATES and replace it with the part intersecting with other and the part not intersecting with other.

Procedure simple_split(source,other)

```
reconsider_predecessors(source)
STATES.remove(source)
new_state(source $\cap$ other)
new_state(source$\setminus$other)
```

Add Path

The predecessor set of target is updated, and target is added to WAITING unless it is already in WAITING or STABLE.

Procedure add_path(source,target)

```
target.predecessors.add(source)
if not STABLE.contains(target) then
    WAITING.add(target)
```

5.6 State Space Exploration and Reduction Algorithm II Termination

In this section we argue that Algorithm 5.5.1 terminates. We do this by establishing that the size of STATES is non descending, in Lemma 5.6.4, and that the size of STATES is always changed within a bounded number of iterations, N (the size of the region graph), in Lemma 5.6.3 and Theorem 5.6.5 — that is, the size of STATES is increased within a bounded number of iterations. Since the size of STATES has an upper bound, also N, we can establish that the algorithm will run in $\mathcal{O}(N^2)$ iterations, which is finite.[3]

Definition 5.6.1
Let A be a variable in an algorithm with a main loop. The value of A before the first iteration of the main loop is written A_0 and the value of A after the n'th iteration is written A_n. □

Lemma 5.6.2
Invariantly WAITING *and* STABLE *are disjoint subsets of* STATES.

Proof
Enough to show that

i) WAITING and STABLE are invariantly subsets of STATES and

[3]This should not be interpreted as the algorithm runs in squared time of the input, since N is exponential in the number of clocks and in the number of probabilistic timed automata in the network of probabilistic timed automata.

ii) $\mathsf{WAITING}$ and STABLE are invariantly disjoint.

We do this by induction over the number of iterations n.

Ad i) and ii) for $n = 0$: Initially STABLE is empty and $\mathsf{WAITING} \subseteq \mathsf{STATES}$. i) and ii) are therefore trivially true.

Ad i) for $n > 0$: Assume ii) for the $n-1$'th iteration.

Enough to show i) for `new_state`, `simple_split` and `add_path`, in the n'th iteration, since these are the only ones altering the contents of $\mathsf{WAITING} \cup \mathsf{STABLE}$ or STATES.

In `new_state`, s is added to STATES before it is added to $\mathsf{WAITING}$.

In `simple_split` it follows by induction that source is not in $\mathsf{STABLE} = \mathsf{STABLE}_{n-1}$ since it is in $\mathsf{WAITING} = \mathsf{WAITING}_{n-1}$ in Algorithm 5.5.1 line 6, where it is removed from $\mathsf{WAITING}$; source is neither in STABLE nor $\mathsf{WAITING}$ when it is deleted from STATES in `simple_split`.

In `add_path`, we know that target is taken from STATES by `split_on_delays` or `split_on_transitions` respectively, therefore target can safely be added to $\mathsf{WAITING}$.

Ad ii) for $n > 0$: By induction, assume i) for the n'th iteration.

Enough to show that $\mathsf{WAITING}$ and STABLE being disjoint is an invariant of `add_path`, `reconsider_predecessors` and `new_state`, since these are the only procedures altering the contents of $\mathsf{WAITING}$ and STABLE.

In `add_path` nothing already in STABLE will be added to $\mathsf{WAITING}$.

In `reconsider_predecessors` anything added to $\mathsf{WAITING}$ is first removed from STABLE.

In `new_state`, it follows by `simple_split` that only symbolic states not already in STATES are added to STATES, and then added to $\mathsf{WAITING}$. It follows by induction that no such symbolic state can exist in STABLE at this point as $\mathsf{STABLE} \subseteq \mathsf{STATES}$. ■

Whenever a symbolic state is added to $\mathsf{WAITING}$, it holds that that symbolic state has not previously been used as source with the current partitioning in STATES:

Lemma 5.6.3
Let S be a symbolic state. For any $0 \leq n \leq k \leq m$, if

$$S \in \mathsf{WAITING}_n, \quad S \in \mathsf{WAITING}_m \quad \textit{and} \quad \mathsf{STATES}_n = \cdots = \mathsf{STATES}_m \tag{5.9}$$

then $S \in \mathsf{WAITING}_k$.

Proof
Assume (5.9), where $0 \leq n \leq m$. Assume erroneously that there exists a k, where $n \leq k \leq m$, such that $S \notin \mathsf{WAITING}_k$. By assumption, (5.9) and Algorithm 5.5.1 line 6–9 it follows that $S \in \mathsf{STABLE}_k$. By Lemma 5.6.2 it follows from $S \in \mathsf{WAITING}_m$ that $S \notin \mathsf{STABLE}_m$. This implies that `simple_split` has been called somewhere between the k'th and the m'th iteration of Algorithm 5.5.1. But `simple_split` line 2 falsifies (5.9), which is a contradiction. ■

Lemma 5.6.4
$|\mathsf{STATES}_{n+1}| \geq |\mathsf{STATES}_n|$ *for all* $n \geq 0$.

Proof
By `split_on_delays` and `split_on_transitions` we know that `simple_split` is always called with a symbolic state and a federation that intersects, without the symbolic state being fully contained in the federation. By `simple_split` it follows that if a symbolic state is removed from STATES two new are added. ∎

Theorem 5.6.5 (Termination)
Given a network of probabilistic timed automata, Algorithm 5.5.1 will terminate.

Proof
STATES is refined at least every |STATES|'th iteration, since otherwise all symbolic states in WAITING would have been moved to STABLE, by Lemma 5.6.3, and we would have terminated. By image finiteness a region graph R exists and thus |STATES| has an upper bound $N = |R|$.

When STATES_n is refined into STATES_{n+1} we get that

$$|\mathsf{STATES}_{n+1}| = |\mathsf{STATES}_n| + 1,$$

and by Lemma 5.6.4 |STATES| is non descending. It follows that the number of refinements also has the upper bound N. Therefore Algorithm 5.5.1 terminates within N^2 iterations. ∎

6 Probabilistic Reachability for Probabilistic Timed Automata

In this chapter we reduce probabilistic reachability analysis to linear programming problem using the forward stable state space of Chapter 5.

6.1 Reducing Probabilistic Reachability to Linear Programming

Definition 6.1.1 (Probabilistic Reachability)
INPUT: A tuple $(\Phi, \{A_i\}_{1\leq i\leq n})$, where Φ is $P_{\max}(a)$ or $P_{\min}(a)$, $a \in A \times \Gamma$ is an atomic proposition, and $\{A_i\}_{1\leq i\leq n}$ is a network of probabilistic timed automata, where $\mathcal{M} = (\Gamma, A, \rightarrow)$ is the labelled Markov transition system of $\{A_i\}_{1\leq i\leq n}$.

OUTPUT: A value $v \in \mathbb{R}$ defined by

$$v = \begin{cases} \max_{f\in\mathcal{F}(\mathcal{M})} \Pr(\mathcal{M}, x_0, \models f_n : a) & \text{if } \Phi = P_{\max}(a) \\ \min_{f\in\mathcal{F}(\mathcal{M})} \Pr(\mathcal{M}, x_0, \models f_n : a) & \text{if } \Phi = P_{\min}(a) \end{cases},$$

where x_0 is the initial configuration of $\mathcal{M}$. □

Definition 6.1.2 (Linear Programming)
INPUT: A tuple (V, K, B) where V is a vector of free variables $(v_1, v_2, \ldots, v_n)$, K is a set of linear constraints

$$\begin{aligned} \sum_{j=1}^{n} a_{1,j} v_j &\leq k_1 \\ &\vdots \\ \sum_{j=1}^{n} a_{m,j} v_j &\leq k_m \end{aligned}$$

where each $a_{i,j} \in \mathbb{R}$, and B is a linear combination of variables

$$b_1 v_1 + b_2 v_2 + \cdots b_n v_n,$$

where each $b_j \in \mathbb{R}$.

OUTPUT: A valuation vector $(c_1, c_2, \ldots, c_n) \in \mathbb{R}^n$ such that B is maximal under the linear constraints of K when v_i is bound to c_i for $1 \leq i \leq n$. □

Definition 6.1.3 (Symbolic Labelled Markov Transition System)
Let $\{A_i\}_{1\leq i\leq n}$ be a network of probabilistic timed automata, let $\mathcal{M} = (\Gamma, A, \rightarrow)$ be a labelled Markov transition system of $\{A_i\}_{1\leq i\leq n}$ and let ρ be a symbolic state space of Γ.

We define the labelled Markov transition system of ρ and $\mathcal{M}$ as $\mathbf{M} = (\rho, A, \rightarrow')$, where

$$\frac{x \in S \quad x \xrightarrow{a} X}{S \xrightarrow{a}{}' \mathbf{X}} \qquad \text{(SYMBOLIC)}$$

such that $\mathbf{X}$ is a random variable with $\mathrm{img}(\mathbf{X}) \subseteq \rho$ and

$$\{\mathbf{X} = S'\} \overset{\text{def}}{=} \bigcup_{x\in S'} \{X = x\},$$

for all $S' \in \rho$. □

Theorem 6.1.4
Probabilistic reachability is reducible to linear programming solving.

PROOF
Given a probabilistic reachability instance $D = (\Phi, \{A_i\}_{1\leq i\leq n})$ we build a linear programming instance $E = (V, K, B)$. It is then enough to show how to find the solution for D, given the solution for E.

We assume $\Phi = P_{\max}(a)$, where a is an atomic proposition. For $\Phi = P_{\min}(a)$, see Section 11.3.

Ad building E from D:

1. Take the probabilistic reachability instance $D = (\Phi, \{A_i\}_{1\leq i\leq n})$ as input.
2. Let $\mathcal{M} = (L^n \times \mathcal{U}(C), A, \rightarrow)$ be the labelled Markov transition system of $\{A_i\}_{1\leq i\leq n}$.
3. Let the atomic formula of Φ be
$$a = \bigcup_{1\leq j\leq m} \langle s_j, U_j\rangle.$$
4. Let
$$\rho_0 = \left\{ \langle s, U\rangle \middle| s \in L^n,\ U \in \left(I(s) \sqcup \mathcal{U}(C) \bigsqcup_{1\leq j\leq m} U_j \right) \right\}$$
be an initial state space partitioning.
5. Run Algorithm 5.1.1 on ρ_0 and get the forward stable partitioning ρ.
6. Let V be an initially empty variable vector.
7. Add each symbolic state $S \in \rho$ as a new free variable S' to V.
8. Let K be an initially empty set of linear constraints on V.

9. For each variable S'_i in V, add the linear constraint

$$-S'_i \leq 0$$

to K.

10. Let $\mathbf{M} = (\rho, A, \rightarrow')$ be the labelled Markov transition system of ρ and $\mathcal{M}$.
11. For each transition $S \rightarrow' \{S_{1,p_1}, S_{2,p_2}, \dots S_{q,p_q}\}$ of $\mathbf{M}$, add the linear constraint

$$-S' + p_1 S'_1 + p_2 S'_2 + \cdots + p_q S'_q \leq 0$$

to K, where S', S'_j is the variable of S and S_j, respectively, in V, for $1 \leq j \leq q$.

12. For each $S \subseteq a$ add $-S' \leq -1$ to K, where S' is the variable of S in V.
13. Let $B = -\sum_{S \in \rho} S'$, where S' is the variable of S in V.
14. Give the linear programming instance $E = (V, K, B)$ as output.

Ad solving D from a solution of E:

1. Take $(c_1, c_2, \dots, c_{|\rho|}) \in \mathbb{R}^{|\rho|}$ as input.
2. Give the c_j corresponding to the variable of the symbolic state in ρ containing the initial configuration of $\mathcal{M}$ as output.

Soundness follows from Theorem 3.3.9 since c_j is the least upper bound (supremum) for any configuration in S_j to reach a, for $1 \leq j \leq |\rho|$. ■

Example 6.1.5
In the following we go step-wise through the first part of the proof of Theorem 6.1.4, on a simple probabilistic reachability instance.

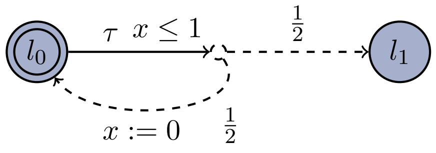


Figure 6.1: The network of probabilistic timed automata $\{A\}$

1. Take the probabilistic reachability instance $D = (\Phi, \{A\})$ as input, where $\Phi = P_{\max}(l_1)$ and $\{A\}$ is given in Figure 6.1.

3. Let $a = \langle l_1, x \geq 0 \rangle$.

4. Let $\rho_0 = \{\langle l_0, x \geq 0 \rangle, \langle l_1, x \geq 0 \rangle\}$.

5. By Algorithm 5.1.1 we get the forward stable partitioning

$$\rho = \{\langle l_0, x \geq 0 \wedge x \leq 1\rangle, \langle l_0, x > 1\rangle, \langle l_1, x \geq 0\rangle\}.$$

7. We represent these symbolic states with variables $V = \{v_0, v_1, v_2\}$, in the written order.

9. Let

$$K = \left\{ \begin{array}{c} -v_0 \leq 0, \\ -v_1 \leq 0, \\ -v_2 \leq 0 \end{array} \right\}.$$

10. Let $\mathbf{M}$ be the labeled Markov transition system illustrated in Figure 6.2.

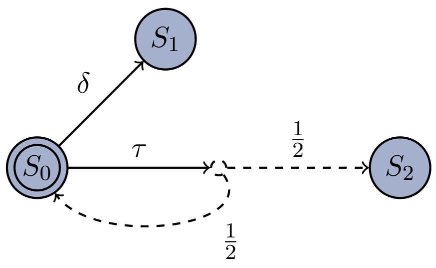


Figure 6.2: The labelled Markov transition system $\mathbf{M}$

11. We have two "probabilistic" transitions in the labeled Markov transition system, labeled τ and δ. The delay-transition, δ, is treated as a probabilistic transition where the probability of reaching v_1 is 1. From this, we get

$$K = K \cup \left\{ \begin{array}{r} -v_0 + v_1 \leq 0 \\ -v_0 + \dfrac{1}{2}v_0 + \dfrac{1}{2}v_2 \leq 0 \end{array} \right\}.$$

12. We have that $\langle l_1, x \geq 0\rangle$ fulfills the atomic proposition;

$$K = K \cup \{-v_2 \leq -1\}.$$

13. We construct the linear combination to be maximised:

$$B = -v_0 - v_1 - v_2.$$

14. We now output the linear programming problem (V, K, B), where

$$\begin{aligned}
V &= \{v_0, v_1, v_2\} \\
K &= \left\{ \begin{aligned} -v_0 &\leq 0, \\ -v_1 &\leq 0, \\ -v_2 &\leq 0, \\ -v_0 + v_1 &\leq 0, \\ -v_0 + \frac{1}{2}v_0 + \frac{1}{2}v_2 &\leq 0, \\ -v_2 &\leq -1 \end{aligned} \right\} \\
B &= -v_0 - v_1 - v_2.
\end{aligned}$$

□

Part III

Implementation

In this part we describe how State Space Exploration and Reduction Algorithm II is implemented, and how checking probabilistic reachability is implemented.

7 State Space Exploration and Reduction Algorithm II Implementation

In this chapter, we give an overview on how we have implemented State Space Exploration and Reduction Algorithm II. Finding predecessors is troublesome in UPPAAL, and we elaborate this further in Section 7.1. In Section 7.2 we present our solution for keeping track of predecessors. Finally, we give a UPPAAL pipeline for State Space Exploration and Reduction Algorithm II in Section 7.3.

7.1 Probabilistic Timed Automata in Uppaal

In UPPAAL PROB, we use an extended version of the existing language of UPPAAL for specifying timed automata and networks thereof. This has the obvious advantage to us that we do not need to implement the parts of UPPAAL PROB that are already in UPPAAL. But this also entails some limitations, especially regarding computations of predecessors, that we must somehow circumvent in UPPAAL PROB.

In UPPAAL, timed automata are instantiated from templates and composed into networks of timed automata in a system declaration. The configurations of timed automata in UPPAAL also contains shared and local variables, besides locations. Naturally, in the context of variables, it is possible to evaluate user defined procedures in the update part of edges.

Since the number of and range of variables is limited, the timed automata in UPPAAL are equivalent in computation power to those of Definition 4.1.4 and Definition 4.3.1, in the absence of probabilities. However, since a target state of a transition may be computed as the result of multiple procedural expressions, we get the aforementioned limitation of predecessor computations; before an assignment, a variable could have had any value in its entire domain.

Our extension of the timed automata in UPPAAL — probabilistic timed automata, and networks thereof — lies in branching edges; that is, edges with multiple targets (locations, updates and resets) and a probability procedure for each such target. The probability procedures may use the values of local and global variables and output an integer between 0 and 100, representing the probability of that target given in percentages (the sum should be 100). This is an analogue to the difference between formal timed automata and UPPAAL timed automata.

7.2 Singleton Data Structures

Since the algorithm is Section 5.5 is still based on a number of sets; WAITING, STABLE and STATES; we will require some kind of data structures for representing these efficiently.

As described in Section 7.1, we will need to keep track of reached predecessors of the symbolic states that we reach. Naturally we may simple store a symbolic state as predecessor of any symbolic states that it can reach in a single transition.

We use a single data structure to keep track of STATES and STABLE as well as predecessors. This data structure is, in a sense, a predecessor graph, hence we have called it the PredecessorGraph.

For use in the implementation, this data structure provides lookups by location and variable vectors, by intersection with symbolics states which may or may not be members of STATES and by *identity* — where we have assigned identifiers to each symbolic state in STATES to simplify parts of the implementation.

Separately, we have WAITING as a set of symbolic state identifiers inside the WaitingBuffer component. WaitingBuffer is described with the other "pipeline" components in Section 7.3.

7.3 Explore and Reduce Pipeline

In UPPAAL, algorithms are composed of components, in what in the UPPAAL architecture is called a "pipelines". There are a number of different interfaces for components, depending on the way they can be connected.

The most frequently used component type is filter: Filters are procedure wrapping objects that can be connected sequentially after each other, and instead of returning the result of the procedure, the result is forwarded to the next filter[1]. The advantage is that no bookkeeping is necessary for handling collections of data between each procedure call, since data may be passed on one piece at the time.

The overall pipeline of explore and reduce is shown in Figure 7.1, where the idea is that we keep splitting until the WaitingBuffer is empty.

The complete pipeline of SplitFilter, i.e. the main part of the algorithm, can be seen in Figure 7.2. In the following, the filters used for building the exploration and reduction algorithm are described.

SplitFilter

The SplitFilter is responsible for splitting a symbolic state with respect to all outgoing transitions, and all possible delay transitions. It is also responsible for marking symbolic states as stable, when they can not be split. In other words, SplitFilter is the main component of the pipeline constructed here — actions are taken directly by this component or components within it.

[1]Hence it is called a "pipeline" architecture

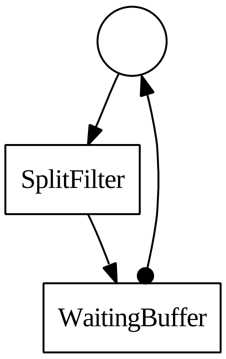


Figure 7.1: The pipeline of explore and reduce. The components that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows. Where data is pulled from a buffer is marked with a dot.

In Figure 7.3 the pipeline of the SplitFilter is shown. As input the SplitFilter takes a SymbolicStateHandle. It then tries to split the corresponding symbolic state with respect to first jump-transitions and then delay-transitions.[2] This is done with the internal SplitOnTransitionFilter and SplitOnDelayFilter. During these computations, new symbolic states may be found reachable, or, in case the input symbolic state is split, some symbolic states previously considered stable needs to be checked again. In both cases, these are sent as output of the SplitFilter. The output from SplitFilter is expected to be be sent to WaitingBuffer or some component with similar semantics.

If the symbolic state corresponding to the input SymbolicStateHandle is not split according to either jump-transitions or delay-transitions, it is marked as stable.

SplitOnDelayFilter

The SplitOnDelayFilter is responsible for splitting a symbolic state with respect to delay-wise successor symbolic states.

As input the SplitOnDelayFilter takes a SymbolicStateHandle. The symbolic state for this handle, as well as other with the same location- and variable-vectors, for it to be split according to, are looked up in the PredecessorGraph.

This component corresponds closely to the code for `split_on_delays` in Section 5.5. To work with the pipeline architecture, adding symbolic states to WAITING in `split_on_delays` corresponds to outputting SymbolicStateHandles from the SplitOnDelayFilter.

The SplitOnDelayFilter is intended for use inside the SplitFilter component.

[2]In the UPPAAL code base, these are commonly referred to as *transitions* and *delays*; this convention has been adopted in our implementation.

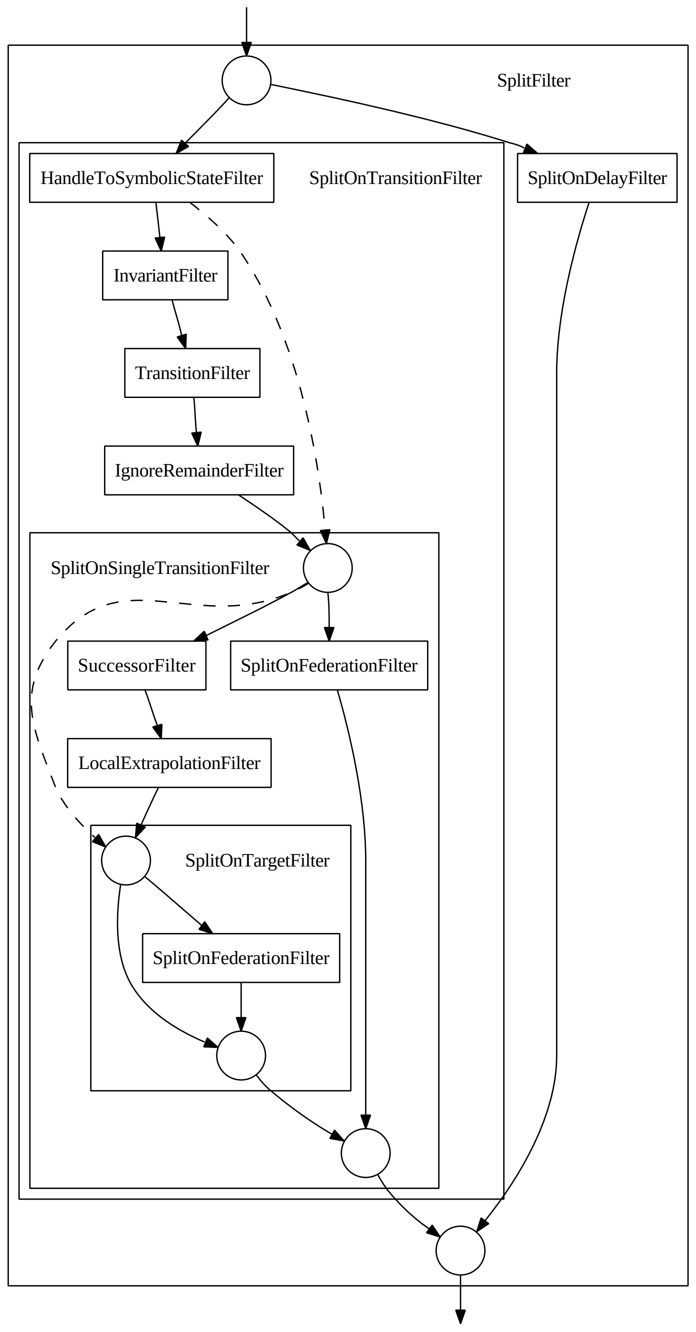


Figure 7.2: Pipeline for the contents of the main loop in Algorithm 5.5.1. The internal logic of composite components is drawn as circles. The filters that compose the pipeline are drawn as boxes. The flow of data through the pipeline is illustrated by arrows.

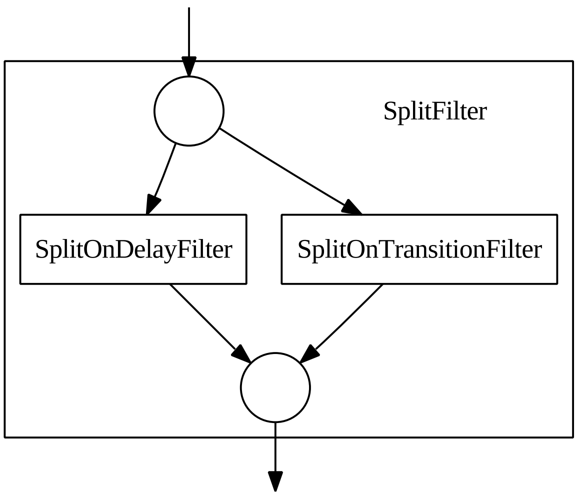


Figure 7.3: The pipeline of the SplitFilter. The internal logic of the pipeline is drawn as circles. The filters that compose the pipeline are drawn as boxes. The flow of data through the pipeline is illustrated by arrows.

SplitOnTransitionFilter

The SplitOnTransitionFilter is responsible for splitting a symbolic state with respect to all outgoing transitions. This corresponds to `split_on_transitions` in Section 5.5.

In Figure 7.4, the pipeline of the SplitOnTransitionFilter is shown. As input the SplitOnTransitionFilter takes a SymbolicStateHandle. This SymbolicStateHandle is converted to a SymbolicState by the HandleToSymbolicStateFilter and handed on to the InvariantFilter. The InvariantFilter only lets SymbolicStates through that satisfy their invariant. If the SymbolicState got through the InvariantFilter, it is handed on to the TransitionFilter. The TransitionFilter enumerates all satisfied outgoing discrete transitions as Successor structures and sends them on to the IgnoreRemainderFilter. The IgnoreRemainderFilter will forward these to the SplitOnSingleTransitionFilter until the SplitOnSingleTransitionFilter signals a “stop” for the current symbolic state, i.e. when the symbolic state under consideration has been split by SplitOnSingleTransitionFilter.

Note that since TransitionFilter does not support SymbolicStateHandles, it is necessary for the HandleToSymbolicStateFilter to forward the SymbolicStateHandle to the SplitOnSingleTransitionFilter. This is illustrated by the dashed arrow in Figure 7.4.

The SplitOnTransitionFilter is intended for use inside the SplitFilter component.

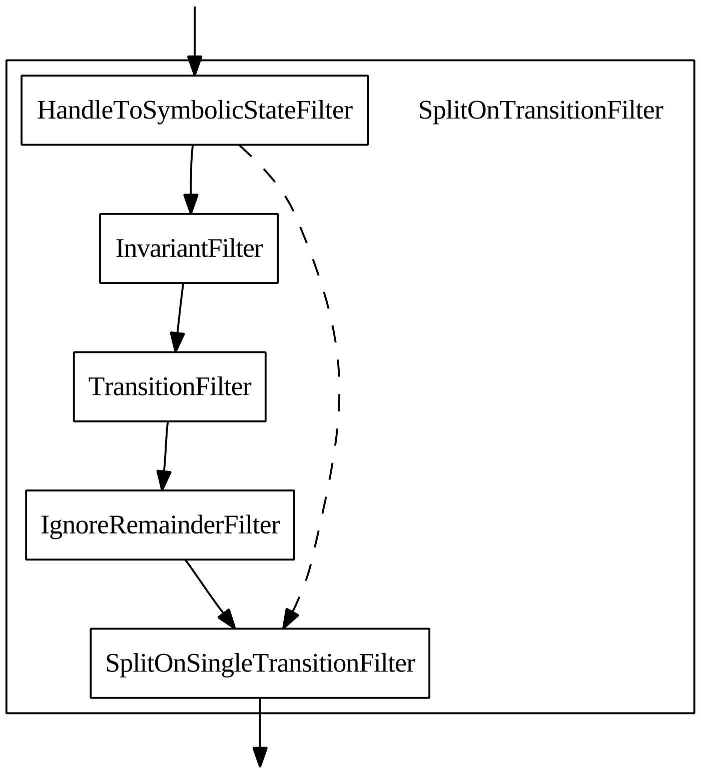


Figure 7.4: The pipeline of the SplitOnTransitionFilter. The filters that compose the pipeline are drawn as boxes. The flow of data through the pipeline is illustrated by arrows and the flow of meta data through the pipeline is illustrated by dashed arrows.

HandleToSymbolicStateFilter

This simply looks up a SymbolicState from a SymbolicStateHandle in our Predecessor-Graph data structure. The obtained SymbolicState is passed on as output.

InvariantFilter

This filter checks that all relevant invariants for the symbolic state are fulfilled. The symbolic state will be passed on only when they are.

TransitionFilter

This is a pre-existing component from the verifier module of UPPAAL. It generates a number of Successor structures representing each jump-transition whose guard is satisfied somewhere in the input SymbolicState.[3] These are followed by a special sentinel value indicating that all transitions from the input symbolic state has been generated.

IgnoreRemainderFilter

When some symbolic state is split by some later component, specifically SplitOnSingleTransitionFilter, further transitions from that symbolic state should be disregarded. There is no built-in way to prematurely end the generation of Successors from TransitionFilter. In the interest of abstraction and separation of concern, the IgnoreRemainderFilter makes it appear that there is: When the next filter returns *true* to signal that the symbolic state has been split, it will not receive subsequent Successor structures for that symbolic state.

SplitOnSingleTransitionFilter

The SplitOnSingleTransitionFilter is responsible for splitting a symbolic state with respect to a single outgoing transition.

In Figure 7.5, the pipeline of the SplitOnSingleTransitionFilter is shown. As input the SplitOnSingleTransitionFilter takes a Successor structure; a source SymbolicState and a transition.

- If all of the source SymbolicState can take the given transition, i.e. $s \Uparrow g = s$, where s is the federation of the source SymbolicState, and g is the guard of the given transition, the Successor is forwarded to the SuccessorFilter, which calculates the target SymbolicState of the transition taken from the source SymbolicState and sends it on to the LocalExtrapolationFilter and then the SplitOnTargetFilter.
- If only some of the source SymbolicState can take the given transition, i.e. $s \Uparrow g$ is non empty and $s \Uparrow g \neq s$, where s is the federation of the source SymbolicState, and g is the guard of the given transition, then the source SymbolicState and the

[3]The filter has some more advanced features regarding broadcasts and committed locations and edges, though those are not otherwise supported in our implementation.

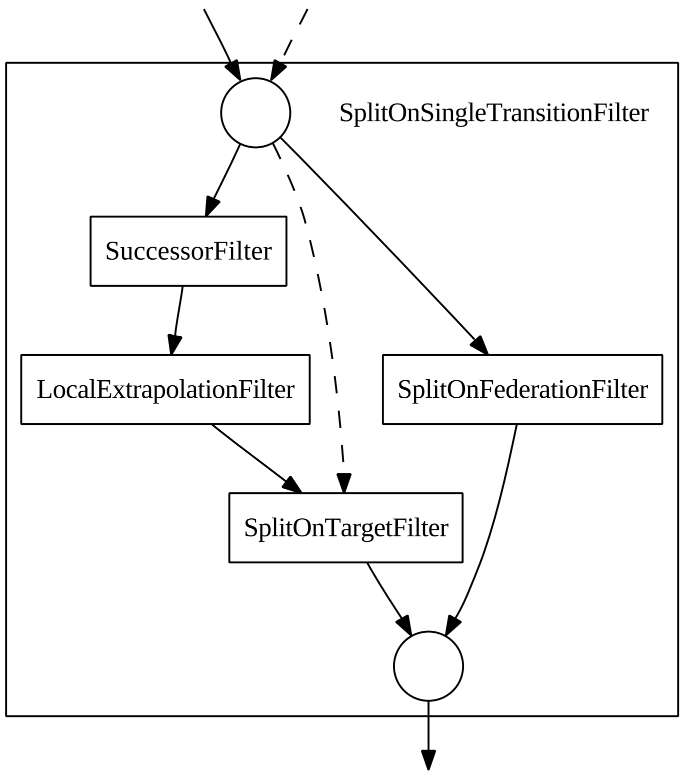


Figure 7.5: The pipeline of the SplitOnTransitionFilter. The filters that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows and the flow of meta data through the pipeline is illustrated by dashed arrows.

direct delay predecessor between the source Symbolic State and the guard of the transition, $s \Uparrow g$, is send on to the SplitOnFederationFilter.

Note that the SplitOnSingleTransitionFilter maintains a source SymbolicStateHandle, which was also mentioned in the description of SplitOnTransitionFilter: Since the SuccessorFilter does not support SymbolicStateHandles, and since its output does not even contain information on what the source was, the source SymbolicStateHandle of the SplitOnTargetFilter is updated whenever the source SymbolicStateHandle of SplitOnSingleTransition is. The flow of the source SymbolicStateHandle is illustrated in Figure 7.5 by the dashed arrows.

SuccessorFilter

This is a pre-existing component from the verifier module of UPPAAL. It takes a (source, transition) Successor data structure, computes the results of the transition and outputs this in a (target, transition) Successor data structure.

LocalExtrapolationFilter

This is a pre-existing component from the verifier module in UPPAAL. It optimises the difference bound matrices of the federation of a symbolic state such that they only contain important bound combinations. The class of important bound combinations are established by static analysis, before the filter is used the first time. This filter ensures that we will only consider a finite symbolic state space.

SplitOnTargetFilter

The SplitOnTargetFilter is responsible for splitting a SymbolicState with respect to the target SymbolicStates of a given transition.

A SplitOnTransitionFilter has a current source SymbolicState. As input the SplitOnTransitionFilter takes a Successor data structure containing a target SymbolicState and a transition. It obtains the SymbolicStates in our PredecessorGraph that overlap the target SymbolicState, and checks what parts may reach each specific symbolic state through the transition. In case some target symbolic state can only be reach from part of the source, SplitOnFederation is used to split the source symbolic state according to the first of these.

The output is SymbolicStateHandles for consideration in our WaitingBuffer; either a single SymbolicStateHandle reachable through a stable transition or a number of SymbolicStateHandles to be considered as the consequence of a split.

The SplitOnTargetFilter is intended for use inside the SplitOnSingleTransitionFilter component.

SplitOnFederationFilter

The SplitOnFederationFilter is responsible for splitting a SymbolicState into the part that intersects a given federation and the part that does not.

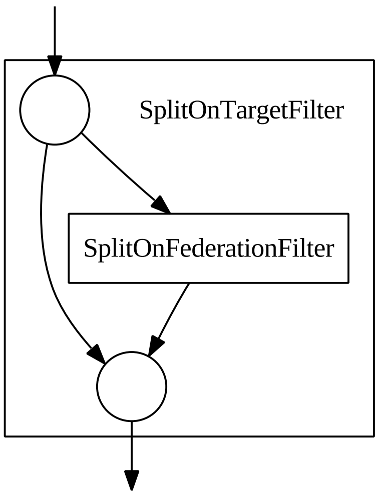


Figure 7.6: The pipeline of the SplitOnTransitionFilter. The filters that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows.

As input the SplitOnFederationFilter takes a SymbolicStateHandle and a federation. As output it gives two SymbolicStateHandles. No checks are performed to ensure that the resulting symbolic states are non-empty. For valid input, the federation should describe a non-empty strict subset of the clock valuations given in the symbolic state.

WaitingBuffer

The WaitingBuffer is a pipeline-component for maintaining the WAITING set: Its input is added to the set, and it provides methods for checking whether the set is empty and for obtaining and removing "some member" from the set.

8 Probabilistic Reachability in Probabilistic Timed Automata

In this chapter, we document how the probabilistic reachability solver of UPPAAL PROB is implemented. Probabilistic reachability was defined in Definition 6.1.1. The main idea is closely related to that of Theorem 6.1.4, with a few modifications. As in the proof of Theorem 6.1.4, we only consider $P_{\max}(a)$ formulae, where a is an atomic proposition.

First, the predecessor graph is generated so that it respects the atomic propositions. The details of this pipeline is found in Section 8.1.

When the predecessor graph is stable with respect to a, the probabilistic symbolic transitions are translated to linear constraints, as in Theorem 6.1.4. This is done in the probabilistic reachability pipeline described in Section 8.2, that also takes care of the rest of the reduction to linear programming.

8.1 Split on Atomic Propositions Pipeline

Since the predecessor graph does not initially contain a partitioning of the reachable state space, we can not do exactly as in Theorem 6.1.4, where we pre splitted the initial state space partitioning with respect to the atomic proposition a.

Instead, we do the splitting with respect to atomic propositions *after* the predecessor graph has been built. This destabalises the predecessors of the symbolic states that are split. The predecessors of the symbolic states that are split are yet again added to the WaitingBuffer. The overall pipeline of splitting with respect to atomic proposition is shown in Figure 8.1.

After splitting with respect to atomic propositions the main pipeline of Chapter 7 is rerun, to restabalise the predecessor graph.

8.2 Probabilistic Reachability Pipeline

As before, we use the pipeline-architecture of UPPAAL when implementing construction of linear programming problems. The InvariantFilter, TransitionFilter, SuccessorFilter and LocalExtrapolationFilter are reused; these have been described in Section 7.3.

The overall pipeline of probabilistic reachability is shown in Figure 8.2.

In the following, the StableToTransitionFilter, CopyFilter, SuccessorToProbabilistic-TransitionClusterFilter and StableToDelayTransitionFilter will be described.

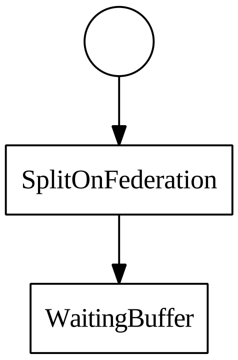


Figure 8.1: Split on atomic propositions pipeline. The filters that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows.

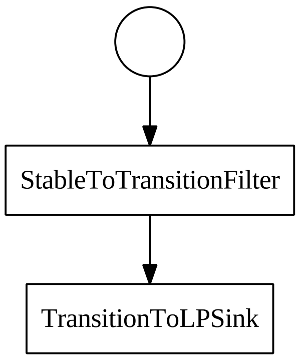


Figure 8.2: The pipeline of probabilistic reachability. The filters that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows.

StableToTransitionFilter

The StableToTransitionFilter will, given a handle to a stable symbolic state, generate a sequence of tuples representing probabilistic symbolic transitions from this symbolic state.

StableToTransitionFilter uses InvariantFilter, TransitionFilter, CopyFilter, Successor-Filter, LocalExtrapolationFilter and SuccessorToProbabilisticTransitionClusterFilter in sequence to generate action-transitions, and StableToDelayTransitionFilter to generate delay-transitions. This is illustrated in Figure 8.3.

CopyFilter

This component is part of UPPAAL. The CopyFilter exists mainly to support the differences in memory management conventions between TransitionFilter and SuccessorFilter.[1] Variants with different optimisation exist.

SuccessorToProbabilisticTransitionClusterFilter

This component will receive a sequence of (target, transition) Successor data structures, and from these generate a sequence of (source, transition, target, probability) tuples. Transitions that are the probabilistic consequences of the same choices are sent as a sequence, with a sentinel at the end of each such sequence.

The component is responsible for remembering the source symbolic state handle, obtaining a symbolic state handle for the target symbolic state, computing the probability for probabilistic transitions and ordering the transitions according to what probabilistic edges take part in them.

StableToDelayTransitionFilter

Given a stable symbolic state, StableToDelayTransitionFilter outputs a sequence of (source, transition, target, probability) tuples, with sentinels after each transition. The output format is compatible with that of SuccessorToProbabilisticTransitionClusterFilter.

The probability is always one, i.e. with the semantics from SuccessorToProbabilistic-TransitionClusterFilter, the probability of reaching a specific target through a delay is one when at all possible, and for stable symbolic states, at most one other symbolic state is accessible through delays.

TransitionToLPSink

The overall functionality of this component is to solve probabilistic reachability, by reducing to linear programming and using LP_SOLVE to solve the linear programming

[1] TransitionFilter and SuccessorFilter are often used together, though they disagree strongly on memory management conventions: The TransitionFilter will reuse the same data structure as "source" for each transition, while the SuccessorFilter will immediately overwrite the received data structure with the target computed from the source and transition.

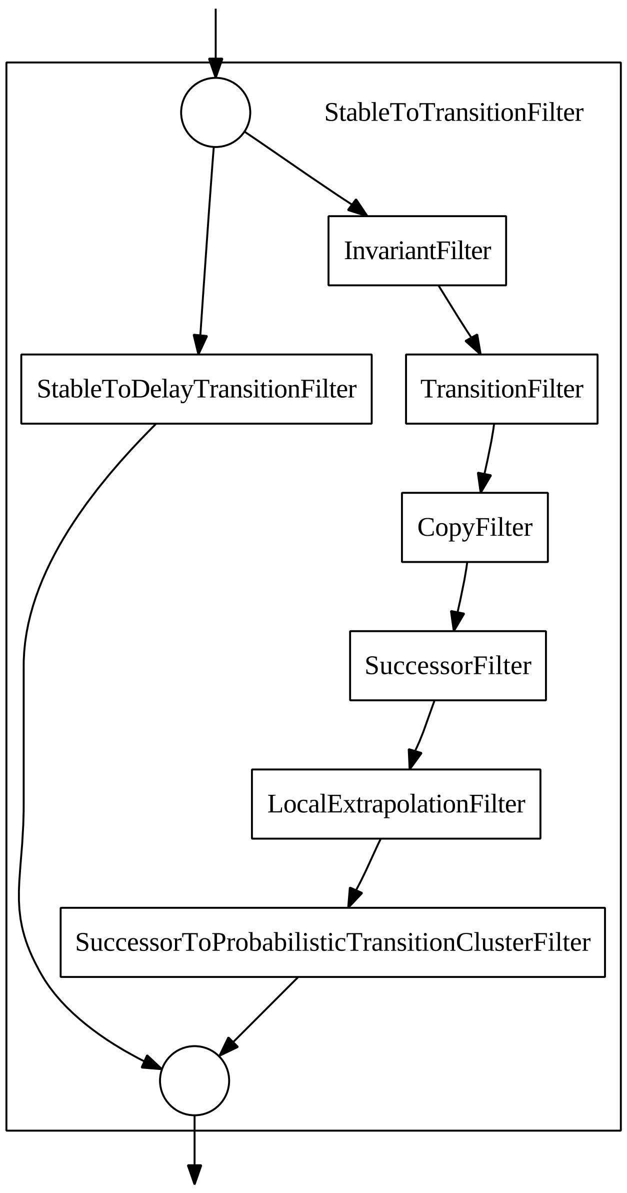


Figure 8.3: The pipeline of the StableToTransitionFilter. The internal logic of the pipeline is drawn as circles. The filters that compose the pipeline are drawn as boxes. The flow of data through the pipeline is illustrated by arrows.

instance. The reduction is done as described in the proof of Theorem 6.1.4.

This component takes a sequence of transition tuples separated by sentinels as input. Multiple transitions received successively are expected to be transitions of the same probabilistic transition. When a sentinel is received, a linear constraint on the form specified by step 11 in the proof of Theorem 6.1.4 is constructed from the current probabilistic transition.

Part IV

Uppaal Prob

In this part, we describe some basic features of Uppaal Prob: The two exporters and we give a short tour of the graphical user interface.

9 Exporting the Symbolic State Space

9.1 Exporting to the Graphviz DOT language

Graphviz is an open source graph visualisation software suite, capable of graphically organising nodes and edges in graphs described in the DOT language. For debugging purposes we support exporting the predecessor graph to a DOT file.

Recall the timed automata example from Chapter 1 in Figure 9.1. In Figure 9.2 the predecessor graph of Figure 9.1 is shown. Qualitative reachability analysis can easily be performed by humans given such a predecessor graph; if you can see it, it is reachable.

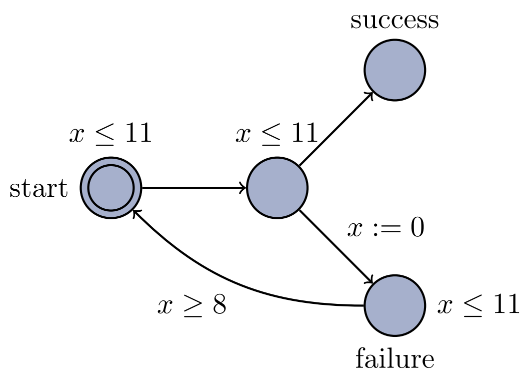


Figure 9.1: The timed automaton $\mathcal{A}$, again.

9.2 Encoding Networks of Probabilistic Timed Automata in Prism

Prism [HKNP06] is a probabilistic model checker for discrete time Markov chains, continuous time Markov chains, Markov decision processes, and variations thereof. In Prism the specification language for discrete stochastic processes is called PCTL[1].

Definition 9.2.1
Let $\mathcal{A}$ be a network of probabilistic timed automata with state space $\Gamma \times \mathcal{U}(C)$. An atomic PCTL proposition of $\mathcal{A}$ is a set $a \subseteq \Gamma$ of discrete vectors. $\square$

[1]probabilistic computation tree logic

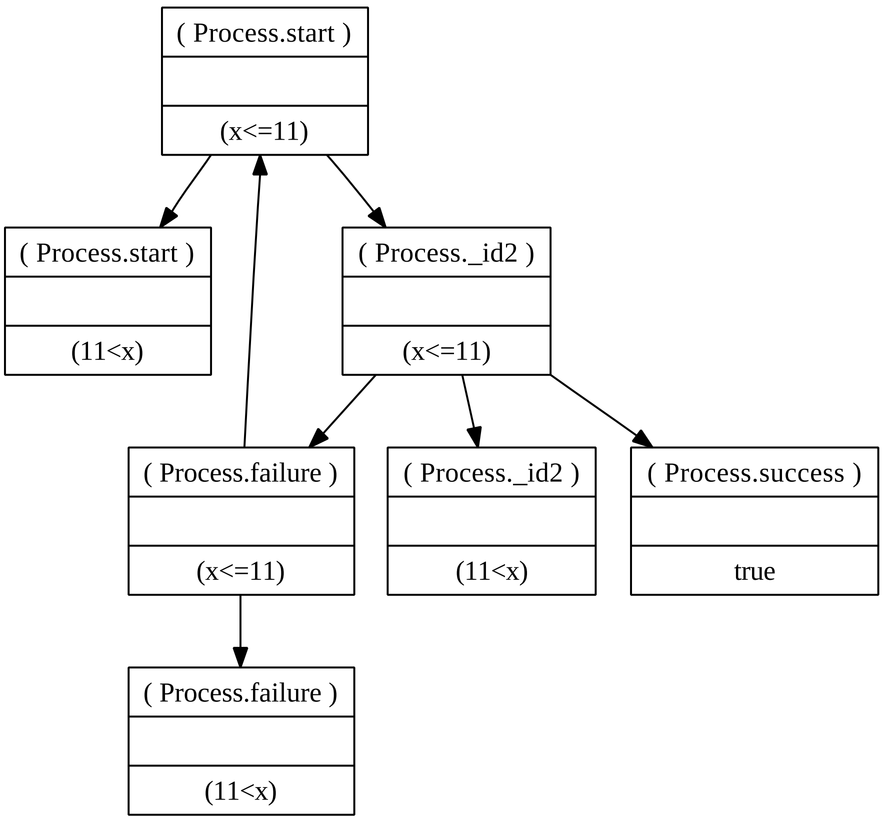


Figure 9.2: The predecessor graph of the timed automaton $\mathcal{A}$. The arrows indicates a predecessor relation (the arrow goes from source to target). The tables are symbolic states: The first entry in the tables is the location vector, the second is the variable vector (does not apply for this model), and finally the third entry in the table is a human readable description of the federation of clock valuations.

With the above definition of atomic propositions, clock valuations need not be represented in any human readable form in the PRISM model, and it is now enough to encode a network of probabilistic timed automata as a single PRISM module where:

- The current location is stored in an integer variable for each probabilistic timed automata.
- The current value of a variable is stored in an integer variable for each variable in the network of probabilistic timed automata.
- The current federation is stored in an integer variable using the serial number of the federation.

The pipeline for building such a prism module is illustrated in Figure 9.3. The components that we have not seen before are documented in the following.

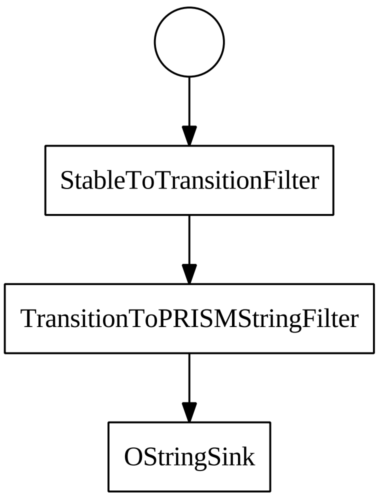


Figure 9.3: The PRISM exporter pipeline. The components that compose the pipeline are drawn as boxes. The internal logic of the pipeline is drawn as circles. The flow of data through the pipeline is illustrated by arrows.

TransitionToPRISMStringFilter

This component takes a sequence of transition tuples separated by sentinels as input. Multiple transitions received successively are expected to be transitions of the same probabilistic transition. When a sentinel is received, a PRISM transition is constructed for the currently stored probabilistic transition and passed on as a string to the sink.

OStringSink

The responsibility of this component is to write its input to a given stream.

10 Graphical User Interface

10.1 Editor

An edge with multiple targets could be drawn as a tree, with the root at the source location and each leaf at a target location. To simplify the file format and the GUI, we allow only to use one intermediate node, from which all branches start.

In Figure 10.1 an edge with multiple targets is shown. The connection from the source to the intermediate node is a regular arrow, while the one from the dashed intermediate node to the targets are dashed arrows.

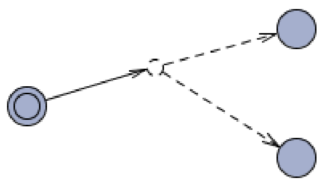


Figure 10.1: Edge with multiple targets.

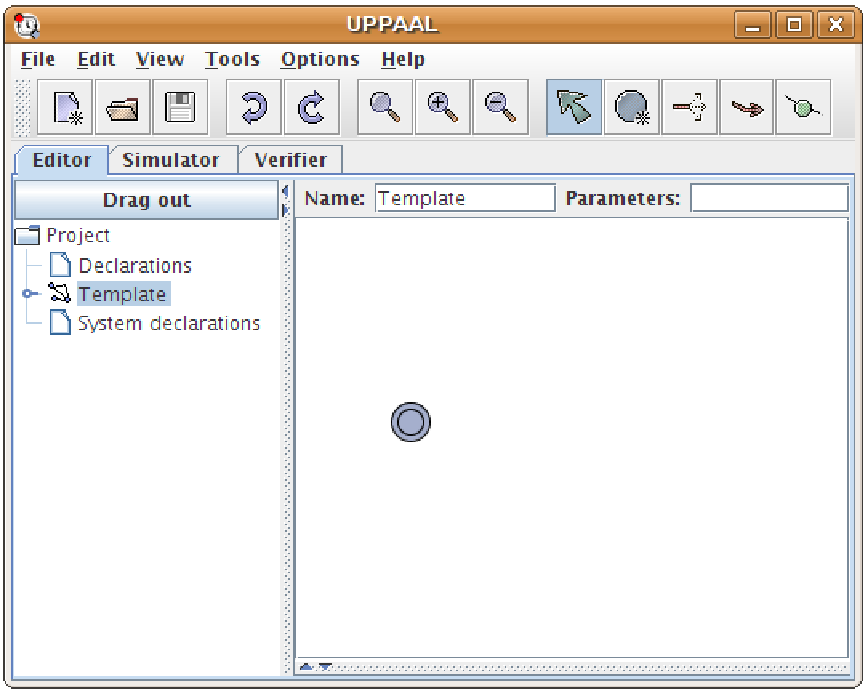
For each probabilistic branch, a probability must be assigned. The probabilities are currently expressed in percent; using an integer-based representation means that we can allow UPPAAL's general integer expressions to be used. The probability distribution for a transition can thus be computed from the values of integer variables in the configuration.

Guard and synchronisation can be specified only on the edge leading to the intermediate node; as it is only the target of the edge that should be probabilistic.

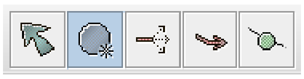
Updates, i.e. clock resets and changes to integer variables may, on the other hand, be specified on both parts of the probabilistic edge. Here, having some update on the edge to the intermediate node is simply shorthand for including it in all the branches: When evaluating updates, the updates from the edge leading to the intermediate node are performed first, followed by updates from the selected branch.

10.2 Making a Simple Probabilistic Model

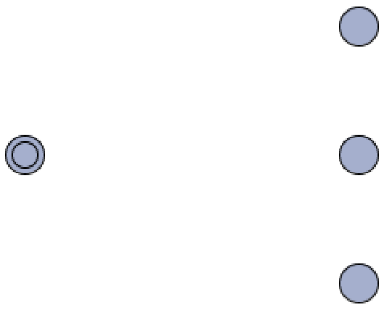
We give a step-by-step description of how some simple probabilistic model may be entered in UPPAAL PROB.



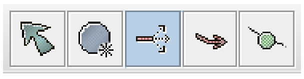
Uppaal Prob, like the normal Uppaal, by default starts with a model containing only an initial location in a single template.



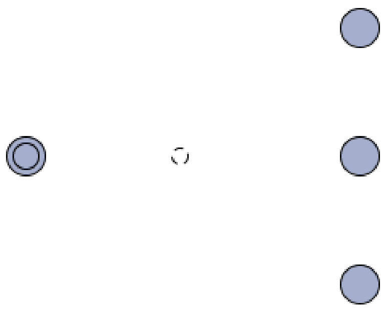
Select the location tool. Hereafter, each click on the modelling canvas adds a new location. Use this to add three locations.



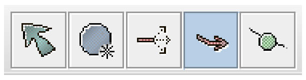
We want to make a probabilistic model. To do this, we need to add a *branching node*. Select the branching tool.



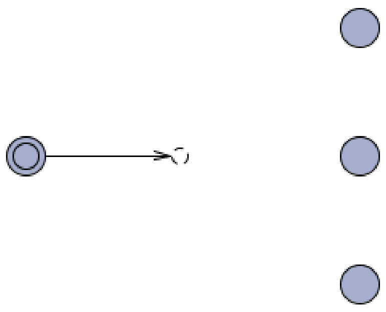
Again, clicking on the modelling canvas will add branching nodes to the model.



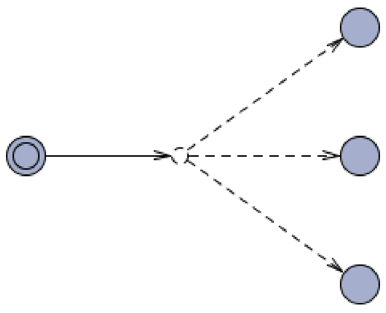
Now, we wish to connect the locations in the model by some transition. Select the edge tool.



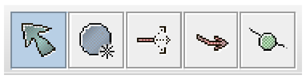
Clicking on the initial location, and then clicking on the branching node will add an edge part between them.



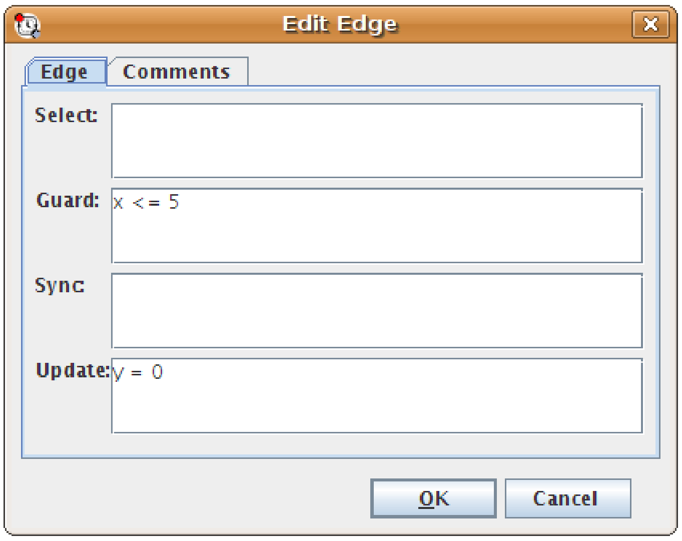
Repeat this for edge parts from the branching node to the other locations. Note that these edge parts are dashed, to signify that they are part of a probabilistic edge.



Finally, we wish to specify a guard, some updates and probabilities for the probabilistic edge. To do this, we need the selection tool.



Double-clicking on the edge part from the initial location to the intermediate node brings up an “edit edge” window. Here, we enter the guard $x \leq 5$ and the update $y := 0$.



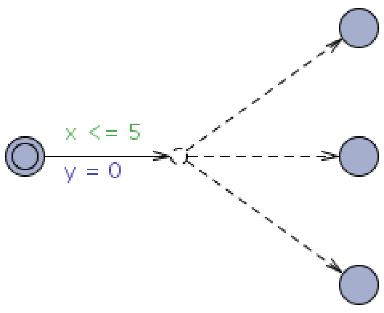
Select:

Guard: x <= 5

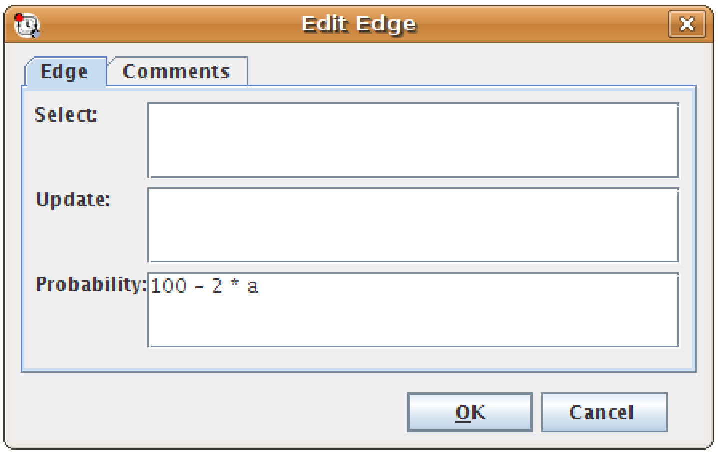
Sync:

Update: y = 0

The entered information is shown as text labels in different colours; these can be dragged around to make the model clearer, or more aesthetically pleasing.



The edge parts from the intermediate node can be similarly selected and edited. Here, we may not specify guards or synchronisation channels; instead, we can — and should — specify a probability. Probabilities are specified in percent; general UPPAAL integer expressions may be used.

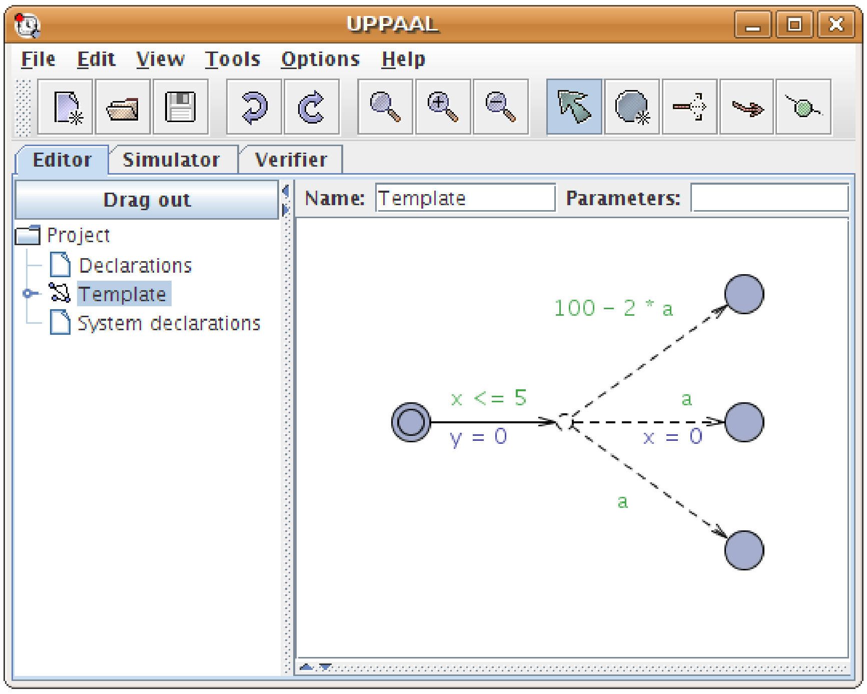


Select:

Update:

Probability: 100 - 2 * a

We now have a simple probabilistic model. With the guards, updates and probabilities entered here, we will need x to be a clock, y to be a clock or variable and a to be a variable or constant. For declaring these, we refer to [BDL04].



Name: Template Parameters:

Part V

Conclusion

In this part, we conclude the project with future work and a summary.

11 Future work

11.1 Prism as a Back-end

In Section 9.2 we explained how we encode networks of probabilistic timed automata as PRISM models in UPPAAL PROB.

In our current version of UPPAAL PROB, any query containing clock constraints would require the user to figure out which federation index corresponds to the given clock constraint if any: As future work we would really like to see PRISM as a back-end for UPPAAL PROB, so that we may make these formula translations automatically, and hide the indexing of federations from the user. This would make it possible to verify PTCTL formulae from UPPAAL PROB, using PCTL in PRISM.

11.2 Partial Exploration and Refinement

Running the exploration and refinement algorithm to completion should not always be necessary: When some specific formula is to be checked, what is reachable *after* fulfilling the formula may not be important. Furthermore, some simpler analysis may reveal e.g. that the target cannot be reached from certain location vectors; exploration and refinement of these are then unnecessary.

If only some upper or lower bounds on e.g. $P_{\max}$ are desired, further optimisations can be done: The order in which exploration and reduction takes place will impact the number of steps necessary to obtain some given accuracy. It would be beneficial to use the maximum probabilities of reaching various symbolic states to select which symbolic state should first be further refined. On the other hand, the overhead from bookkeeping to obtain accurate numbers may negate the gain. Similarly, focusing on transitions leading to symbolic states where the actual probabilities of fulfilling the formula are high would give the most useful information on the symbolic state under consideration.

In both cases, however, obtaining accurate data would in itself be rather expensive. Instead, we expect that focusing on various heuristics can lead to significant improvements in model-checking "common" models.

Since the the exploration and refinement algorithm only has one main loop, and since the WaitingBuffer is defined using the Buffer interface of the pipeline library, it is straight forward to insert heuristics for which symbolic states not to consider, and which to consider. This can be done by either changing the main loop, which is very simple or by making some AdvancedWaitingBuffer and insert it in place of WaitingBuffer.

11.3 Computing $P_{\min}$

Computing $P_{\min}$ is more involved than computing $P_{\max}$: Rather than minimise with transitions giving lower bounds, we ought to maximise with transitions giving upper bounds; we expect the sibling theorem of Theorem 3.3.9 to be true.

A naive approach would be to translate the method for $P_{\max}$ in this way; i.e. "reverse" the constraints from transitions and then obtain the maximal valuations of the variables. The problem here is that no actually interesting constraints are introduced; assigning 1.0 to each variable is a solution to the linear programming problem. While this is clearly the solution maximising the valuations of the variables, it is also completely independent of the actual probabilities of fulfilling the formula from each symbolic state.

To compute the actual probabilities, we need to find the symbolic states for which some strategy with probability 0 of fulfilling the formula exists. Setting the variables for these to 0, the solution to the linear programming problem should provide the correct probabilities.

However, rather than computing this set directly, where we may have to consider deadlocks, infinite delays and complex "loops" in transition sequences, we may instead compute the complement subset of STATES.

Algorithm 11.3.1: Finding the symbolic states for which all adversaries will fulfill some formula with positive probability.

```
GOAL := the set of symbolic states fulfilling the formula to be checked
WAITING := predecessors(GOAL)
while WAITING $\neq \emptyset$ do
    CLEARED := $\emptyset$
    foreach $s$ in WAITING do
        if successors($s$) $\subseteq$ GOAL then
            CLEARED := CLEARED $\cup \{s\}$
    GOAL := GOAL $\cup$ CLEARED
    WAITING := predecessors(CLEARED )$\setminus$GOAL;
```

11.4 From Reachability to Model Checking

Fundamentally, our translation to a linear programming problem is based on probabilistic transitions between symbolic states, with some known set of symbolic states fulfilling an atomic proposition. The translation is independent of *how* the set of symbolic states fulfilling the atomic proposition is computed. Using some threshold on probabilities, filtering out those symbolic states that meet these thresholds is easy.

A formula like

$$P_{\max}(P_{\max}(l_x) > 0.8 \vee P_{\max}(a = 11) > 0.7)$$

can be considered in parts: First, splitting and stabilising the state space according to the inner formulae; here l_x and $a = 11$.[1] Subsequently, computing $P_{\max}(l_x)$ and $P_{\max}(a = 11)$ for all symbolic states. When solving the constructed linear programming problems, values are assigned to all variables, and so obtaining these for all symbolic states rather than just the initial state will not significantly increase the computational cost. Finally, as both $P_{\max}(l_x) > 0.8$ and $P_{\max}(a = 11) > 0.7$ specifies sets of symbolic states, we may take their union — this is now the target set of symbolic states for the outer instance of $P_{\max}$, which may then be computed normally.

11.5 Probabilistic Simulator in Uppaal

In addition to editing and model-checking networks of timed automata, UPPAAL also has a part dedicated to *simulation*. Here, specific execution traces may be examined in detail. Furthermore, the normal non-deterministic reachability model checker is capable of generating *diagnostic traces* that can be loaded in the simulator.

The simulator already has a “random simulation” function; this may be extended to take probability distributions into account. To examine specific execution traces, it should also be possible to explicitly specify the result of a transition, among the results with positive probability. Between these, we may have a third kind of simulation: The user may specify the non-deterministic choices, i.e. which transitions to take; the results should then be automatically selected, with the simulator making a probabilistic choice based on probability distributions in the model.

When considering diagnostic traces, single execution traces are significantly less useful in demonstrating probabilistic reachability than in demonstrating nondeterministic reachability.

As an example, consider Figure 1.2 in the introduction. Clearly, a strategy with high probability of success exists: Retry in case of failure. This strategy cannot be demonstrated by a single execution trace. For network of probabilistic timed automata, a strategy should encompass only the nondeterministic choices. This adds complications to the data exchange format; rather than a sequence of actions — sufficient for the normal traces — what is relevant for the simulator is a mapping from symbolic states to actions.

Again, two modes of simulation may be beneficial; one with the simulator performing the probabilistic choices, and another with the probabilistic choices under the users' control.

[1]We assume that l_x specifies that some process must be in a specific location, while a is a variable which is here checked for the numeric value 11.

12 Summary

Stochastic real time systems are modelled as probabilistic timed automata. Probabilistic timed automata were defined in Chapter 4 as traditional timed automata, with the extension of discrete probabilistic edges.

To make probabilistic reachability analysis on a network of probabilistic timed automata, we require time abstract Markov equivalence quotients.

In Section 5.1, we specified State Space Exploration and Reduction Algorithm I. This algorithm symbolically explores the reachable parts of the state space of networks of probabilistic timed automata. It will, on the fly, refine the explored parts of the state space, partitioning symbolic states according to time abstract Markov equivalence. Upon completion, the algorithm has constructed a stable partitioning of the reachable parts of the symbolic state space; a time abstract Markov equivalence quotient.

We make the algorithm more concrete with State Space Exploration and Reduction Algorithm II. Here, we include pseudocode for the non-trivial parts of the algorithm; this specification is more useful when actually implementing the algorithm as a computer program.

In the UPPAAL model of timed automata, timed automata may include (bounded) integer variables. Variable updates may be specified by general user-defined procedures. If we wish to support these extensions, we need to avoid computing predecessors; we can not, in general, expect to generate the inverse of some procedure. In Chapter 7 we implement State Space Exploration and Reduction Algorithm II by the means of a predecessor graph, and a pipeline for splitting symbolic states with respect to their transitions. The predecessor graph solves the problem of predecessor generation.

Probabilistic reachability is reduced to linear programming in Chapter 6. We show that given a probabilistic reachability instance, we can construct a linear programming instance using a time abstract Markov equivalence quotient. We also show how to calculate the solution to the probabilistic reachability instance from a solution to the linear programming instance.

The implementation of probabilistic reachability solving is described in Chapter 8, where we use the implemented State Space Exploration and Reduction Algorithm II along with LP_SOLVE to calculate the maximal probability for reaching a set of states.

Networks of probabilistic timed automata are exported to PRISM models in Section 9.2. Locations and variables from UPPAAL are stores as integer variables in PRISM. PRISM does not support clocks, so federations are indexed, and stored as integer variables too. The federations stems from a time abstract Markov equivalence quotient. As a consequence of the absence of clocks, only probabilistic discrete specifications may be verified in PRISM.

The data structure called predecessor graph from Chapter 7 is exported to a graphviz

DOT file in Section 9.1.

Bibliography

[ACH$^+$92] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, David L. Dill, and Howard Wong-Toi. Minimization of timed transition systems. In *International Conference on Concurrency Theory*, pages 340–354, 1992.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BFHR92] Ahmed Bouajjani, J. C. Fernandez, Nicolas Halbwachs, and Pascal Raymond. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–269, 1992.

[EH07] Robert Jørgensgaard Engdahl and Arild Martin Møller Haugstad. State space reduction for probabilistic timed automata. Pre-master thesis thesis, December 2007.

[HKNP06] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[KNSS02] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 282:101–150, 2002.

[Olo05] Peter Olofsson. *Probability, Statistics and Stochastic Processes.* John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.

Part VI

Appendix

A Howtos

A.1 Uploading an XML file to Uppaal Prob server

Below, we give the XML encoding of $\mathcal{A}$ from Figure 9.1.[1]

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//Uppaal Team//DTD Flat System 1.1//EN'
  'http://www.it.uu.se/research/group/darts/uppaal/flat-1_1.dtd'>
<nta>
  <declaration>
// Place global declarations here.
clock x;
  </declaration>
  <template>
    <name x="5" y="5">A</name>
    <declaration>
// Place local declarations here.
    </declaration>
    <location id="id0" x="184" y="40">
      <name x="160" y="56">failure</name>
      <label kind="invariant" x="208" y="32">x&lt;=11</label>
    </location>
    <location id="id1" x="184" y="-104">
      <name x="160" y="-136">success</name>
    </location>
    <location id="id2" x="120" y="-40">
      <label kind="invariant" x="72" y="-72">x &lt;= 11</label>
    </location>
    <location id="id3" x="16" y="-40">
      <name x="-32" y="-56">start</name>
      <label kind="invariant" x="-8" y="-80">x&lt;=11</label>
    </location>
    <init ref="id3"/>
    <transition>
      <source ref="id0"/>
      <target ref="id3"/>
      <label kind="guard" x="32" y="48">x&gt;=8</label>
```

[1]this is simply the XML file one gets when saving from UPPAAL PROB or UPPAAL.

```
      <nail x="72" y="40"/>
    </transition>
    <transition>
      <source ref="id2"/>
      <target ref="id0"/>
      <label kind="assignment" x="92" y="0">x=0</label>
    </transition>
    <transition>
      <source ref="id2"/>
      <target ref="id1"/>
    </transition>
    <transition>
      <source ref="id3"/>
      <target ref="id2"/>
    </transition>
  </template>
  <system>
// Place template instantiations here.
Process = A();

// List one or more processes to be composed into a system.
system Process;
  </system>
</nta>
```

To send this XML file to the Uppaal Prob server start the Uppaal Prob server binary (`./server`) and type in `newXMLSystem3` followed by a newline, the contents of the above XML file and then finally a dot and a newline.

A.2 Exporting a Predecessor Graph to a DOT file

To export the predecessor graph of the network of timed automata given in Figure 9.1 start the Uppaal Prob server and upload the XML file for the above (see Section A.1) and type `getDotModel`

The server will then output various protocol keywords, and replace each dot with two dots — after stripping this away, we have the dot file below, which renders as Figure 9.2.

```
digraph predecessorgraph {
node [shape = circle ];
"((4121546772,0,0),0)" [ shape = record,
    label = " { ( Process.start ) |  | (x\<=11) } " ];

"((4121546804,0,0),0)" -> "((4121546772,0,0),0)";
"((4121546772,0,0),1)" [ shape = record,
    label = " { ( Process.start ) |  | (11\<x) } " ];

"((4121546772,0,0),0)" -> "((4121546772,0,0),1)";

"((4121546788,0,0),0)" [ shape = record,
    label = " { ( Process._id2 ) |  | (x\<=11) } " ];

"((4121546772,0,0),0)" -> "((4121546788,0,0),0)";

"((4121546788,0,0),1)" [ shape = record,
    label = " { ( Process._id2 ) |  | (11\<x) } " ];

"((4121546788,0,0),0)" -> "((4121546788,0,0),1)";

"((4121546804,0,0),0)" [ shape = record,
    label = " { ( Process.failure ) |  | (x\<=11) } " ];

"((4121546788,0,0),0)" -> "((4121546804,0,0),0)";

"((4121546804,0,0),1)" [ shape = record,
    label = " { ( Process.failure ) |  | (11\<x) } " ];

"((4121546804,0,0),0)" -> "((4121546804,0,0),1)";

"((4121546820,0,0),0)" [ shape = record,
    label = " { ( Process.success ) |  | true } " ];

"((4121546788,0,0),0)" -> "((4121546820,0,0),0)";
}
```

B Additional Theorems

B.1 Different Direct Delay Predecessors

In Section 5.5, we claimed that if a symbolic state source has two symbolic states as time abstract successors by the same transition, the source is unstable. This is, however, not completely obvious.

Given a configuration $c \in$ source it may be possible that one can take the same transition and end up in two different target symbolic states by means of different delays from c.

At any one point, however, if no further delay is allowed, the target of a transition is unique. Having two such targets indicates the existence of two disjoint sets of clock valuations $U, U' \subseteq$ source, where configurations contained in U will have one unique target, and the configurations contained in U' will have a different unique target. It is now enough to show that at least one configuration in source is unable to have both targets as time abstract successors by the same transition:

Proposition B.1.1
Let F be a time convex set of clock valuations, and let $U, U' \subseteq F$ be two nonempty disjoint sets of clock valuations. Then

$$F \Uparrow U \neq F \Uparrow U'. \tag{B.1}$$

Proof
Assume erroneously that

$$F \Uparrow U = F \Uparrow U'. \tag{B.2}$$

Let $u \in F \Uparrow U$. By (B.2) there must exist at least one $d \in \mathbb{R}^+$ such that $u + d \in U$. Therefore the suprememum

$$d_{\text{sup}} \overset{\text{def}}{=} \sup_{u+d \in U} d$$

is well defined.

Let $\{d_n\}_{n \geq 0}$ be an non-descending sequence in $\mathbb{R}$ that converges to d_{sup}, such that $u + d_n \in U$ for all $n \geq 0$. Note that $\{d_n\}_{n \geq 0}$ is well defined since U is not empty.

Since $u + d_n \in U$ and $U \subseteq F \Uparrow U$, it follows by (B.2) that $u + d_n \in F \Uparrow U'$ for all $n \geq 0$. But by Definition 5.2.3 there exists an $N > 0$ such that for all $n > N$ we have $u + d_n \in U'$, which violates disjointness of U and U'; a contradiction, and (B.1) follows. ∎