AALBORG UNIVERSITY

The Quotient Technique for Probabilistic Systems

Master thesis

by

Jacob Vestergaard

January 2001

AALBORG UNIVERSITY

INSTITUTE OF COMPUTER SCIENCE

Aalborg University

Institute of Computer Science - Fredrik Bajersvej 7, 9220 Aalborg Øst.

TITLE:

The Quotient Technique for Probabilistic Systems

PROJECT PERIOD:

2. September 2000 - 19. January 2001

AUTHOR:

Jacob Vestergaard

SUPERVISOR:

Anna Ingólfdóttir

NUMBER OF COPIES: 7

NUMBER OF PAGES: 79

ABSTRACT:

In this thesis we present a compositional technique, called the quotient technique, for verification of probabilistic transition systems. The quotient technique is a promising method for avoiding the state explosion problem, arising from the many possible combinations of component states. The technique works by gradually removing components from the system, while transforming the specification accordingly.
We present the theory of probabilistic alternating transition systems, system which can have both non-deterministic and probabilistic behavior, and allow these systems to have both synchronous and asynchronous behavior.
We furthermore present a probabilistic modal logic, PML, for which we define the quotient technique. This leads to an extension of the logic, a so called general modality, which we explore in detail in order apply simplification heuristics to the formulas.
The correctness of the quotient technique is justified in a formal proof, and the very important aspect of simplification of quotiented formulas is studied.
We implement the technique in Moscow ML, and explain included elements of the implementation in the report. Finally we run tests on the implementation, to evaluate the effectiveness of the quotient technique.

Aalborg Universitet
Institut for Datalogi - Fredrik Bajersvej 7, 9220 Aalborg Øst.

TITEL:
The Quotient Technique for Probabilistic Systems

PROJEKT PERIODE:
2. September 2000 - 19. January 2001

FORFATTER:
Jacob Vestergaard

VEJLEDER:
Anna Ingólfdóttir

ANTAL KOPIER: 7

ANTAL SIDER: 79

Synopsis:

I denne afhandling præsenterer vi en kompositionel teknik, kaldet kvotient teknikken, til verifikation af probabilistiske transitions systemer. Kvotient teknikken er en lovende metode til at undgå “state explosion” problemet, som opstår ud fra de mange mulige kombinationer af komponenternes tilstande. Teknikken virker ved at fjerne komponenter fra systemet, og samtidig transformere specifikationen tilsvarende.
Vi præsenterer probabilistiske alternerende transitions systemer, som er systemer som kan have både nondeterministiske og probabilistiske egenskaber, og tillader disse systemer at have både synkron og asynkron adfærd.
Vi præsenterer desuden en probabilistisk modal logik, PML, som vi definerer kvotient teknikken for. Dette leder til en udvidelse af logikken, med en såkaldt generel modalitet. Denne modalitet studerer vi i detaljer, for at sætte os i stand til at simplificere vores formler.
Korrektheden af kvotient teknikken vises i et formelt bevis, og det vigtige punkt vedrørende simplification af kvotient formler undersøges.
Vi implementerer teknikken i Moscow ML, og forklarer de inkluderede elementer fra implementationen i rapporten. Til sidst tester vi den implementerede teknik, for at kunne evaluere effektiviteten af kvotient teknikken.

Preface

This Master thesis is written by Jacob Vestergaard, and is the result of my work at the DAT6 semester at the Institute of Computer Science at Aalborg University.

Even though only one person has been working on this report, I have chosen throughout the report to use “we” instead of “I”.

I would like to thank Nicky Bodentien for always being ready to try to answer any ML-related question I have had, and also my girlfriend Louise for making me believe.

When reading this report, it is a great help if the reader has a little understanding for the programming language ML.

Aalborg, Denmark

January 19th, 2001

Jacob Vestergaard

Contents

Chapter 1

Introduction

1.1 Formal Verification

Formal verification methods are a strong tool in the development of high quality products. The presence of bugs in, for example, embedded software is very costly and can cause losses of lives. Software in devices, such as ABS breaks in cars or flight instruments, has to be as close to error free error free as possible. Formal methods provide a precise notion between systems and their specifications, so that it can be decided without ambiguity whether or not a system meets its specifications.

1.1.1 Concurrent Systems

Concurrent systems is a collection of components that are executed simultaneously while interacting with each other. Most commonly concurrent systems have non-deterministic behavior, but sometimes also have time or probability properties added. Verification of the correct behavior of concurrent systems is not an easy task to perform. First of all we need to provide a description of the system and the way it interacts, second we must have a specification of properties for the system, and a formal criterion for concluding correctness of the system. Last we need an algorithm to decide correctness of the system.

Verification is the process of checking of correctness. Kristoffersen suggests in [Kri98] a classification of computer aided verification techniques in two major trends: Theorem proving and model checking. In theorem proving, the user himself provides a formal proof of correctness, which is then checked by a tool. Model checking is fully automatic, which is why the industry tend to find this method more appealing.

1.1.2 Model Checking

Verification of large systems via model checking has become a widely used technique within the last decade. Model checking has been applied to many types of systems, ranging from finite state to real time and probabilistic systems. However a major problem arises when applying model checking to even moderate sized parallel systems. This problem is known as the state explosion problem, and arises from the many possible combinations of component states (in fact exponentially many in the number of components). Model checking of concurrent systems has been proven to be PSPACE-complete, and is therefore most likely theoretically intractable. However a lot of work has been done in the field of attacking the state explosion problem for practical systems, some with great success. Section 2.2 present some of these techniques.

1.2 Probabilistic Systems

Some systems can be designed so that they are guaranteed to behave correctly no matter what happens. In most systems, though, this requirement can not be met, as there is always a risk of power failure, hardware failure or even a failure caused by human interaction. Examples of such systems are telecommunication systems, computer networks or distributed systems built on these networks.

Due to the fact that a system can not be guaranteed to work correctly, we need a way of describing the unreliability of the system. This is especially important in safety critical systems, such as ABS breaks or flight control systems.

Probabilistic transition systems provide a framework that allows us to express that a failure can only occur with a certain probability, and as a tool it can be used to verify that the system, with some probability, behaves according to its specification (i.e. there is only 0,0001% chance that an airplanes flaps doesn't come up).

Probabilistic systems have been studied in many different forms, and in chapter 2 we present some of the existing work and the different probabilistic models.

In this thesis we have chosen to work with the so called alternating probabilistic model, which will be described later.

1.3 The Quotient Technique

This report focuses on the quotient technique, which is a promising technique for avoiding the state explosion problem. However when applying the quotient technique several other problems arise, such as very large formulas.

The idea behind the quotient technique is to factor out components of a parallel system, one at a time, and by continuously applying simplification heuristics. Consider the following model checking problem involving a system with n processes in parallel:

$$P_1 | \cdots | P_n \models \varphi$$

We wish to verify that the parallel composition of these systems satisfies φ without having to construct the complete state space of $P_1 | \cdots | P_n$. We will avoid this by removing P_i one by one while simultaneously simplifying the formula. So when factoring out P_n we will transform the formula φ into the quotient formula φ / P_n and applying simplification heuristics, such that

$$(P_1 | \cdots | P_n) \models \varphi \Leftrightarrow (P_1 | \cdots | P_{n-1}) \models (\varphi / P_n)^s,$$

where s denotes simplification of the formula.

The quotient technique has been studied for several years now, and has been proven to be successful for finite state systems and real-time systems. The technique has also been applied to Hierarchical State-event systems.

As an example of the quotient technique assume that we want to prove

$$\overbrace{a.NIL | a.NIL | \cdots | a.NIL}^{n} \models \langle a \rangle tt$$

where $|$ denote parallel composition of CCS.

Clearly it seems a waste to examine the entire state space (2^n states) to establish this simple property. Using the quotient technique this may be avoided:

$$\begin{array}{rl} & \overbrace{a.NIL | a.NIL | \cdots | a.NIL}^{n-1} \models (\langle a \rangle tt) / a.NIL \\ \text{Quotient+Simpl.} \Updownarrow & \\ & \underbrace{a.NIL | a.NIL | \cdots | a.NIL}_{n-1} \models tt \end{array}$$

So we have avoided examining 2^n states, but yet proved that the system satisfied the property.

A formal presentation of the quotient technique for finite state systems is given in Appendix A.

In this thesis we define the quotient technique for probabilistic alternating transition systems, and implement the technique in ML. Our main goal is to examine the technique, the formulas and some simplification rules, in order to provide a working model checker using the quotient technique. We will test this implementation, to verify that it is indeed a promising method for avoiding the state explosion problem.

1.4 Outline

The outline of this report is as follows.

In the next chapter we present the existing related work, and gives examples and definitions of other probabilistic models.

Chapter 3 is an introduction to probability theory.

In chapter 4 we define probabilistic alternating transition systems, and give a probabilistic process calculus. We have chosen to be able to express asynchronous communication in our transition systems, a not so straight forward application in the alternating model. The motivation for and consequences of this choice are also described in this chapter. Finally we give a probabilistic modal logic PML, that allows us to express properties of such systems.

Chapter 5 introduces the quotient technique to the alternating probabilistic model. We define the quotient rules, and show that our initial logic needs to be expanded with a more general modality in order to support the technique. We then give proof of correctness of the quotient rules and give a small example of the technique.

Chapter 6 is dedicated to the discussion of the general modality.

In chapter 7 we define a set of simplification rules and prove that they are sound with respect to the semantics.

In Chapter 8 our model checker is presented and in chapter 9 we present an example and run tests on the implemented checker.

Chapter 10 ends this report with summary, conclusions and ideas for further research in the area of probabilistic transition systems and the quotient technique.

Throughout the report we have chosen to include extracts from our implementation and corresponding explanations and comments to this.

Chapter 2

Related Work

In this chapter we take a look at some of the existing work in the field of probabilistic systems and model checking in general.

2.1 Probabilistic Models

We present the work in the field of probabilistic processes and transition systems. In [vGSST90] van Glabbeek, Smolka, Steffen and Tofts classify probabilistic processes in three types: Reactive, generative and stratified models.

- Reactive Model
  The reactive model consists of states and labelled transitions associated with probabilities. For each state, the sum of probabilities on outgoing transitions must be 1 for transitions with the same label.

- Generative Model
  This model consists also of states and labelled transitions with probabilities, but with the sum of probabilities of *all* outgoing transitions equal to 1.

- Stratified Model
  Stratified models consist of states and two kinds of transitions, probabilistic and action based. In the case of probabilistic transitions, the sum of probabilities must be 1, and for the actions transitions the restriction is that there must be only one outgoing action transition from a state.

In the following we will present the results on these three models, and give a formal definition of the first two.

Though van Glabbeek et al. only distinguish between the three models mentioned, this shall not be seen as the only probabilistic models available. This report focuses on the alternating model which, to our knowledge, was first studied by Hansson and Jonsson in [HJ89] and which is derived from concurrent Markov chains. Later in this report we define probabilistic alternating transition systems, and give a full definition of a probabilistic calculus for the alternating model.

2.1.1 The Reactive Model

In [LS91], Larsen and Skou define a probabilistic bisimulation based on the reactive model, and in the same reference, the authors provide a probabilistic logic based on HML, which they call probabilistic modal logic (PML).

In [LS92], Larsen and Skou define a reactive probabilistic transition system as follows:

Definition 2.1
A (reactive) probabilistic transition system is a structure $\mathcal{P} = (Pr, Act, \pi)$, where Pr is a set of processes (or states), Act is the set of actions that the processes may perform, and π is a transition probability function $\pi : Pr \times Act \times Pr \to [0,1]$ such that for each $P \in Pr$ and $a \in Act$:

$$\sum_{P' \in Pr} \pi(P, a, P') = 1 \textit{ or } \sum_{P' \in Pr} \pi(P, a, P') = 0$$

indicating the possible next states and their probabilities after P has performed the action a.

In figure 2.1 is an example of a reactive process.

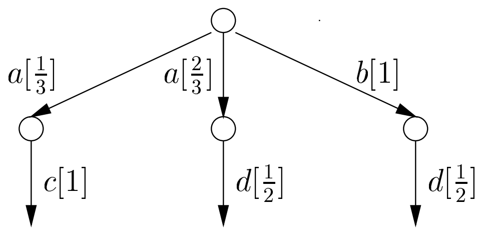

Figure 2.1: An example of a reactive process

In [LS92] the authors develop a synchronous calculus based on the reactive model. They use a probabilistic choice operator parameterized by a probability, to obtain probabilistic behavior. They show that when decomposing a PML formula, the logic PML is not strong enough to express parallel decomposition, and present an extension to PML, called EPL, which support decomposition. They further axiomatize the extended logic EPL.

2.1.2 The Generative Model

The generative models define context dependent probability distributions, and the probabilities have to be calculated every time an action is received. The generative model has been formally defined by Jou and Smolka in [JS90] as follows:

Definition 2.2
A (generative) probabilistic transition system (PLTS) is a triple $\langle Pr, \sum, \mu\rangle$ where:

- *Pr is the set of all processes;*
- *Σ is the set of all atomic actions, and 0 is a special symbol not in Σ called the zero action;*
- *$\mu : (Pr \times (\Sigma \cup \{0\}) \times Pr) \to [0,1]$ is a total function called the probabilistic transition function satisfying the following restriction: $\forall P \in Pr$,*

$$\sum_{a \in \Sigma \cup \{0\}, Q \in Pr} \mu(P, a, Q) = 1$$

An example of the generative model can be seen in Figure 2.2

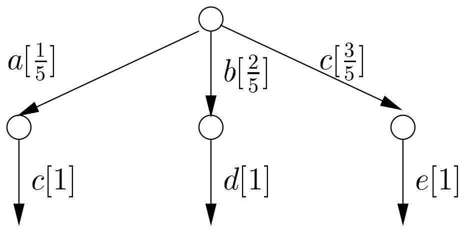

Figure 2.2: Example of a generative process

2.1.3 The Stratified Model

In the stratified model, pure probabilistic choices can be made. Glabbeek et al. defines in [vGSST90] stratified operational semantics for a probabilistic process calculus (PCCS). The calculus is separated in two parts, action transitions and probability transitions, which enables the use of pure probabilistic choices. The two types of transitions are denoted $P \xrightarrow{a} Q$ and $P \xrightarrow{p} Q$, where p is the probability that P can behave as Q. The sum of all outgoing probabilistic transitions from a state is 1, thereby making the transition system stochastic. Glabbeek et al. also provides a bisimulation for the stratified model, called stratified bisimulation.

An example of the stratified model can be seen in figure 2.3

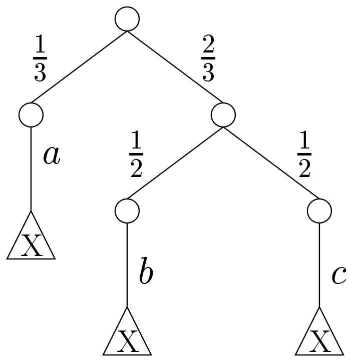

Figure 2.3: Example of a stratified transition system

In [vGSST90] the authors furthermore form a hierarchy of the probabilistic models. They show that the generative model is an abstraction of the stratified model, and that the reactive model is an abstraction of the generative model.

2.1.4 Other Probabilistic Models

Apart from the reactive, generative and stratified models, other probabilistic models have been studied. In [HJ89] and [Ves00] the authors consider the alternating model, which is also the model used in this report, and therefore is described later. Hansson and Jonsson present a CTL like logic in [HJ89], in order to be able to describe properties like “After a request, there is a 90% probability that the request will be carried out in 2 seconds”.

The alternating model originates, to our knowledge, from the joint work between Hansson and Jonsson in 1989, which is presented along with the work in [HJ89] in Hans Hanssons book [Han94].

2.2 Model Checking

In this section we look at some of the existing model checking techniques. We distinguish between model checking for finite state systems, real-time systems and probabilistic system.

2.2.1 Finite State Systems

Several techniques have been applied to finite state systems with great success. One such technique is based on Binary Decision Diagrams (BDD), proposed by Bryant in [Bry86]. BDD's provide a canonical form for boolean functions that are often more compact than formulae on conjunctive and disjunctive normal form. Several efficient algorithms have been developed for manipulating formulae based on their BDD representation, and the model checking tool SMV is based on BDD's.

Partial Order Reduction is another attack on the state explosion problem with promising results. This method is used by the tool SPIN. Compositional Backwards Reachability (CBR) is a technique which has had great success. Tests with applying the CBR technique to large concurrent systems have proven that CBR definitely is a good way to attack the state explosion problem. The CBR technique is used in the commercial tool VisualSTATE, which uses the state event model. The last technique we will mention for finite state systems is subject of this report, the quotient technique. Larsen was one of the first to propose this technique in [Lar86], and in Appendix A we give example of the technique used on a simple HML logic. The quotient technique has also been applied to State-Event systems in [NJJ$^+$99], a work which has been greatly extended in [BP00].

2.2.2 Real-Time Systems

Methods for avoiding the state explosion problem in real-time systems include Difference Bound Matrix (DBM) an efficient data-structure for the time space and the rather new data-structure Clock Decision Diagrams (CDD), which can handle both discrete control space and continuous time space symbolically. Of model checking tools for real-time systems, we can mention Kronos [kro], Hytech and UPPAAL [BLL$^+$95]. In [Seg95], Segala builds a framework for verification of randomized distributed real-time systems, systems with both timed and probabilistic properties. The quotient technique has also been studied for real-time systems, by Laroussinie and Larsen in [LL95] and by Andersen in [rHA97].

2.2.3 Probabilistic Systems

The state explosion problem in probabilistic transition systems, has so far been attacked by extending Binary Decision Diagrams (BDD), to treat probabilistic transition systems. Bozga and Maler introduce Probabilistic Decision Graphs (PDG) in [BM99], and in [BCGH$^+$97] Bahar et.al. apply Multi-terminal BDD's (MTBDD) to probabilistic verification.

The quotient technique for probabilistic transition systems has been studied by Larsen & Skou in [LS92] but only for a reactive model, and with no direct intension of applying it to model checking. They introduced a simple calculus of probabilistic processes and a probabilistic modal logic. In their paper they study the problem of applying the quotient technique (or decomposition) and identify a new extended probabilistic logic, which is needed to support the technique. Furthermore they give complete axiomatization for both the calculus and the logic.

The present report extends the work in [Ves00], in which the quotient technique for probabilistic alternating transition systems was first introduced.

Chapter 3

Preliminaries

Before we can define probabilistic alternating transition systems and a probabilistic process calculus, we need some general results on probabilities. The results in this chapter is mainly extracted from DeGroot's "Probability and Statistics" [DeG89].

3.1 Probability Theory

In this section we give an axiomatic definition of the term probability, and give a few important consequences of the axioms.

First we need the notion of *sample space*. A sample space of an experiment is a collection of all the possible outcomes of the experiment. A sample space can be thought of as a set, or collection, of different possible outcomes, and each outcome can be thought of as a point, or an element, of the sample space.

As an example consider a roll with a six sided die, then the sample space can be written $S = \{1, 2, 3, 4, 5, 6\}$. An *event* of an experiment occurs when the outcome of the experiment satisfies certain conditions specified by that event. So an event $A \subseteq S$ that an even number is obtained in our die example is $A = \{2, 4, 6\}$.

We define the probability function π as follows:

Definition 3.1
The probability function π is a function from the sample space S to a number between 0 and 1:

$$\pi : S \to [0, 1]$$

In a given experiment we assign each event A in the sample space S with a number $\Pi(A)$, which is the probability that A will occur. The number $\Pi(A)$ must satisfy three axioms in order to satisfy the mathematical notion of probability. These axioms ensure certain properties that a probability is expected to have.

The first axiom states the fact that the probability Π of any event A, denoted $\Pi(A)$, has to be non-negative.

Axiom 1
For any event A, $\Pi(A) \geq 0$.

The next axiom states that if an event is certain to occur, then the probability of that event is 1.

Axiom 2
$\Pi(S) = 1$.

Axiom 3
For any infinite sequence of disjoint events $A_1, A_2, \ldots$,

$$\pi\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \pi(A_i).$$

We can now formally define probability.

Definition 3.2 (Probability)
A probability distribution, or a probability, on a sample space S is a specification of numbers $\pi(A)$ which satisfy Axioms 1, 2 and 3.

We shall now give a few important consequences of the axioms, starting by showing that if an event is impossible, then the probability of that event is 0.

Theorem 3.3
$\Pi(\emptyset) = 0$.

Proof
Consider the infinite sequence of events $A_1, A_2, \ldots$, such that $A_i = \emptyset, i = 1, 2, \ldots$. Then this sequence is a sequence of disjoint events, since $\emptyset \cap \emptyset = \emptyset$. Furthermore, $\cup_{i=1}^{\infty} A_i = \emptyset$. Therefore, it follows from Axiom 3 that

$$\Pi(\emptyset) = \Pi\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \Pi(A_i) = \sum_{i=1}^{\infty} \Pi(\emptyset).$$

So when $\Pi(\emptyset)$ is added in an infinite series, the sum of that series is the number $\Pi(\emptyset)$. The only number with this property is $\Pi(\emptyset) = 0$. $\square$

We state another general theorem, which can easily be proved.

Theorem 3.4
For any event A, $0 \leq \Pi(A) \leq 1$.

For a given probability function Π on a finite sample space S, let Π be defined by

$$\Pi(A) = \sum_{a \in A} \pi(a)$$

It is not difficult to see that Π defined this way is a probability distribution on $\mathcal{P}(S)$, the set of all subsets of S_1. Usually we use π instead of Π, if the meaning is clear from the context.

Chapter 4

Probabilistic Transition Systems

In this chapter we define probabilistic alternating transition systems. Although we work with the quotient technique, for which we only need a parallel operator, we also give a probabilistic process calculus. We also define a probabilistic modal logic, for describing properties in our transition systems.

4.1 PTS

The idea of a probabilistic alternating transition system is that we have two kinds of states, probabilistic and non-deterministic. Only in the probabilistic states can the transition system take a probability transition, and in the non-deterministic states, the system behaves like a normal non-deterministic process, i.e. by performing some action.

An example of an probabilistic alternating transition system can be seen in figure 4.1, and is formally defined in definition 4.1.

The formal definition of a probabilistic alternating transition system is given as follows:

Definition 4.1 (Probabilistic Alternating Transition System (PTS)) *Let Act be a set of actions. A probabilistic alternating transition system is a triple $\langle S, \longrightarrow, \pi_0 \rangle$, where*

- *S is a non-empty set of states*
- *$\longrightarrow \subseteq S \times Act \times Dist(S)$ is a finite transition relation*
- *$\pi_0 \in Dist(S)$ is an initial distribution on S*

We shall use $P \xrightarrow{a} \pi$ to denote that $\langle P, a, \pi\rangle \in\longrightarrow$, and $P \overset{a}{\not\rightarrow}$ to denote that $\langle P, a, \pi\rangle \notin\longrightarrow$, for all π. We will sometimes write $\pi \rightsquigarrow_{\mu} P$ instead of $\pi(P) = \mu$.

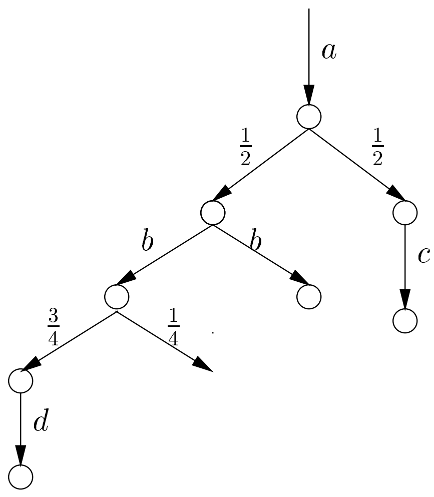

Figure 4.1: A probabilistic alternating transition system

In the next section we give a probabilistic process calculus, with an asynchronous parallel operator. When composing transition systems later on we use this parallel operator

We will later show that parallel composition of probabilistic systems is symmetric (theorem 4.8), that is, the order of which processes or systems are composed does not matter.

4.2 Probabilistic Process Calculus for PTS

We will in this section give a probabilistic process algebra, very similar to the classic process calculus CCS. Probabilistic extensions to several classic process calculi have been studied for many years, but we will here give a calculus that, to our knowledge, differs from the ones studied by others. Unlike other calculi, the process calculus for PTS is split in two, and consists of two different types of terms, namely process terms ranged over by P, which have non-deterministic behavior, and probabilistic terms ranged over by π. The main reason for this split-up is, that when implementing PTS it will be easier to differentiate between process and probabilistic terms.

Although this report concentrates on the quotient technique, and therefore on the parallel operator, we chose to give a full probabilistic process calculus. We do this mainly to show what a calculus for the alternating model could look like, so that this provides a basic framework, if others are interested in exploring this model.

4.2.1 Syntax of PTS

We start out by giving a syntax for describing probabilistic transition systems. The syntax consists of a NIL operator, a choice operator, a prefix, a parallel operator and the special probabilistic choice operator.

The syntax is given in definition 4.2

Definition 4.2

$$
\begin{aligned}
P &::= NIL \mid P_1 + P_2 \mid a.\pi \mid P_1|_{\mathcal{A}}P_2 \mid N && (4.1)\\
\pi &::= \pi_{NIL} \mid \pi_1 + \pi_2 \mid \pi_{a.\pi} \mid \pi_1|_{\mathcal{A}}\pi_2 \mid \pi_1 \oplus_{\mu} \pi_2 && (4.2)\\
& && (4.3)
\end{aligned}
$$

where $\mathcal{A}$ is a set of actions that the system synchronizes on, where $N \overset{Def}{=} P$.

4.2.2 Semantics of PTS

The semantics of PTS is given in terms of two types of judgments:

$$
\begin{aligned}
&P \xrightarrow{a} \pi, && \text{where } a \in \mathcal{A}\\
&\pi \leadsto_{\mu} P, && \text{where } \mu \in [0,1]
\end{aligned}
$$

The last is, as described before, just another way of writing $\pi(P) = \mu$.

We refer to the first as process transitions and the latter as probabilistic transitions.

Formal inference rules of P and π can be found in table 4.1, and are further explained in the following.

Inference Rules for nondeterministic transitions

- NIL denotes a state with no outgoing transitions, hence no rule.
- The non-deterministic choice operator is a choice between the transitions of the two arguments.

Nondeterministic			Probabilistic	
NIL	NIL		π_{NIL}	$\pi_{NIL} \rightsquigarrow_1$ NIL
Parallel1	$\dfrac{P_1 \xrightarrow{a} \pi_1 \; P_2 \xrightarrow{a} \pi_2}{P_1 \vert_{\mathcal{A}} P_2 \xrightarrow{a} \pi_1 \vert_{\mathcal{A}} \pi_2}$	if $a \in \mathcal{A}$	Parallel	$\dfrac{\pi_1 \rightsquigarrow_{\mu_1} P_1 \; \pi_2 \rightsquigarrow_{\mu_2} P_2}{\pi_1 \vert_{\mathcal{A}} \pi_2 \rightsquigarrow_{\mu_1 \cdot \mu_2} P_1 \vert_{\mathcal{A}} P_2}$
Parallel2	$\dfrac{P_1 \xrightarrow{a} \pi_1}{P_1 \vert_{\mathcal{A}} P_2 \xrightarrow{a} \pi_1 \vert_{\mathcal{A}} \pi_{P_2}}$	if $a \notin \mathcal{A}$		
Prefix	$a.\pi \xrightarrow{a} \pi$		Prefix	$\pi_{a.\pi} \rightsquigarrow_1 a.\pi$
Choice1	$\dfrac{P_1 \xrightarrow{a} \pi}{P_1 + P_2 \xrightarrow{a} \pi}$		Choice	$\dfrac{\pi_1 \rightsquigarrow_{\mu_1} P_1 \; \pi_2 \rightsquigarrow_{\mu_2} P_2}{\pi_1 + \pi_2 \rightsquigarrow_{\mu_1 \cdot \mu_2} P_1 + P_2}$
Choice2	$\dfrac{P_2 \xrightarrow{a} \pi}{P_1 + P_2 \xrightarrow{a} \pi}$		Prob.Choice	$\dfrac{\pi_1 \rightsquigarrow_{\mu_1} P \; \pi_2 \rightsquigarrow_{\mu_2} P}{\pi_1 \oplus_{\mu} \pi_2 \rightsquigarrow_{\mu \cdot \mu_1 + (1-\mu) \cdot \mu_2} P}$

Table 4.1: Inference rules for P and π

- The prefix operator, $a.\pi$ performs an a-transition and goes to state π.
- When composing two non-deterministic transitions in parallel, we need to determine whether the action to be taken is part of our synchronizing set $\mathcal{A}$ or not. If $a \in \mathcal{A}$ then the system synchronizes, and both machines have to be able to take an a transition. If $a \notin \mathcal{A}$ then it suffices to have only one machine being able to take the a transition. How this works is explained in section 4.3.

Inference rules for probabilistic transitions

- We define the probabilistic version of π_{NIL} to have a probabilistic transition with probability 1 to the process NIL.
- Probability states always synchronizes on probabilistic transitions, so compared to process states, there is only one rule for parallel composition here.
- The probabilistic transition for $\pi_{a.\pi}$ is similar to that for π_{NIL}, with a probabilistic transition with probability 1 to the process prefix.

- The choice operator is not resolved by probabilistic transitions but by process transitions. This explains the choice operator for probabilistic transitions in table 4.1.

- Probabilistic choice is a binary operator which specifies the probabilistic transitions.

4.2.3 Example of PTS

As an example of PTS, consider the following expressions:

$$A = a.\left(\frac{1}{3}.b \oplus \frac{2}{3}.c\right) \quad B = a.\left(\frac{1}{2}.b \oplus \frac{1}{2}.d\right)$$

The two transition systems can be found in figure 4.2. We have also included the parallel composition $A|_{\mathcal{A}}B$ in our example, which can also be found the figure below.

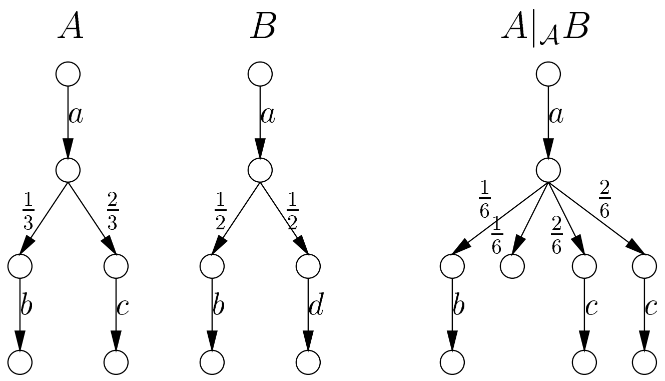

Figure 4.2: The example PTS

4.3 Asynchronous Parallel Composition

We have included asynchronicity in our transition systems, by allowing some process transitions to be asynchronous. This has not been straightforward, and has taken a lot of consideration. We will in this section discuss the subject of asynchronous probabilistic transition systems, especially in the case of the alternating model, which leads to the model we have chosen to use, and why.

In finite state systems asynchronous composition is straight forward. Either process can take a transition independently of the other, with the resulting composition still well defined, as in the following

$$\frac{P \xrightarrow{a} P'}{P|Q \xrightarrow{a} P'|Q}$$

The problem with asynchronicity in the alternating probabilistic model is the fact that the transitions alternate. When allowing process transitions to be asynchronous the resulting composed system ends up in two different kinds of states, a non-deterministic state and a probabilistic state. This composition is not defined in our calculus, so we have to define a means of expressing this situation. The problem can be exemplified by figure 4.3. If we compose P and Q, and let $\mathcal{A} = \{a, c\}$, we can see the problem in states s_1 and s_2, P is allowed to take the b transition, but Q has to stay in state s_2. If we do this, the next composition will be transitions from state s_3 and s_2, two different kinds of states. This is not allowed in our calculus, hence the need for a way of expressing this.

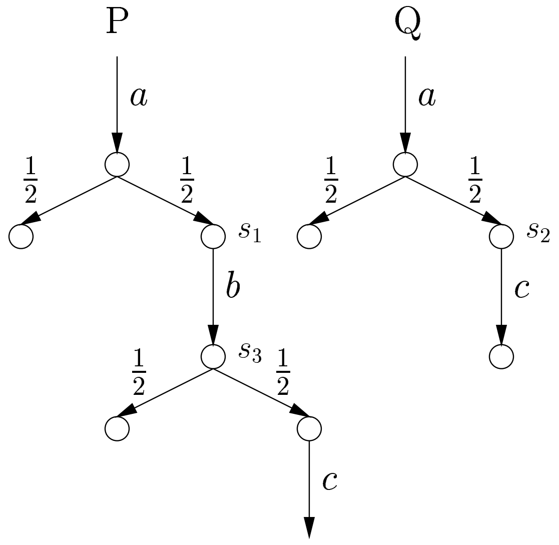

Figure 4.3: Two transition systems. The b transition is asynchronous.

Other probabilistic models do not have the same problem, for example the reactive model, as described by Larsen and Skou in [LS92], only has one kind of states, and with the transitions being a combination of both probability and actions ($P \xrightarrow{a}_{\mu} P'$).

There are different ways of attacking this problem. One way of doing it would be to include this mixed composition in our calculus, as in $P|_{\mathcal{A}}\pi$. This is not desirable though, because it would destroy the meaning of having a split-up calculus.

Instead we have chosen to allow asynchronous composition with the help of a fictive distribution. We simply introduce a helping distribution, which with probability 1 can reach the non-deterministic state. As an example, consider figure 4.4.

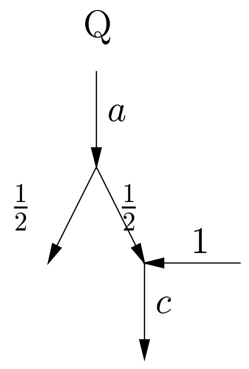

Figure 4.4: We solve the asynchronous problem by adding a 1 transition.

So we can view the parallel composition $P|\pi$ as $\pi_P|\pi$, where π_P is a distribution, which with probability 1 takes a transition to P, that is

$$\pi_P(Q) = \begin{cases} 0, & P \neq Q \\ 1, & P = Q \end{cases}$$

or equivalently $\pi_P \rightsquigarrow_1 P$. This effectively solves the problem, and the resulting parallel composition of P and Q can be seen in figure 4.5.

4.4 Probabilistic Modal Logic

In this section we will give a probabilistic modal logic for our transition systems. The logic is HML-like, but split in two parts, non-deterministic and probabilistic properties, ranged over by F and φ respectively. This split-up makes it easier to apply the quotient technique to the alternating probabilistic model, and to implement it.

First we give the syntax for the logic, and then its semantics is defined. We have chosen to give two different, but equivalent versions of the semantics

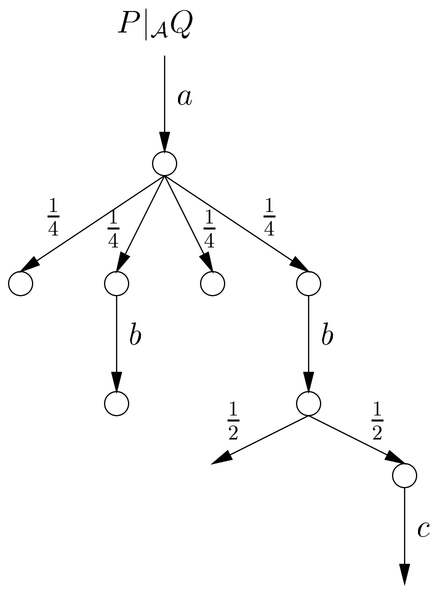

Figure 4.5: The final parallel composition $P|_{\mathcal{A}}Q$

for the logic, a semantics based on a satisfiability relation and a denotational semantics. The satisfiability relation semantics can sometimes be easier to read, but later when proving simplification rules, the denotational semantics proves to be useful. In fact the proofs of some of the simplification rules follow almost directly from the formulation in terms of denotational semantics.

4.4.1 Syntax

As mentioned above, the syntax is divided into two parts which refer to each other by their diamond modality. The non-deterministic part of the logic is like normal HML logic, with the only exception that in the diamond modality it does not refer to a non-deterministic property, but to a probabilistic one.

Definition 4.3 (Probabilistic Modal Logic (PML))
The non-deterministic (ranged over by F) and probabilistic (ranged over by φ) properties are defined as follows:

$$F ::= tt \mid F_1 \wedge F_2 \mid \neg F \mid \langle a \rangle \varphi$$

$$\varphi ::= tt \mid \varphi_1 \wedge \varphi_2 \mid \neg \varphi \mid \diamond_{\geq \mu} F$$

We shall later see that the logic is not strong enough to describe certain properties when factoring out processes using the quotient technique. Actually we need to extend the logic with a more general modality of the form $[\diamond_{x_1}F_1, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]$, as will be explained in more detail later.

4.4.2 Semantics

Definition 4.4 (Satisfiability for PML***)***
We define $\models \subseteq (Pr \times PML) \cup (Dist(Pr) \times PML)$ *inductively as follows*

$$
\begin{array}{lll}
P \models tt & \textit{iff} & P \in Pr \\
P \models F_1 \wedge F_2 & \textit{iff} & P \models F_1 \text{ and } P \models F_2 \\
P \models \neg F & \textit{iff} & P \not\models F \\
P \models \langle a \rangle \varphi & \textit{iff} & \exists \pi . P \xrightarrow{a} \pi \wedge \pi \models \varphi \\
 & & \\
\pi \models tt & \textit{iff} & \pi \in Dist(P) \\
\pi \models \varphi_1 \wedge \varphi_2 & \textit{iff} & \pi \models \varphi_1 \text{ and } \pi \models \varphi_2 \\
\pi \models \neg \varphi & \textit{iff} & \pi \not\models \varphi \\
\pi \models \diamond_{\geq \mu} F & \textit{iff} & \sum_{P.P \models F} \pi(P) \geq \mu .
\end{array}
$$

We now give the equivalent denotational semantics for PML. As written in the introduction, certain properties are easier to reason about using denotational semantics.

Definition 4.5 (Denotational semantics for PML***)***
First the definition of $[\![F]\!]$

$$
\begin{array}{lll}
[\![tt]\!] & = & Pr \\
[\![F_1 \wedge F_2]\!] & = & [\![F_1]\!] \cap [\![F_2[\![\\
[\![\neg F]\!] & = & \overline{[\![F]\!]} \\
[\![\langle a \rangle \varphi]\!] & = & \langle \cdot a \cdot \rangle \{\!|\varphi|\!\} .
\end{array}
$$

The operator $\langle \cdot a \cdot \rangle \varphi$ *is defined as* $\{p \in Pr | \exists a \in A, \pi . p \xrightarrow{a} \pi \wedge \pi \models \varphi\}$.

The semantics for φ *are defined as*

$$
\begin{array}{lll}
\{\!|tt|\!\} & = & Dist(P) \\
\{\!|\varphi_1 \wedge \varphi_2|\!\} & = & \{\!|\varphi_1|\!\} \cap \{\!|\varphi_2|\!\} \\
\{\!|\neg \varphi|\!\} & = & \overline{\{\!|\varphi|\!\}} \\
\{\!| \diamond_{\geq \mu} F |\!\} & = & \{\pi | \pi([\![F]\!]) \geq \mu\} .
\end{array}
$$

As a consequence of the definition, we state the following theorem:

Theorem 4.6
The following expressions are equivalent:

$$\langle a\rangle(\varphi_1 \wedge \varphi_2) \equiv \langle a\rangle\varphi_1 \wedge \langle a\rangle\varphi_2.$$

We can now define a connection between transitions and formulas, by defining the semantics of process and probabilistic transitions in the following way:

Definition 4.7

$$\langle\langle P\rangle\rangle = \{F|P \models F\}$$
$$\langle\langle \pi\rangle\rangle = \{\varphi|\pi \models \varphi\}$$

We state and prove the following property for parallel composition, which will be useful for describing some important properties for the quotient technique.

Theorem 4.8 (Associativity and commutativity of parallel composition)

$$\langle\langle p \mid q\rangle\rangle = \langle\langle q \mid p\rangle\rangle$$
$$\langle\langle (p \mid q) \mid r\rangle\rangle = \langle\langle p \mid (q \mid r)\rangle\rangle$$
$$\langle\langle \pi_1 \mid \pi_2\rangle\rangle = \langle\langle \pi_2 \mid \pi_1\rangle\rangle$$
$$\langle\langle (\pi_1 \mid \pi_2) \mid \pi_3\rangle\rangle = \langle\langle \pi_1 \mid (\pi_2 \mid \pi_3)\rangle\rangle$$

Proof
If we consider the transition trees for the different compositions, we see that they are isomorphic, which is enough for theorem 4.8 to hold. □

4.5 Implementation

In this section we describe the implementation of probabilistic transition systems, and the probabilistic logic we have defined.

We have chosen to use the programming language Moscow ML to implement our systems and techniques. ML is a powerful functional programming language, which suits our needs perfectly.

We start by developing a data type for the probabilistic transition systems, including a parallel operator.

Next we define a data structure for PML, and a model checker for verifying PML properties in any probabilistic alternating transition system.

4.5.1 Datatypes

We start by defining a few simple types, mainly for making reading easier. We define key concepts as probability, action and state as well as an index to distinguish between different components of a parallel system.

```
type state = string;
type probability = real;
type identifier = string;
type action = string;
type index = int;
```

We define two datatypes for transitions, which are both parameterized by a state type, `PTrans` for probabilistic transitions which are defined by a list of states associated with a list of probabilities and target states. `NTrans` is the datatype for process transitions, defined in same manner as probabilistic transitions, but with actions instead of probabilities.

```
datatype  'state NTrans = transrelN of
          ('state * ((action * 'state) list)) list;
datatype  'state PTrans = transrelP of
          ('state * ((real * 'state) list)) list;
```

The last datatype we need for PML is `System`, which defines a probabilistic alternating transition system of given process and probabilistic transitions. Also the datatype for the parallel operator $|_{\mathcal{A}}$, named `||` in the implementation, is defined as a composition of two systems. Note that `||` is made infix as to match our syntax.

```
infix 5 ||

datatype 'state System = system of
         ('state NTrans) * ('state PTrans) * 'state * index
       | || of 'state System * 'state System;
```

Besides a list of process transitions and probability transitions, the datatype `System` is also defined by a start state and an index which is the systems "number", as mentioned above the means of distinguishing the different subsystems of a parallel system.

The Datatypes for our probabilistic modal logic is like the formal definition split in two parts. We have chosen to include the terms *False* and *Or* in the implementation of the datatype, because it makes it easier to read and specify formulas including these terms (e.g. *ff* instead of $\neg tt$).

```
datatype nonformula = nAp of state * index
                    | nFalse
                    | nTrue
                    | nAnd of nonformula * nonformula
                    | nOr of nonformula * nonformula
                    | nNot of nonformula
                    | nDiamond of action * probformula
and probformula = pAp of state * index
                | pFalse
                | pTrue
                | pAnd of probformula * probformula
                | pOr of probformula * probformula
                | pNot of probformula
                | pDiamondsimp of probability * nonformula
                | pDiamond of
                 ((probability * nonformula) list *probability)

infix 7 nAnd pAnd nOr pOr
```

The boolean operators *And* and *Or* have been made infix, again to match the syntax.

We have implemented the general diamond modality by a list of probability $*$ formula with a corresponding probability (μ). We do not explicitly include the dependency variables, but take them into account when using the modality.

4.5.2 The Simple Checker

To be able to check the correctness of our implementation of the quotient technique later on, we implement a simple model checker, which can determine if a system satisfies any formula. This model checker takes any system and a formula, and checks if the formula is satisfied, by going through all the states in the system. It should be clear that if the systems have many parallel components, each with a considerable amount of states, then the simple model checker fails to perform well because of the state explosion.

We start by defining a function `der` (derivative) which takes any system with a configuration `c` and a list `A` and returns a list of probabilities or actions that the system is able to take in the state given in the configuration, and the corresponding target state.

It is also in the function `der` that the parallel operator is defined, including both synchronous and asynchronous compositions depending on whether or not the action in the transition is included in the list `A`. A configuration is simply the current state of the system.

```
fun der ((system (ntrans, ptrans, s0, n)):string System) c A =
    let val (s,_) = find (fn (s',i) => i = n) c
    in let val nlist = derN s ntrans
           val plist = derP s ptrans
       in (map (fn (a,t)=>(a,[(t,n)])) nlist,
           map (fn (p,t)=>(p,[(t,n)])) plist)
       end
    end
  | der ( S1 || S2 ) c A =
    let val (nlist1,plist1) = der S1 c A
        val (nlist2,plist2) = der S2 c A
        val source1 = subconfig c S1
        val source2 = subconfig c S2
    in
        let val nlist =
            (reduce (fn ((a1,t1),restofMerge1) =>
                     if (memberof a1 A)
                        then (reduce
                              (fn ((a2,t2),restofMerge2) =>
                               (case ((memberof a2 A),(a1=a2))
                                  of (true,true) => [(a1,t1@t2)]
                                  | (_,_) => []) @ restofMerge2)
                              nil
                              nlist2)
                            @ restofMerge1
                     else (a1,(t1@source2)):: restofMerge1)
             nil
             nlist1)
            @
            (reduce (fn ((a2,t2),restofasync) =>
                     if not (memberof a2 A)
                        then (a2,source1@t2):: restofasync
                     else restofasync)
             nil
             nlist2)
            val plist = reduce
                (fn ((p1:real,t1),restofMerge1) =>
                 (reduce (fn ((p2,t2),restofMerge2) =>
                          [((p1*p2),t1@t2)] @restofMerge2)
                  nil
                  plist2)@restofMerge1)
                nil
                plist1
        in (nlist,plist)
        end
    end
```

The function `der` uses a few small helping functions, the functions `derN` and `derP` take as arguments a state *s* and a transition relation, and returns the outgoing transitions from state s. `subconfig` takes a global configuration and a system and returns the specific configuration for that system.

The `memberof` function takes as argument an action and a set, runs through the set ($\mathcal{A}$) and returns true if the action is found in the set.

Some other basic functions are also used, `map`, `filter`, `find` and `reduce`, the first being a standard ML function and the three others defined as follows:

```
fun find f nil = raise notfound
  | find f (h::t) = if f h then h
                    else find f t;

fun reduce f b nil = b
  | reduce f b (h::t) = f (h,reduce f b t);

fun filter f nil = nil
  | filter f (h::t) = if f h then h::filter f t
                      else filter f t;
```

We are now ready to define our simple model checker. First we define a mutually recursive function `nSatInner` and `pSatInner`, which take as arguments the following:

- Any transition system of the datatype `System` which can also be a parallel composition.
- A PML formula (starting with a process or a probability expression, respectively)
- A configuration
- A synchronization set $\mathcal{A}$.

The function scans the formula and handles each term recursively, and makes use of the function `getactivestate`, which, given a configuration and an index, returns the current state being examined. First we present the part that handles process formulas.

```
fun nSatInner S nForm c A =
    case nForm of
        (nAp (t,i)) => (getactivestate c i) = t
      | (nTrue) => true
      | (nFalse) => false
      | (nNot nform) => not (nSatInner S nform c A)
      | (lnon nAnd rnon) => (nSatInner S lnon c A)
                            andalso
                            (nSatInner S rnon c A)
      | (lnon nOr rnon) => (nSatInner S lnon c A)
                           orelse
                           (nSatInner S rnon c A)
      | (nDiamond (a,pForm)) =>
        (case (find (fn (act,c') => (act=a)
                                    andalso
                                    (pSatInner S pForm c' A))
                    (let val (nlist , _) = der S c A
                     in nlist
                     end))
          of _ => true)
        handle notfound => false
```

In the case of `nDiamond`, `nSatInner` refers to the second part of the function, `pSatInner`. This is completely analog to the formal definition, e.g. $\langle a \rangle \varphi$.

The only difference in `pSatinner` is the probabilistic modality, or rather the two modalities, the rest is therefore omitted in the following. The case of `pDiamondsimp` builds a `Sum` variable, which for all the transitions that satisfy `nForm` collects the probabilities and sum them up using the function `sumprob`. This sum of probabilities is then checked against the μ, to evaluate to true or false.

```
and pSatInner S pForm c A =

                 ⋮

      | (pDiamondsimp (mu,nForm)) =>
        let val Sum =
                (sumprob (filter (fn(prob,c')=>
                                    (nSatInner S nForm c' A))
                                 (let val (_,plist) = der S c A
                                  in plist
                                  end)))
        in (Sum > mu) orelse (Sum = mu)
        end
```

The general modality works much in the same way as the simple one, only it operates on a list of `nForm`'s, and has corresponding alpha values.

```
  | (pDiamond ((nil),mu)) => false
  | (pDiamond (((alpha,nForm)::T),mu)) =>
        let fun multsum nil = 0.0
              | multsum ((alpha,nForm)::T) =
            ((sumprob (filter (fn(prob,c')=>
                                (nSatInner S nForm c' A))
                         (let val (_,plist) = der S c A
                          in plist
                          end)))*alpha) + multsum T
        in ((multsum ((alpha,nForm)::T))>mu) orelse
            ((multsum ((alpha,nForm)::T))=mu)
        end
```

To define the final function `Satisfy`, we need a function that finds the initial configuration of the parallel system. The job is done by the function `initialconf`, which takes a transition system and returns the initial configuration.

```
fun initialconf (system (ntrans, ptrans, s0, i)) = [(s0,i)]
  | initialconf (S1 || S2) = (initialconf S1)@(initialconf S2)
```

This is in fact the only difference between `Satisfy` and `nSatInner`, that the initial configuration is found automatically. We here require formulas to begin with type `nFormula`.

```
fun Satisfy S Form A = nSatInner S Form (initialconf S) A
```

Chapter 5

The Quotient Technique for PTS

In this chapter the quotient technique is defined and proved correct for the asynchronous version of PTS. We give a structural definition of the quotient technique and show that our logic is not strong enough to support the technique. We then introduce a general modality, which completes our logic, and give the two types of semantics for it. We prove the quotient theorem by structural induction, and end this chapter with an example of verification of a simple parallel transition system by using the quotient technique.

5.1 Definition of the Quotient Technique

The quotient technique for probabilistic transition systems works the same way as for finite state systems, described in the introduction and in appendix A. We recall that the purpose of the quotient technique is to try to avoid the state explosion problem in parallel systems, by factoring out machines one at a time and placing their properties in the formula for the whole system. By doing this, and by repeatedly applying simplification techniques, we should be able to avoid the state explosion problem, and thereby reduce the verification time of the system.

The quotient operator / is defined in Definitions 5.1 and 5.2.

Definition 5.1 (Structural definition of F/P)

$$\begin{array}{rrcl}
(i) & tt/P_2 & = & tt \\
(ii) & (F_1 \wedge F_2)/P_2 & = & (F_1/P_2) \wedge (F_2/P_2) \\
(iii) & \neg F/P_2 & = & \neg(F/P_2) \\
(iv) & \langle a \rangle \varphi/P & = & \begin{cases} \langle a \rangle \left(\bigvee_{P \xrightarrow{a} \pi} \varphi/\pi \right) & a \in \mathcal{A} \\ \langle a \rangle [(\varphi/P) \vee \bigvee_{P \xrightarrow{a} \pi} \varphi/\pi] & a \notin \mathcal{A} \end{cases}
\end{array}$$

Definition 5.2 (Structural definition of φ/π)

$$\begin{array}{rrcl}
(v) & tt/\pi_2 & = & tt \\
(vi) & (\varphi_1 \wedge \varphi_2)/\pi_2 & = & \varphi_1/\pi_2 \wedge \varphi_2/\pi_2 \\
(vii) & \neg\varphi/\pi_2 & = & \neg(\varphi/\pi_2) \\
(viii) & \diamond_{\geq\mu} F/\pi_2 & = & [\diamond_{x_1}(F/P_1), \diamond_{x_2}(F/P_2), \ldots, \diamond_{x_k}(F/P_k) : \\
 & & & \quad \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_k x_k \geq \mu]
\end{array}$$

where $\pi_2 \rightsquigarrow_{\alpha_1} P_1, \ldots, \pi_2 \rightsquigarrow_{\alpha_k} P_k$ enumerates all probabilistic transitions of π_2.

As we see, the definition of the the quotient formula for $\diamond_{\geq\mu} F$ is not included in our logic. We therefore need to extend our logic, as explained next.

5.2 Generalization of the Diamond Modality

The logic we have given is not strong enough for describing certain properties, so we have to extend this logic with a more general construct.

Apart from the definition of the simple diamond modality above, it may not be obvious why we need the generalized probabilistic diamond modality, and how it works. Therefore we will give an example to illustrate that the simple modality $\diamond_{\geq\mu} F$ is not expressive enough, and with the need for this modality.

We start out by assuming that we only have the simple modality in our logic. Figure 5.1 show a system with a distribution π_2. Assume that we want to find a distribution π_1, which when in parallel with π_2, satisfies the following property: $\varphi = \diamond_{\geq\frac{1}{4}}(\langle b \rangle tt \wedge \langle c \rangle tt$. That is we want π_1 to be such that:

$$\pi_1 |_{\mathcal{A}} \pi_2 \models \varphi.$$

Now, given the existence of a quotient construction in our probabilistic setting this should be equivalent to:

$$\pi_1 \models \varphi/\pi_2.$$

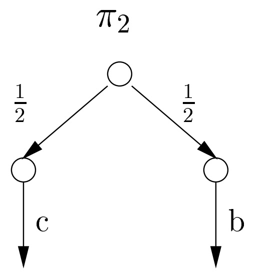

Figure 5.1: A small system

where φ/π_2 is the quotient formula for φ with respect to π_2. Consider π_1's transitions (see Figure 5.2), we denote the unknown sum of the probabilities of the transitions leading to a state where c and b is possible by x_1 and x_2, respectively. In order for $\pi_1|_{\mathcal{A}}\pi_2$ to satisfy φ it is clear that the requirement

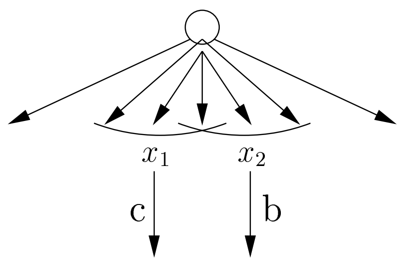

Figure 5.2: The transitions of π_1

to π is that $\frac{1}{2}x_1 + \frac{1}{2}x_2 \geq \frac{1}{4}$. We can express this in our semantic terms the following way

$$\frac{1}{2}\pi_1[\![\langle c\rangle tt \wedge \langle b\rangle tt/P_1]\!] + \frac{1}{2}\pi_1[\![\langle c\rangle tt \wedge \langle b\rangle tt/P_2]\!] \geq \frac{1}{4}$$

However this is not expressible in our logic as a single formula. We therefore extend our logic with a more general modality, that allows us to express this.

Definition 5.3 (Extension of PML)
We define the following to be part of the syntax for PML

$$\varphi \models [\diamond_{x_1}F_1, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu].$$

The semantics for this modality is defined as:

$$\pi \models [\diamond_{x_1}F_1, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] \textit{iff } \alpha_1\xi_1 + \cdots + \alpha_n\xi_n \geq \mu$$

where $\xi_i = \sum_{P.P\models F} \pi(P)$.

To see that this is a generalization of the simple modality, we note that:

$$\pi \models \diamond_{\geq\mu} F \Leftrightarrow \pi \models [\diamond_x F : x \geq \mu].$$

The equivalent denotational semantic for the general modality is:

$$\{\!|[\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]|\!\} = \{\pi \mid \alpha_1 \pi [\![F_1]\!] + \cdots + \alpha_n \pi [\![F_n]\!] \geq \mu\}.$$

We can now give the definition of the quotient technique for the general modality:

Definition 5.4 (Extended definition)
We extend definition 5.2 to include the following definition of the general modality.

$$(ix) \qquad [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]/\pi_2 =$$

$$[\diamond_{y_{11}} F_1/Q_1, \ldots, \diamond_{y_{nk}} F_n/Q_k : \sum_{j=1}^{k} \nu_1 \alpha_1 y_{1j} + \cdots + \sum_{j=1}^{k} \nu_n \alpha_n y_{nj} \geq \mu]$$

where $\pi_2 \leadsto_{\nu_n} Q_n$ enumerates all the probabilistic transitions of π_2.

We will later show that this extension the logic is strong enough to express the properties that arise from the quotient procedure, i.e. applying the quotient technique on a formula of the form $[\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]$, results in a formula of the same form.

5.3 Proving Correctness of the Quotient Technique

We now state the quotient theorem, and give the proof of it:

Theorem 5.5
Given two processes P_1 and P_2 and two distributions π_1 and π_2 in PTS, then

$$P_1|_{\mathcal{A}} P_2 \models F \textit{ iff } P_1 \models F/P_2$$

$$\pi_1|_{\mathcal{A}} \pi_2 \models \varphi \textit{ iff } \pi_1 \models \varphi/\pi_2.$$

Proof
We prove theorem 5.5 by induction on the structure F respectively π.

(i) $F = tt$ is trivial

(ii) $F = F_1 \wedge F_2$:

$$\begin{array}{rl} & P_1|_{\mathcal{A}}P_2 \models F_1 \wedge F_2 \\ \Leftrightarrow & P_1|_{\mathcal{A}}P_2 \models F_1 \text{ and } P_1|_{\mathcal{A}}P_2 \models F_2 \\ \overset{\text{IH}}{\Leftrightarrow} & P_1 \models F_1/P_2 \text{ and } P_1 \models F_2/P_2 \\ \Leftrightarrow & P_1 \models F_1/P_2 \wedge F_2/P_2 \\ \overset{\text{Def}}{\Leftrightarrow} & P_1 \models (F_1 \wedge F_2)/P_2 \end{array}$$

(iii) Negation follows like $\wedge$, directly from the induction hypothesis.

(iv) $F = \langle a \rangle \varphi$:
This case is divided in two parts, the synchronous and the asynchronous. First the synchronous case:

$$\begin{array}{rl} & P_1|_{\mathcal{A}}P_2 \models \langle a \rangle \varphi \\ \Leftrightarrow & \exists j,k.P_1|_{\mathcal{A}}P_2 \xrightarrow{a} \pi_j \| \pi_k \wedge \pi_j \| \pi_k \models \varphi \\ \overset{\text{IH}}{\Leftrightarrow} & \exists j,k.P_1 \xrightarrow{a} \pi_j \wedge P_2 \xrightarrow{a} \pi_k \wedge \pi_j \models \varphi/\pi_k \\ \Leftrightarrow & \exists k.(P_1 \models \langle a \rangle (\varphi/\pi_k)) \wedge P_2 \xrightarrow{a} \pi_k \\ \Leftrightarrow & P_1 \models \bigvee_{k.P_2 \xrightarrow{a} \pi_k} \langle a \rangle \varphi/\pi_k \\ \Leftrightarrow & P_1 \models \langle a \rangle \bigvee_{P_2 \xrightarrow{a} \pi_2} \varphi/\pi_2 \\ \overset{\text{Def}}{\Leftrightarrow} & P_1 \models (\langle a \rangle \varphi)/P_2 \end{array}$$

Then the asynchronous case:

$$\begin{array}{rl} & P_1|_{\mathcal{A}}P_2 \models \langle a \rangle \varphi \\ \Leftrightarrow & \exists \pi_1.(P_1 \xrightarrow{a} \pi_1 \wedge \pi_1|_{\mathcal{A}}P_2 \models \varphi) \text{ or } \exists \pi_2.(P_2 \xrightarrow{a} \pi_2 \wedge P_1|_{\mathcal{A}}\pi_2 \models \varphi) \\ \Leftrightarrow & \exists \pi_1.P_1 \xrightarrow{a} \pi_1 \wedge \pi_1 \models \varphi/P_2 \text{ or } \exists \pi_2.P_2 \xrightarrow{a} \pi_2 \wedge P_1 \models \varphi/\pi_2 \\ \Leftrightarrow & P_1 \models \langle a \rangle \varphi/P_2 \text{ or } \bigvee_{P_2 \xrightarrow{a} \pi_2} \wedge \varphi/\pi_2 \\ \Leftrightarrow & P_1 \models \langle a \rangle \varphi/P_2 \vee \bigvee_{P_2 \xrightarrow{a} \pi_2} \varphi/\pi_2 \\ \overset{\text{Def}}{\Leftrightarrow} & P_1 \models \langle a \rangle \varphi/P_2 \end{array}$$

(v) $\varphi = tt$ is trivial

(vi) $\varphi = \varphi_1 \wedge \varphi_2$. The proof of this is similar to $F_1 \wedge F_2$.

(vii) Negation follows directly from the induction hypothesis, like the previous case.

(viii) $\varphi = \diamond_{\geq\mu} F$:

$$\begin{array}{cl}
 & \pi_1|\pi_2 \models \diamond_{\geq\mu} F \\
\Leftrightarrow & \displaystyle\sum_{Q|_{\mathcal{A}}P \models F} \pi_1(Q)\cdot\pi_2(P) \geq \mu \\
\overset{\text{IH}}{\Leftrightarrow} & \displaystyle\sum_{P}\sum_{Q\models F/P} \pi_1(Q)\cdot\pi_2(P) \geq \mu \\
\Leftrightarrow & \pi_1 \models [\diamond_{x_1}(F/P_1),\ldots,\diamond_{x_n}(F/P_n) : \alpha_1\cdot x_1 + \cdots + \alpha_n\cdot x_n \geq \mu] \\
 & \text{where } \pi_2 \rightsquigarrow_{\alpha_1} P_1,\ldots,\pi_2 \rightsquigarrow_{\alpha_n} P_n \\
\overset{\text{Def}}{\Leftrightarrow} & \pi_1 \models (\diamond_{\geq\mu} F)/\pi_2
\end{array}$$

(ix) $\varphi = [\diamond_{x_1} F_1,\ldots,\diamond_{x_n} F_n : \Phi(x_1,\ldots,x_n)]$:

$$\begin{array}{cl}
 & \pi_1|_{\mathcal{A}}\pi_2 \models [\diamond_{x_1} F_1,\ldots,\diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] \\
\Leftrightarrow & \alpha_1\xi_1 + \cdots + \alpha_n\xi_n \geq \mu,\ \text{where } \xi_i = \sum_{P,Q_j.P|Q_j\models F_i} \pi_1(P)\cdot\pi_2(Q_j) \\
\Leftrightarrow & \xi_i = \sum_{Q_j}\left[\underbrace{\pi_2(Q_j)}_{\alpha_j}\cdot\overbrace{\sum_{P.P\models F_i/Q_j}\pi_1(P)}^{y_{ij}}\right] \\
\Leftrightarrow & [\diamond_{y_{11}} F_1/Q_1,\ldots,\diamond_{y_{nk}} F_n/Q_k : \sum_{j=1}^{k}\nu_1\alpha_1 y_{1j} + \cdots + \sum_{j=1}^{k}\nu_n\alpha_n y_{nj} \geq \mu] \\
 & \text{where } \pi_2 \rightsquigarrow_{\nu_i} Q_i \text{ enumerates all probabilistic transitions of } \pi_2 \\
\overset{\text{Def}}{\Leftrightarrow} & [\diamond_{x_1} F_1,\ldots,\diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]/\pi_2
\end{array}$$

□

5.4 The Use of the Quotient Technique

In this section we discuss the practical use of the quotient technique. We show that when factoring out machines, the order does not matter, and we will examine how many machines we have to factor out before reaching any conclusions.

Theorem 5.6
When factoring out machines by using the quotient technique, the order does not matter.

That is

$$[\![(F/P_1)/P_2]\!] = [\![(F/P_2)/P_1]\!]$$

Proof
We know that by theorem 5.5:

$$Q \models (F/P_1)/P_2 \Leftrightarrow Q \mid P_2 \models F/P_1 \Leftrightarrow (Q \mid P_2) \mid P_1 \models F$$

We know from theorem 4.8 that the order of parallel composition doesn't matter, so we can write this as

$(Q|P_2)|P_1 \models F \Leftrightarrow Q \mid (P_2 \mid P_1) \models F \Leftrightarrow Q \mid (P_1 \mid P_2) \models F \Leftrightarrow (Q \mid P_1) \mid P_2 \models F$

Again we use theorem 5.5 to get

$$(Q|P_1)|P_2 \models F \Leftrightarrow Q|P_1 \models F/P_2 \Leftrightarrow Q \models (F/P_2)/P_1$$

which concludes the proof. □

So for the quotient technique to work, it is not necessary to factor out components in any specific order.

The question is then, will the time complexity be the same, when factoring out components in different orders. We believe the answer to this question is no. We might get smaller formulas by factoring out one component compared to another, and some formulas may be easier to simplify than others.

In fact we can see that theorem 5.6 allows us to write $F/\mathcal{P}$, where $\mathcal{P}$ is a set of processes. In particular if $F/P \equiv \mathit{ff}$ for $P \in \mathcal{P}$, then $F/\mathcal{P} \equiv \mathit{ff}$. This implies that if we have to check $P_1 \mid \cdots \mid P_n \models F_1 \vee \cdots \vee F_m$ and $F_k/P_j \equiv \mathit{ff}$ for some k, j, then

$$P_1|\cdots|P_j|\cdots|P_n \models F_1 \vee \cdots \vee F_k \vee \cdots \vee F_m$$

iff

$$P_1|\cdots|P_j|\cdots|P_n \models F_1 \vee \cdots \vee F_{k-1} \vee F_{k+1} \vee \cdots \vee F_m.$$

So F_k can be removed. We will discuss this further in chapter 7.

Another question when using the quotient technique is, when to terminate the process of factoring out components. The answer to this question depends greatly on the formulas involved, but with n components, we cannot factor out more than $n-1$. This is because that $\langle\langle P|NIL\rangle\rangle = \langle\langle P\rangle\rangle$ does *not* hold in general, because:

if $a \in \mathcal{A}$ then $a.\pi_{NIL} \xrightarrow{a} \pi_{NIL}$, but $a.\pi_{NIL}|NIL \not\xrightarrow{a}$. Thus $a.\pi_{NIL} \models \langle a\rangle tt$, but $a.\pi_{NIL}|NIL \not\models \langle a\rangle tt$.

So because we have synchronous transitions, we cannot remove the last machine from a system, and thus we have to manually check the last machine with the specification.

5.5 Example of the Quotient Technique

We will now give a simple example of the use of the quotient technique. We consider a faulty medium which can send, reject and accept messages. It accepts and rejects the send messages by a certain rate defined by a probability. The system M can be seen in figure 5.3.

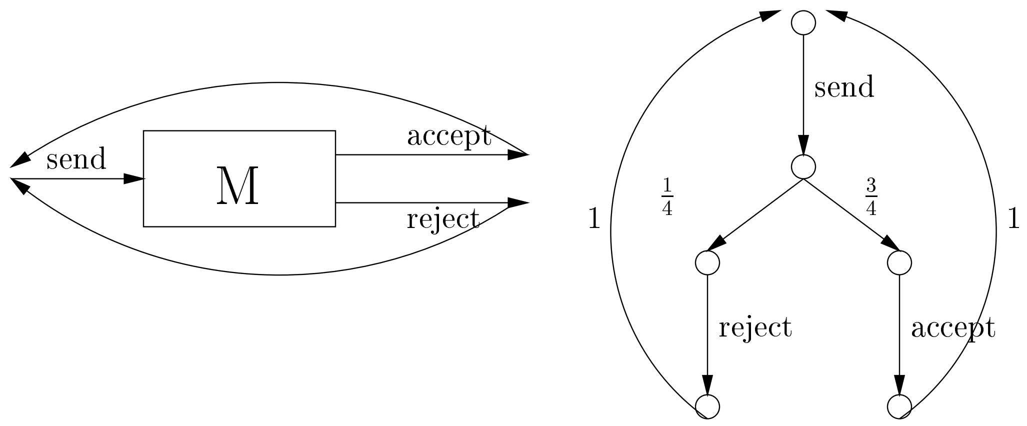

Figure 5.3: The media M

As we can see in the figure the media accepts send messages with a probability $\frac{3}{4}$, and rejects with $\frac{1}{4}$. When a message has been either rejected or accepted, it returns to the initial state, ready to send a new message.

Now suppose we want to increase the rate of accepted messages. This can be done by putting one or more new medias in parallel with the original one, and letting the composed system be asynchronous on the accept transition. We can now do two things, either using more of the same type of media (with loss rate of $\frac{1}{4}$), and check if the resulting composed system satisfies our demands, or we can try finding a specification for a new media to use with the original one.

We can use the quotient technique to do both, in the last case, we can simply take our specification and factor out the original media, to obtain a specification for the new media. In the first case we simply put as many components in parallel with the original one as we think is enough, and then verify the composed system by factoring out medias on at a time.

We start by putting another media N of the same type in parallel with M, to obtain $M|_{\mathcal{A}}N$. The system synchronizes on all other transitions but *accept*, that is $\mathcal{A} = \{send, reject\}$.

We want the final system to be able to accept messages by a rate of 90%, that is

$$M|_{\mathcal{A}}N \models \langle send \rangle \diamond_{\geq \frac{9}{10}} \langle accept \rangle tt$$

We will now use the quotient technique to verify this formula. We factor out

M to obtain a specification for N, which we then check manually.

$$\begin{aligned} N &\models (\langle send \rangle \diamond_{\geq \frac{9}{10}} \langle accept \rangle tt)/N \\ &= \langle send \rangle (\diamond_{\geq \frac{9}{10}} \langle accept \rangle tt)/N \\ &= \langle send \rangle [\diamond_{x_1} \langle accept \rangle tt/s_1, \diamond_{x_2} \langle accept \rangle tt/s_2 : \tfrac{1}{4}x_1 + \tfrac{3}{4}x_2 \geq \tfrac{9}{10}] \end{aligned}$$

When M is in state s_1, it can't perform any *accept* transitions, so the rest of our system has to do that in order to satisfy the specification. In state s_2, M has an accept transition, so the rest of the system only has to be able to reach this state to be satisfied.

$$N \models \langle send \rangle \left[\diamond_{x_1} \langle accept \rangle tt, \diamond_{x_2} tt : \frac{1}{4}x_1 + \frac{3}{4}x_2 \geq \frac{9}{10} \right]$$

As we shall see in chapter 7, we can set x_2's value to 1, so we can solve our inequality:

$$\frac{1}{4} + \frac{3}{4} \cdot 1 \geq \frac{9}{10} \Leftrightarrow \frac{1}{4}x_1 \geq \frac{9}{10} - \frac{3}{4} \Leftrightarrow x_1 = \frac{3}{5}$$

We use this value in our specification and get

$$N \models \langle send \rangle \diamond_{\geq \frac{3}{5}} \langle accept \rangle tt$$

So we now have a specification for N, and can manually verify that if N is of same type as M, then it clearly satisfies the specification. We can also choose to find a media with the specification of N, which may be cheaper (because of the lower accept rate), and use that in parallel with M.

We shall later see that formulas do not always reduce so easily, and when applying the quotient technique to a general modality, we actually get quite large formulas.

5.6 Implementation

We will now describe the implementation of the quotient technique. The technique is implemented as a single function `evalQuotient`, which take the following as argument:

- The process (of type `System`) that is to be factored out
- A formula
- The synchronizing set `A`

This function declares `c` to be the initial configuration of `S`, and calls a mutually recursive sub-function `nonquotient` and `probquotient`. This sub-function is called recursively on every instance of the formula, and simply returns the new quotient formula.

We will start by defining the more simple formula types, like true, false, AND and OR. Below is the code for the process version of these operators.

```
fun evalQuotient S formula A =
    let val c = initialconf S
    in let fun nonquotient (nTrue) _ = nTrue
             | nonquotient (nFalse) _ = nFalse
             | nonquotient (nAp (s,i)) c =
               let val acstate = getactivestate c i
               in if acstate = s
                  then nTrue
                  else nFalse
               end
             | nonquotient (nNot nform) c =
               nNot (nonquotient nform c)
             | nonquotient (L nAnd R) c =
               let val LQ = nonquotient L c
                   val RQ = nonquotient R c
               in LQ nAnd RQ
               end
             | nonquotient (L nOr R) c =
               let val LQ = nonquotient L c
                   val RQ = nonquotient R c
               in LQ nOr RQ
               end
```

The probabilistic counterparts for these simple types are similar to those of the process version, and are therefore omitted here.

The function for process modality $\langle a \rangle \varphi$ we start by calling the function `probquotient` to create the disjunction $\bigvee_{P \xrightarrow{a} \pi} \varphi/\pi$. It then checks whether the action a is in the synchronizing set or not, and returns the corresponding formula.

```
| nonquotient (nDiamond (a,pForm)) c =
  let val Disjunction =
      reduce (fn ((act,c'),tailDisjunction) =>
              ((probquotient pForm c')
               pOr tailDisjunction))
      pFalse
      (filter (fn (act,c') => (act=a))
       (let val (nlist,_) = der S c A
        in nlist
        end))
  in if memberof a A
        then nDiamond (a,Disjunction)
     else let val notA =
          nDiamond (a,(probquotient pForm c)
                    pOr Disjunction)
          in if (nSatInner S notA c A)
                then nTrue
             else notA
          end
  end
```

This quotient step also checks to see if the formula can be simplified, it checks if the system satisfies `notA`, and if it does returns true.

Now to the probabilistic case, for which we, as mentioned above, only show the diamond modalities.

The simple diamond modality i straight forward, it simply forms a formula of the form `pDiamond`, and calls `nonquotient` on the corresponding process formula.

```
| probquotient (pDiamondsimp (p,nform)) c =
  pDiamond ((map (fn (mu,conf) =>
                  (mu, nonquotient nform conf))
                (let val (_,plist) = der S c A
                 in plist
                 end)), p)
```

The quotient formula for the general diamond modality is implemented in two steps, one where there is no list of probabilities and process formulas (`nil`) and one where there is a list.

The function calls itself recursively to apply the quotient technique to all instances of `prob, nform`.

```
          | probquotient (pDiamond (((prob,nform)::L),p)) c =
             pDiamond (((map (fn (mu,conf) =>
                                 ((mu*prob), nonquotient nform conf))
                         (let val (_,plist) = der S c A
                          in plist
                          end))@
                        (let val pDiamond (T,_) =
                                 probquotient (pDiamond (L,p)) c
                         in T
                         end)),p)
          | probquotient (pDiamond (nil,p)) c = pDiamond (nil,p)
      in nonquotient formula c
      end
  end
```

We will later describe the function that is used to call this quotient function with, but it basicly works by choosing the machine to factor out, and calling the quotient function on that machine and a formula, and then calling the simplification function (see chapter 7) on the quotiented formula.

Chapter 6

The General Modality

In this chapter we examine the general modality, especially the linear inequality $\alpha_1 x_1 + \cdots \alpha_n x_n \geq \mu$, in order to describe it in a way that allows us to implement the technique. In [LS92] Larsen and Skou define a general modality to support their version of the quotient technique (decomposition). They define the modality in more general terms than we do, by having some function $\Phi(x_1, \ldots, x_n)$ instead of the inequality. This notion was also adopted in [Ves00].

We believe that the general construct of Larsen and Skou is unnecessary and confusing to read, and we devote this chapter to examine the general modality. We show that no matter how many times we apply the quotient technique to the modality, we end up with only one type of inequality, though with several unknown variables.

$$[\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]/\pi = \tag{6.1}$$

$$[\diamond_{y_{11}} F_1/Q_1, \ldots, \diamond_{y_{ij}} F_i/Q_j, \ldots, \diamond_{y_{nk}} F_n/Q_k : \sum_{j=1}^{k} \nu_1 \alpha_j y_{1j} + \cdots + \sum_{j=1}^{k} \nu_n \alpha_j y_{nj} \geq \mu].$$

It has taken a lot of considerations to realize that the last part of the general modality is always a linear inequality. The technique developed in this report is based on the ideas of [LS92], and our work prior to this report [Ves00] defines the general modality with a general Φ-function.

Our studies have shown that this function is in fact a linear inequality. Actually, as we shall see, it just operates with new binding variables, y_{ij}, instead of x_1.

6.1 Describing the General Modality

First we need to realize that, when using the quotient technique on a general modality, we simply get another general modality, but with a few more variables and probability constants. That is, the general modality is always of the form:

$$[\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]$$

where x_i is a variable binding the probabilities. It is this inequality that is interesting, and we therefore need to describe exactly how it looks. Furthermore we know that our binding variables, because they are in fact probabilities, have the following property: $0 \leq x_i \leq 1$.

Now, if we look at the definition of the quotient technique for the general modality (see formula 6.1), we see that it is very similar to the one for the simple modality. Actually we should be able to write the quotiented modality as follows:

$$[\diamond_{y_{11}} F_1/Q_1, \ldots, \diamond_{y_{nk}} F_n/Q_k : (\sum_{j=1}^{k} \nu_j y_{1j})\alpha_1 + \cdots + (\sum_{j=1}^{k} \nu_j y_{nj})\alpha_n \geq \mu]$$

The inequality consists of a k and a n vector and a $n \times k$ matrix:

$$\left(\begin{bmatrix} y_{11} & \cdots & y_{1k} \\ \vdots & & \vdots \\ y_{n1} & \cdots & y_{nk} \end{bmatrix} \cdot \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_k \end{bmatrix} \right)^T \cdot \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} \geq \mu$$

So we will have $n \cdot k$ possibly unknown variables, and each time the quotient technique is applied, this number will be multiplicated by n, thus it will grow exponentially. This surely underlines the need for simplification rules and hopefully it will be possible to keep the number of unknown variables to a minimum.

We also need a means of dealing with linear inequalities with multiple unknown variables, this will be examined next.

6.2 Linear Inequalities

In order to reason about the linear inequalities with more than one unknown variables, we need to explore some theory about this subject. We will start by looking at how to define a range of solutions for inequalities of the form $\alpha_1 x_1 + \cdots + \alpha_n x_n \geq \beta$.

Because we work with probabilities, we know a few things about our variables and constants. First of all we know that all constants α_i are non-negative, and will sum up to one, because of the fact that all α_i is from transitions from a single state (e.g. $\pi_1 \rightsquigarrow_{\alpha_i} P_i$):

$$\alpha_1 + \cdots + \alpha_n = 1 \text{ and } 0 \leq \alpha_i.$$

We also know that the variables x_i are non-negative and less than or equal to 1:

$$0 \leq x_i \leq 1, \text{ where } i = 1, \ldots, n$$

Despite this information it is not possible to give a definition on how to find the values of the unknown variables. Instead we can define some simplification rules, which can eventually reduce the inequality, and thereby the general modality.

As we shall see in the next chapter, we will be able to simplify the inequality a bit, removing some of the variables, and there by subtracting or removing some of the α_i's. If we assume that the constant being removed is α_i we get:

$$\alpha_1 + \cdots + \alpha_{i-1} + \alpha_{i+1} + \cdots + \alpha_n \leq 1$$

This means, that if the largest of the constants multiplied with n are smaller than β then we can conclude that the inequality is not solvable, that is

$$\alpha_1 x_1 + \cdots + \alpha_n x_n \geq \beta \Leftrightarrow \mathit{ff} \tag{6.2}$$

if

$$Max(\alpha_i) < \beta/n$$

Note that this rule is only effective when at least one of the α_i's have been removed. We can even give a stronger rule: If $\alpha_1 + \alpha_2 + \cdots + \alpha_n < \beta$, then 6.2 holds.

So we can reduce the inequality to false, if any of the α_i's are smaller than μ/n. As we will see in the next chapter about simplification, this is in fact enough to declare the whole modality false, and thereby simplify the quotient formula a great deal.

Considering repeatedly application of the quotient technique, we see that the number of variables rises exponentially, but the corresponding constants will get smaller and smaller, hopefully causing the inequality and thereby the whole modality to reduce to false. We have to do some tests to see if the constants values lowers faster than n rises, as we can't say anything general about that. It all depends on μ, on the predefined probabilities, and on the size of the transition systems.

Another way to simplify the general modality is when we reach a situation where μ becomes negative. This will happen when we can reduce any of the x_i's to 1, and subtract the corresponding α_i from μ. If μ becomes negative, then the inequality is trivially true, because of all x_i's and α_i's are non-negative.

We will now give an example of the general modality, and some of the simplifications that can be applied when using the quotient technique.

6.3 Example of General Modality

Again we consider the example in 4.2.3, but now with a different formula, and one more machine in parallel which we call C (identical to machine B).

$$\langle a \rangle \diamond_{\geq \frac{1}{3}} (\langle b \rangle tt \vee \langle c \rangle tt)$$

Remember that the synchronization set is $\mathcal{A} = \{a, b, d\}$, so the system is asynchronous on the $c - transition$. We start by factoring out C.

$$\begin{array}{lll} & A|_{\mathcal{A}}B|_{\mathcal{A}}C & \models \langle a \rangle \diamond_{\geq \frac{1}{3}} (\langle b \rangle tt \vee \langle c \rangle tt) \\ \Leftrightarrow & A & \models \langle a \rangle \diamond_{\geq \frac{1}{3}} (\langle b \rangle tt \vee \langle c \rangle tt)/C \\ \Leftrightarrow & A & \models \langle a \rangle [\diamond_{x_1}(\langle b \rangle tt \vee \langle c \rangle tt)/P1, \diamond_{x_2}(\langle b \rangle tt \vee \langle c \rangle tt)/P2 : \frac{1}{3}x_1 + \frac{2}{3}x_2 \geq \frac{1}{3}] \\ \Leftrightarrow & A & \models \langle a \rangle [\diamond_{x_1}(\langle b \rangle tt \vee \langle c \rangle tt), \diamond_{x_2}\langle c \rangle tt : \frac{1}{3}x_1 + \frac{2}{3}x_2 \geq \frac{1}{3}] \end{array}$$

It is not possible to simplify this formula anymore, so we will factor out B.

$$\begin{array}{lll} & A|_{\mathcal{A}}B & \models \langle a \rangle [\diamond_{x_1}(\langle b \rangle tt \vee \langle c \rangle tt), \diamond_{x_2}\langle c \rangle tt : \frac{1}{3}x_1 + \frac{2}{3}x_2 \geq \frac{1}{3}] \\ \Leftrightarrow & A & \models \left(\langle a \rangle [\diamond_{x_1}(\langle b \rangle tt \vee \langle c \rangle tt), \diamond_{x_2}\langle c \rangle tt : \frac{1}{3}x_1 + \frac{2}{3}x_2 \geq \frac{1}{3}]\right)/B \\ \Leftrightarrow & A & \models \langle a \rangle [\diamond_{y_{11}}(\langle b \rangle tt \vee \langle c \rangle tt)/P_1, \diamond_{y_{12}}(\langle b \rangle tt \vee \langle a \rangle tt)/P_2, \\ & & \quad \diamond_{y_{21}}(\langle c \rangle tt)/P_1, \diamond_{y_{22}}(\langle c \rangle tt)/P_2 : \frac{1}{3} \cdot \frac{1}{2}y_{11} + \frac{1}{3} \cdot \frac{1}{2}y_{12} + \frac{2}{3} \cdot \frac{1}{2}y_{21} + \frac{2}{3} \cdot \frac{1}{2}y_{22} \geq \mu] \end{array}$$

When we calculate the last quotients in this formula, and simplify the formula, we get

$$\langle a \rangle \left[\diamond_{y_{11}}\langle b \rangle tt \vee \langle c \rangle tt, \diamond_{y_{12}}\langle c \rangle tt, \diamond_{y_{21}}\langle c \rangle tt, \diamond_{y_{22}}\langle c \rangle tt : \frac{1}{6}y_{11} + \frac{1}{6}y_{12} + \frac{1}{3}y_{21} + \frac{1}{3}y_{22} \geq \mu\right]$$

Again with no chance of simplifying the modality anymore. As we can see the formula is growing quite large, and it will grow even more if more machines of the same type are factored out.

The formula is still satisfied though, which should be easy to see. The question is now, what would have happened if we factored out machine A first?

We will not show all calculations here, but the simplified result of $\langle a \rangle \diamond_{\geq \frac{1}{3}} (\langle b \rangle tt \vee \langle c \rangle tt)/A$ is $\langle a \rangle tt$, which definitely is a smaller formula and easier to verify than the above.

This shows us that the order of which we factor out components does not affect the final result, but formulas may simplify more easily when choosing one machine to factor out, instead of another.

Chapter 7

Simplifying rules

When using the quotient technique, one of the most important things are the application of simplification heuristics. Formulas tend to become quite large when factoring out components, and especially the general modality has an exponential blowup in the number of variables when applying the quotient technique. This can not completely be avoided, but with simplification techniques, it may be possible to keep the formulas small and the number of variables low.

In chapter 6 we discussed simplification to the general modality, and the inequalities in particular. In this chapter we formally define the notion of simplification, and several simplification rules. We prove soundness of all the rules, and finally describe the implementation of them.

7.1 Introduction to Simplification

Simplifying rules is a set of semantics preserving rules which can be used to minimize a formula F or φ. The idea is to apply these rules continuously while quotienting, so that the final expression is small and easy to verify:

$$P_1|_{\mathcal{A}}P_2 \models F \Leftrightarrow P_1 \models (F/P_2)^S$$

In the example in section 5.5, we already used a few simplification rules. We will here give the definition of a set of simplification rules for our systems, and show that they are sound.

Formally we write $F \mapsto F'$ and $\varphi \mapsto \varphi'$, where F' and φ' is smaller (simplified) than F and φ, but still equivalent in the following sense:

$$\forall P : P \models F \Leftrightarrow P \models F' \text{ and } \forall \pi : \pi \models \varphi \Leftrightarrow \pi \models \varphi'$$

or equivalently

$$[\![F]\!] = [\![F']\!] \text{ and } \{\!|\varphi|\!\} = \{\!|\varphi'|\!\}$$

We define the following simple derived operations:

$$\begin{array}{llllll} \mathit{ff} & \overset{Def}{\equiv} & \neg tt & F \wedge G & \overset{Def}{\equiv} & \neg(\neg F \vee \neg G) \\ F \Rightarrow G & \overset{Def}{\equiv} & \neg F \vee G & F \Leftrightarrow G & \overset{Def}{\equiv} & (F \Rightarrow G) \wedge (G \Rightarrow F) \end{array}$$

The definition of simplification rules are split into two sections, one for processes properties, and one for probabilistic. There has already been put a lot of work into simplifying process formulas (e.g. [Kri98], [And95] and [rHA97]), so we will concentrate on simplification of the probabilistic properties.

7.2 Order of Machines Factored Out

As discussed earlier, and as shown in the previous chapter, the order of which machines are factored out, may have something to say when applying simplification rules.

We believe the reason that the system in example 6.3 can be simplified greatly by factoring out A instead of B, is that A included the asynchronous transition c. When a machine has an asynchronous transition, the demands for the rest of the system, when that machine is factored out, are loosened.

This intuition also follows the definition of the quotient operator for asynchronous composition (see definition 5.1).

We can conclude that when we have an asynchronous action in our specification, then we could check to see weather we have any machines in the parallel system, which can take this transition, and factor that out first. As we shall see later, this is not implemented in our model checker, as the model checker is doing fine with the rules described in the next two sections.

7.3 Simplification Rules for F

Besides the logically implied simplification rules like $tt \wedge \mathit{ff} \mapsto \mathit{ff}$ and $tt \vee \mathit{ff} \mapsto tt$, we need some rules to simplify expressions with general properties.

Definition 7.1 (Simplification rules for F)

$$\begin{array}{lll}
1_F: & \langle a\rangle f\!f & \mapsto f\!f \\
2_F: & tt \wedge F & \mapsto F \\
3_F: & f\!f \wedge F & \mapsto f\!f \\
4_F: & \langle a\rangle \varphi / P_i & \mapsto f\!f \\
& \text{iff } P_1|_{\mathcal{A}} \cdots |_{\mathcal{A}} P_n \models F \text{ and } a \in \mathcal{A} & \text{and } P_i \overset{a}{\not\rightarrow} \text{ for some } i = 1\ ,\ldots, n
\end{array}$$

Theorem 7.2
The simplification rules in definition 7.1 are sound.

Proof
Rules 1_F through 3_F are trivial.

The proof of 4_F follows from the definition of the quotient technique, and from theorem 5.6. □

7.4 Simplification rules for φ

Simplification rules for φ are defined in 7.3. They are based on the extended modality and can be applied in the simplifying step in different ways. For example if we have an expression like $[\diamond_x tt : \alpha x \geq \mu]$ then we can apply rule 5_π first to conclude that $x = 1$ and then simplify the whole expression to tt.

Definition 7.3 (Simplification rules for φ)
Formulas of the type φ can be simplified using the following rules.

$$\begin{array}{lll}
1_\varphi: & \neg tt \mapsto f\!f \text{ and } \neg f\!f \mapsto tt & \\
2_\varphi: & f\!f \wedge \varphi \mapsto f\!f \text{ and } tt \wedge \varphi \mapsto \varphi & \\
3_\varphi: & [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] & \mapsto f\!f \\
& \text{if } \alpha_1 + \cdots + \alpha_n < \mu & \\
4_\varphi: & [\diamond_{x_1} f\!f, \diamond_{x_2} F_2, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] & \mapsto \\
& \quad [\diamond_{x_2} F_2, \ldots, \diamond_{x_n} F_n : \alpha_2 x_2 + \cdots + \alpha_n x_n \geq \mu] & \\
5_\varphi: & [\diamond_{x_1} tt, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] & \mapsto \\
& \quad [\diamond_{x_2} F_2, \ldots, \diamond_{x_n} F_n : \alpha_2 x_2 + \cdots + \alpha_n x_n \geq \mu - \alpha_1] & \\
6_\varphi: & [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] & \mapsto tt \\
& if\, \mu < 0 &
\end{array}$$

Explanation of The Simplification Rules

The first two rules are simple boolean rules.

Rule three was discussed in chapter 6. If the constants α_i becomes small enough, the whole expression simplifies to false.

$$\begin{array}{ll} 3_\varphi : & [\diamond_{x_1}F_1, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] \;\; \mapsto \mathit{ff} \\ & \text{if } \alpha_1 + \cdots + \alpha_n < \mu \end{array}$$

The fourth rule states that if $\diamond_{x_1}\mathit{ff} : \alpha_1 x_1 \geq \mu$, then it can be concluded that x_1 is zero, which removes both the variable and the constant from the inequality.

$$\begin{array}{ll} 4_\varphi : & [\diamond_{x_1}\mathit{ff}, \diamond_{x_2}F_2, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n] \;\; \mapsto \\ & \quad [\diamond_{x_2}F_2, \ldots, \diamond_{x_n}F_n : 0 + \alpha_2 x_2 + \cdots + \alpha_n x_n] \end{array}$$

Rule 5 states that if $\diamond_{x_1}tt : \alpha_1 x_1 \geq \mu$ then the probability variable x_1 is equal to one, and thereby disappears from the inequality.

$$\begin{array}{ll} 5_\varphi : & [\diamond_{x_1}tt, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n] \qquad\qquad \mapsto \\ & \quad [\diamond_{x_2}F_2, \ldots, \diamond_{x_n}F_n : \alpha_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n] \end{array}$$

If we can simplify any of the x_i's to 1, we subtract the corresponding α_i from μ. When doing this, we can reach a situation where μ becomes negative, thereby causing the inequality to be trivially satisfied.

$$\begin{array}{ll} 6_\varphi : & [\diamond_{x_1}F_1, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu] \;\; \mapsto tt \\ & if \mu < 0 \end{array}$$

Some of the rules form part of the complete axiomatization of validity for the logic offered by Larsen and Skou in [LS92].

Theorem 7.4
The simplification rules in definition 7.3 are sound.

Proof
We prove the theorem by showing that the semantics for the original formula are the same as for the simplified formula, for all cases in definition 7.3.

The first two are quite simple and standard rules, and we will only prove the correctness of $3_\varphi \;-\; 6_\varphi$

3_φ : Suppose that $\alpha_1 + \cdots + \alpha_n < \mu$. As $\alpha_i \leq 1$ for all i, also $\alpha_1 x_1 + \cdots + \alpha_n x_n < \mu$.

4_φ : This rule is proved in same way as 5_φ, so proof is omitted here.

5_φ : To prove correctness of this, we have to show that

$$\{\!|[\diamond_{x_1}tt, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]|\!\} =$$

$$\{\!|[\diamond_{x_2}F_2, \ldots, \diamond_{x_n}F_n : [\alpha_2 x_2 + \cdots + \alpha_n x_n \geq \mu - 1]|\!\}.$$

We start on the left side of the equation, writing down the semantics, and minimizing:

$$\begin{aligned}
&\{\!|[\diamond_{x_1}tt, \ldots, \diamond_{x_n}F_n : \alpha_1 x_1 + \cdots + \alpha_n x_n \geq \mu]|\!\} = \\
&\{\pi \mid \alpha_1 \pi[\![tt]\!] + \cdots + \alpha_n \pi[\![F_n]\!] \geq \mu\} = \\
&\{\pi \mid \alpha_2 \pi[\![F_2]\!] + \cdots + \alpha_n \pi[\![F_n]\!] \geq \mu - 1\} = \\
&\{\!|[\diamond_{x_2}F_2, \ldots, \diamond_{x_n} : \alpha_2 x_2 + \cdots + \alpha_n x_n \geq \mu - 1]|\!\}
\end{aligned}$$

Since the semantics of the right side is equal to the reduced semantics of the left side, this rule is sound.

6_φ : This proof is trivial since:

$$\alpha_i \geq 0 \text{ and } x_i \geq 0.$$

This concludes the proof of soundness of the rules in definition 7.3 □

7.5 Implementation

We have implemented the simplification rules defined in this chapter as a single function `Simplify`, which is to be called after each call to the `evalQuotient` function. The function has a mutually recursive function called `nonSimp` and `probSimp`, this sub-function is defined on all the possible combination of formula instances.

The function runs through the (quotiented) formula and simplifies it in respect to the simplifying rules, and returns a simplified formula.

```
fun Simplify Form c =
    let fun nonSimp (nTrue) = nTrue
          | nonSimp (nFalse) = nFalse
          | nonSimp (nAp (s,i)) = let val acstate =
                                              getactivestate c i
                                  in if acstate = s
                                     then nTrue
                                     else nFalse
                                  end
          | nonSimp (nNot F) = let val SF = nonSimp F
                               in if (SF = nTrue)
                                  then nFalse
                                  else if (SF = nFalse)
                                  then nTrue
                                  else nNot SF
                               end
```

Above is the code for the simple expression, true and false reduces, not surprisingly, to true and false. `nAp` checks for the current state of the system, and `nNot` calls the simplification formula recursively to get the simplified expression for `F`. If `F` is either true or false, it simply returns the opposite, and in any other case `F`, it returns `nNot F`.

The case of `nAnd` and `nOr` is handled by recursively calling the simplification function on the left and right sides of the operator. It checks if any of the sides reduces to true or false, and if this is the case, it makes the defined simplification, if not, it returns the (still with simplified left and right parts) expressions with the corresponding operator.

```
      | nonSimp (L nAnd R) = let val LS = nonSimp L
                                 val RS = nonSimp R
                             in if (LS = nFalse)
                                 orelse (RS = nFalse)
                                then nFalse
                                else if (LS = nTrue)
                                then RS
                                else if (RS = nTrue)
                                then LS
                                else LS nAnd RS
                             end
      | nonSimp (L nOr R) = let val LS = nonSimp L
                                val RS = nonSimp R
                            in if (LS = nTrue)
                                orelse (RS = nTrue)
                               then nTrue
                               else if (LS = nFalse)
                               then RS
                               else if (RS = nFalse)
                               then LS
                               else
                                   LS nOr RS
                            end
```

The diamond modality for process transitions is quite straightforward, as the only rule for simplification of it is $\langle a \rangle ff \mapsto ff$ The function uses the function `probSimp` on the probabilistic formula `pform`, and if that reduces to false, then the function returns `nFalse`. In any other case it returns formula `nDiamond(a,pform)`, where `pform` is simplified.

```
      | nonSimp (nDiamond (a,pform)) =
                            let val PS = probSimp pform
                            in if (PS = pFalse)
                                   then nFalse
                               else nDiamond (a, PS)
                            end
```

The simple formula types as `pNot`, `pAnd` and `pOr`, are implemented in a similar way as their process counterparts, and are therefore omitted here.

The simplification function for the simple probabilistic modality, $\diamond_{\geq \mu}$ checks if the value of the probability μ is valid $(0 \leq \mu \leq 1)$ and returns `pFalse` if that is not the case.

Then it uses the process simplification formula `nonSimp`, and checks if it either is or simplifies to false, and if so returns `pFalse`.

```
            | probSimp (pDiamondsimp(p,nform)) =
              if (p>1.0) orelse (p<0.0) then pFalse
              else
                  let val NS = nonSimp nform
                  in if (NS = nFalse)
                     then pFalse
                     else pDiamondsimp(p,NS)
                  end
```

Finally we have the simplification function for the general modality. This is the most complex of them all, as it needs to check a lot of cases, and because of the number of rules for this type.

```
            | probSimp (pDiamond(L,mu)) =
              let fun DiamondInner (nil,mu) = (nil,mu)
                    | DiamondInner (((p,nform)::T),mu) =
                      case (nonSimp nform) of
                           (nTrue) => DiamondInner (T,(mu-p))
                         | (nFalse) => DiamondInner (T,mu)
                         | (nform') =>
                                let val (T',mu') = DiamondInner(T,mu)
                                in (((p,nform')::T'),mu')
                                end
              in case DiamondInner((L),mu) of
                      (nil,mu') =>
                           if (0.0>mu') then pTrue else pFalse
                    | ([(p,nform)],mu') =>
                              if ((mu'/p)>1.0)
                              then pFalse
                              else
                                  if ((mu'/p)<0.0)
                                  then pTrue
                                  else (probSimp
                                        (pDiamondsimp((mu'/p),nform)))

                    | (L',mu') => if (mu'<0.0) then pTrue
                                  else
                                     let val problist = getprob (L')
                                         val max = findmax problist
                                         val amount = count problist
                                     in if (max<(mu'/amount))
                                             then pFalse
                                        else (pDiamond(L',mu'))
                                     end
              end
      in nonSimp Form
      end;
```

In the case of `pDiamond` the function `probSimp` defines an inner function called

`DiamondInner`. This function handles different cases of process formulas, and uses rules 4_φ and 5_φ to simplify expressions where a process formula simplifies to true or false.

`DiamondInner` is then checked in different cases, the simple `nil`, the case of only one inner formula and variable (which can then either be simplified or described by `pDiamondsimp`), and the case of a full general modality. In the latter case the function uses three basic sub-functions to check for rule 3_φ. Rule 6_φ is checked in every different case in the function, and if no more simplification can be done, the function returns a general modality.

Chapter 8

The Model Checker

Throughout the report we have described various bits from our implementation of a model checker for probabilistic alternating transition systems. We have presented almost everything from definition of datatypes to the quotient technique and corresponding simplification rules. In this chapter we complete the implementation by describing the last functions in our model checker.

We have chosen not to implement any graphical user interface or other beautification features, as we are solely interested in the results and performance of the model checker.

8.1 The Functions

The functions we need to describe are first of all a function which, given a parallel system, chooses a machine to factor out, and calls the quotient and simplification functions. This function called `chooseIndexAndFactorOut`, needs a helping function `divideMachine`, which basicly divides the system into different cases.

The function `chooseIndexAndFactorOut` takes as input a full parallel system, a formula and a synchronizing set. It chooses a machine to factor out (the first machine of the system), calls the quotient and simplification formulas continuously until one machine is left. It then checks if this machine satisfied the quotiented formula or not, and returns true or false.

```
fun divideMachine (M as system (_,_,_,index)) selectedIndex =
    if index = selectedIndex
    then (SOME M, NONE)
    else (NONE, SOME M)
  | divideMachine (M1 || M2) selectedIndex =
    let val (selectM1,restM1) = divideMachine M1 selectedIndex
        val (selectM2,restM2) = divideMachine M2 selectedIndex
    in (case (selectM1, selectM2) of
              (SOME M1', NONE) => SOME M1'
            | (NONE, SOME M2') => SOME M2'
            | (NONE,NONE)      => NONE
        , case (restM1, restM2) of
               (SOME M1', NONE)       => SOME M1'
             | (NONE, SOME M2')       => SOME M2'
             | (SOME M1', SOME M2')   => SOME (M1' || M2')
             | (NONE, NONE)           => NONE
                                      )
    end
```

```
fun chooseIndexAndFactorOut subMachine currentFormula A =
    let val selectedIndex = indexOfMachine (first subMachine)
    in let val (SOME selectedMachine, restOpt) =
                divideMachine subMachine selectedIndex
       in let val nextFormula =
              Simplify(evalQuotient
                        selectedMachine
                        currentFormula A)(initialconf selectedMachine)
          in case restOpt of
                 SOME (system (ntrans,ptrans,s0,i)) =>
                      nSatisfy
                      (system (ntrans,ptrans,s0,i)) nextFormula A
               | SOME restM =>
                      chooseIndexAndFactorOut restM nextFormula A
          end
       end
    end
```

8.2 Effectiveness of the Model Checker

As described in section 5.4 and 7.3 we might obtain some advantage in creating an algorithm which checks for specific machines to factor out first. As the model checker is now, it factors out machines from one end, not concerned about the description of this machine.

It could add to the effectiveness of the implemented model checker to do such a check, but we have not found it needful in our implementation. This is mainly because that, as our systems and formulas are defined, we believe that we would not gain a lot by adding this check.

If future implementation would include some kind of recursiveness of formulas, and perhaps infinite transition relations, then this check would be much more useful.

In the next chapter we give a large example of our implementation, in order to test the quotient technique against the simple checker (ie. by running through all states).

Chapter 9

Testing the Model Checker

In this chapter we run our implementation on some examples in order to measure performance of the quotient model checker compared to the simple checker.

9.1 A Telephone Call

As an example of systems which can be described using probabilistic system, we assume cell-phone communication system. When making a call from one cell-phone to another, the signal is transmitted via air to a transmitting station, and again via air to the receiver of the call. Obviously it cannot be guaranteed that no errors will occur, and the more users, the bigger chance of losing the connection.

We will try to model a call from a cell phone, by using our probabilistic alternating transition systems. We will then compose this call with other calls, and give a specification for this composed system.

We start by creating a pure synchronous system (with more than half of the population in large cities owning a cell-phone, the chance of at least a few people using their cell-phones simultaneously should be quite large). We will then check if the composed formula satisfies the specification, by using our model checker, and compare the computing time with the time for checking the same system with the simple checker.

In figure 9.1 we show a graphical view of our the intuition we have about a call from a cell phone. Note that this is a theoretical example, and may not have anything to do with how the real life GSM system works.

As we see in the figure, the caller might be placed in between two transmitting

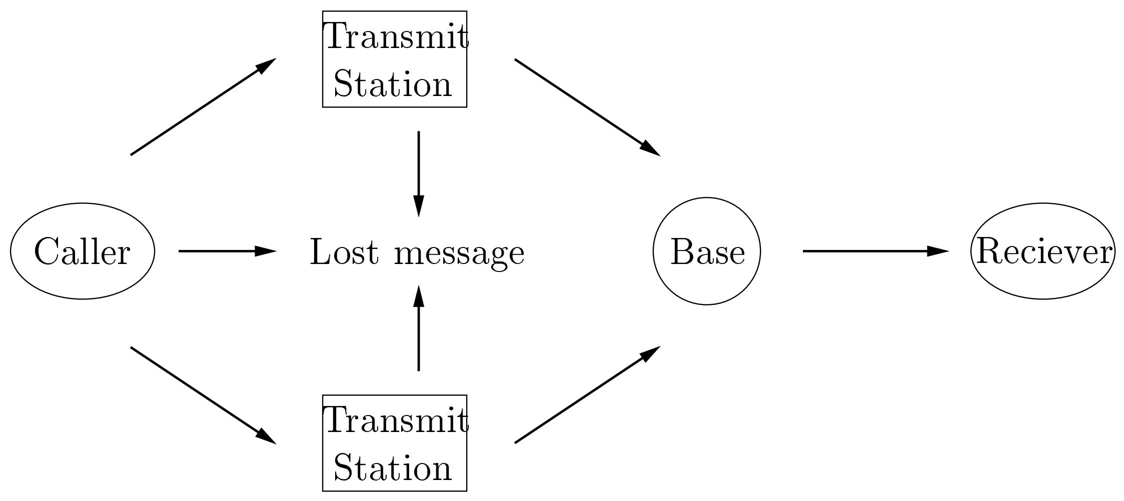

Figure 9.1: A graphical vies of a call from a cell phone

stations, which gives us the choice of two different stations, which we define to act similar to each other. We will therefore assume in our example that we only have one transmitting station. We define our transition system as seen in figure 9.2.

As we see there is 1% chance that the call fails even before it reaches the transmitting station, and again 1% chance that the call won't be finished.

9.2 Test #1

As mentioned before, we start by composing our calls using pure synchronizing transition. That means that our synchronizing set is $\mathcal{A} = \{call, connect, error, complete\}$.

We know that when having synchronous transitions, then the probability connected to that transition will become smaller when composing the system. In this example, this means that if more than one person tries to make a call at the same time, the probability for failure becomes larger.

Initially there is only 1% chance of failure, so what is acceptable when there are, say, 20 people making calls at the same time. Well if you ask a cell-phone user he would probably answer no loss at all. Fortunately we do not have to ask anyone, so we set the allowed loss rate to 5%.

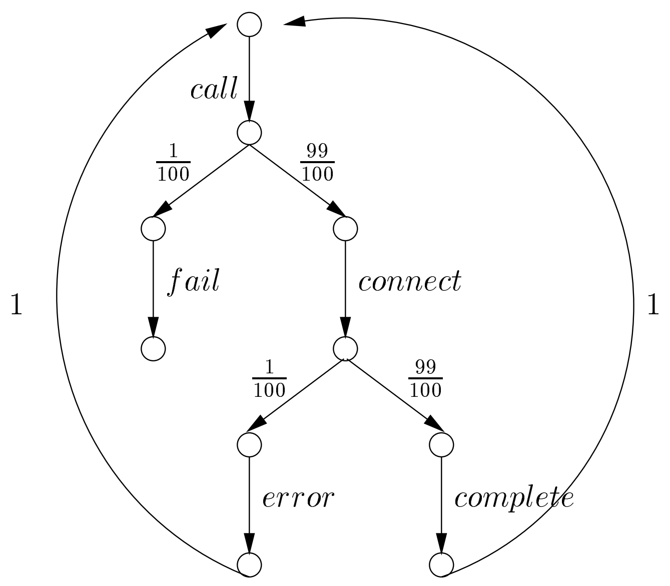

Figure 9.2: The transition system for a phone call

This gives us the following specification for our system:

$$\langle call\rangle \diamond_{\geq \frac{95}{100}} \langle connect\rangle \diamond_{\geq \frac{95}{100}} \langle complete\rangle tt$$

For testing we name the different machines `call1`, `call2` and so on. The composed system is named `Call`, the specification `Callformula` and the synchronizing set is named `CallA`.

9.2.1 Verification

We verify the correctness of the quotient technique by running an example with two machines in parallel. A graphical view of the composed system is seen in figure 9.3.

First we use the simple model checker:

```
- Satisfy (call1 || call2) Callformula CallA;
> val it = true : bool
```

Not surprisingly it returns true, which we can verify by looking at figure 9.3.

We now run the same example with the quotient technique, and get

```
- chooseIndexAndFactorOut (call1 || call2) Callformula CallA;
> val it = true : bool
```

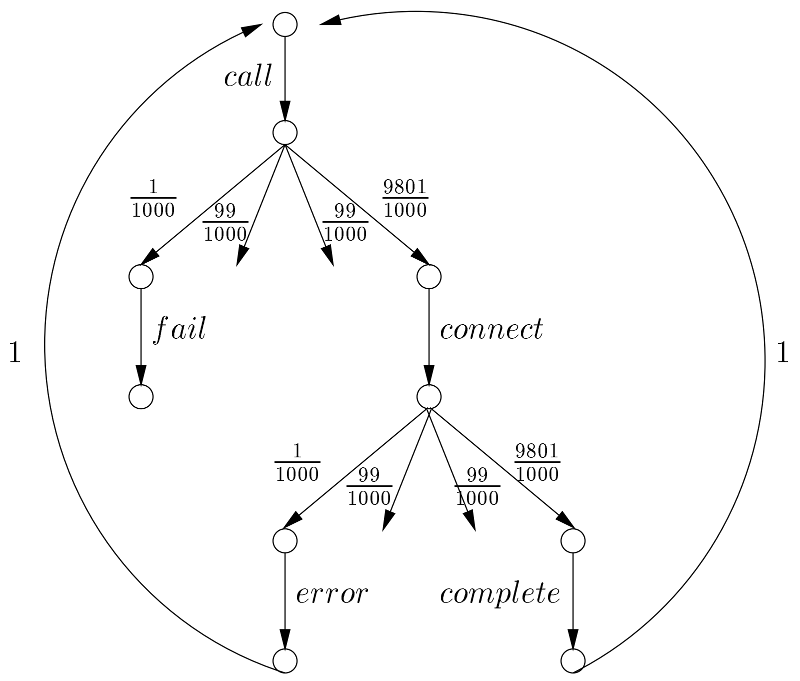

Figure 9.3: The composed system

So we have verified that the two techniques returns the same result.

9.2.2 Test Results

We will now compare the two checkers, to see which one is fastest. By running some tests, we find out that our specification does not hold when we have more than 5 machines in parallel. This is obviously not a good sign if we were to implement a cell-phone communication system, but for testing our technique it should be fine.

The Test setting

For our test setting we use the above specifications of our system and formula, and we run the tests on an AMD Duron 600 MHz processor with Windows ME. The functions are called from within emacs, using MosML.

The Test

We test our example by starting with two machines in parallel, and increasing this number until we have a system consisting of 20 machines in parallel. We plot the times in a diagram, which can be found in figure 9.4.

As we can see in the figure, this example really shows the benefits of the quotient technique. When verifying 20 parallel machines, the technique concludes quite early that the system doesn't satisfy the specification, and thereby reducing to false.

The simple checker does not have this check, and therefore still checks all states of the parallel system. In this example though, we can only check up to 17 machines in parallel with the simple checker, above that the computer runs out of memory.

9.3 Test #2

For the second test we introduce a new machine, which has two ways of connecting a call. We will use one instance of this machine in our composed system, and give a new specification for the system. The new machine can be seen in figure 9.5.

The specification we wish to test in this example is the following:

$$\langle call \rangle \diamond_{\geq \frac{1}{2}} ((\langle connect \rangle \diamond_{\geq \frac{2}{5}} \langle complete \rangle tt) \vee (\langle connect2 \rangle \diamond_{\geq \frac{9}{10}} \langle complete \rangle tt))$$

We test this in similar way as test #1, and plot the result into a diagram, seen in figure 9.6.

The results for this test is also very satisfying. The simple checker grows exponentially in time, whereas the computation time for the quotient technique grows more moderately. Actually we see that in this case the simple checker performs even worse than before, while the quotient technique is much the same as in test #1.

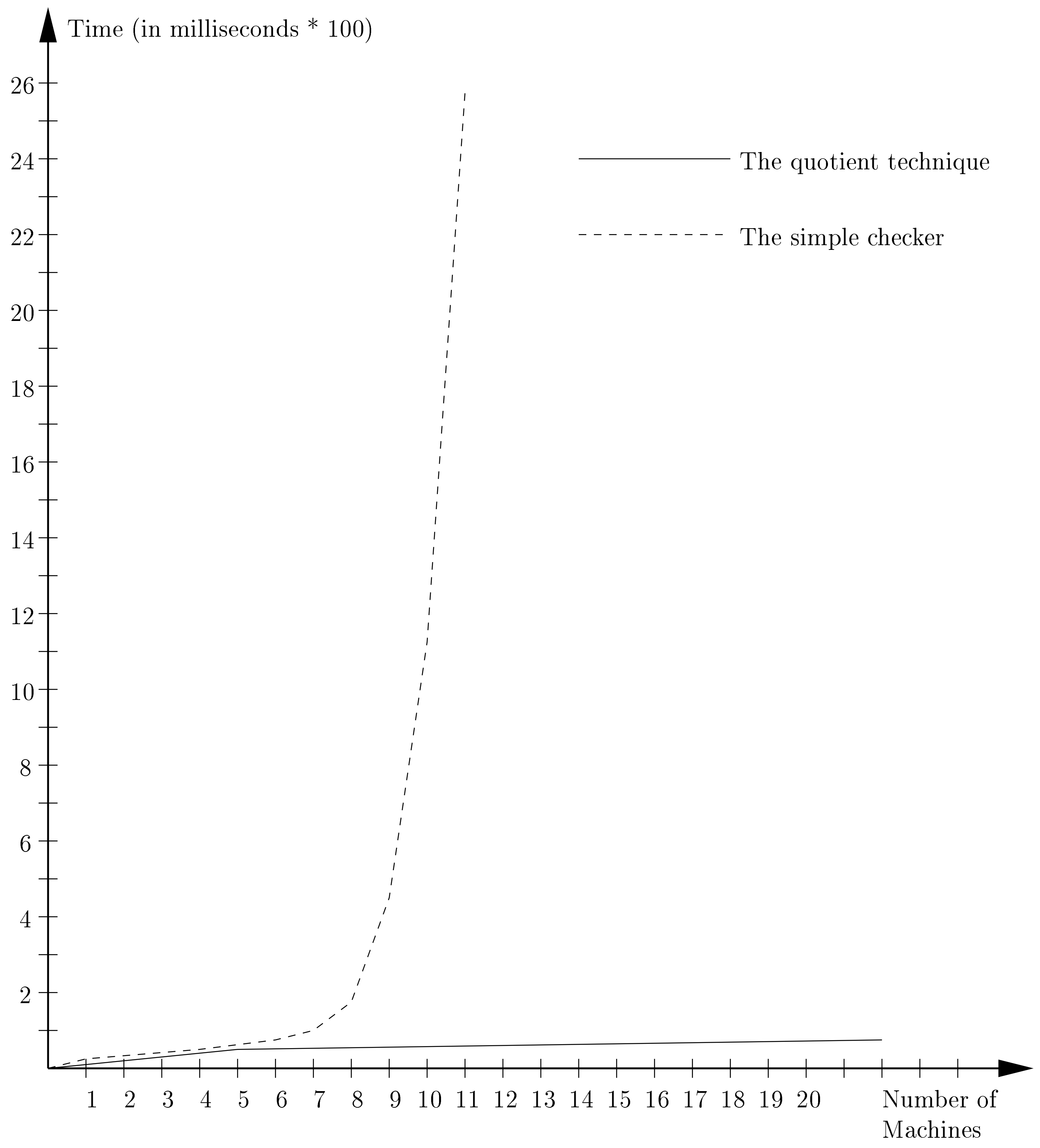

Figure 9.4: Execution time for the simple checker and the quotient technique.

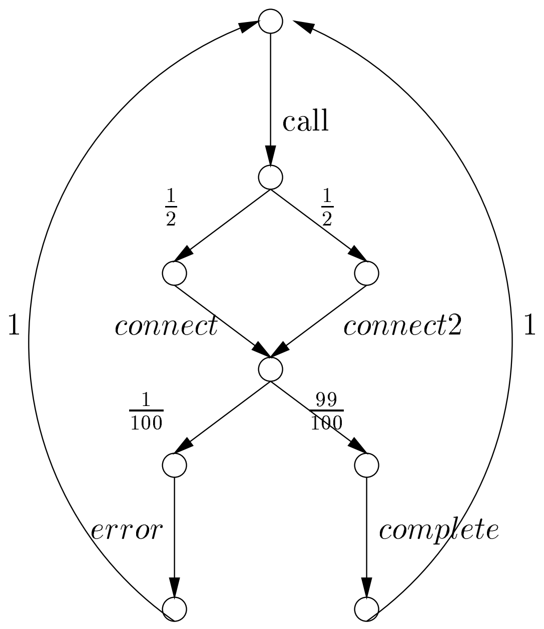

Figure 9.5: The new machine

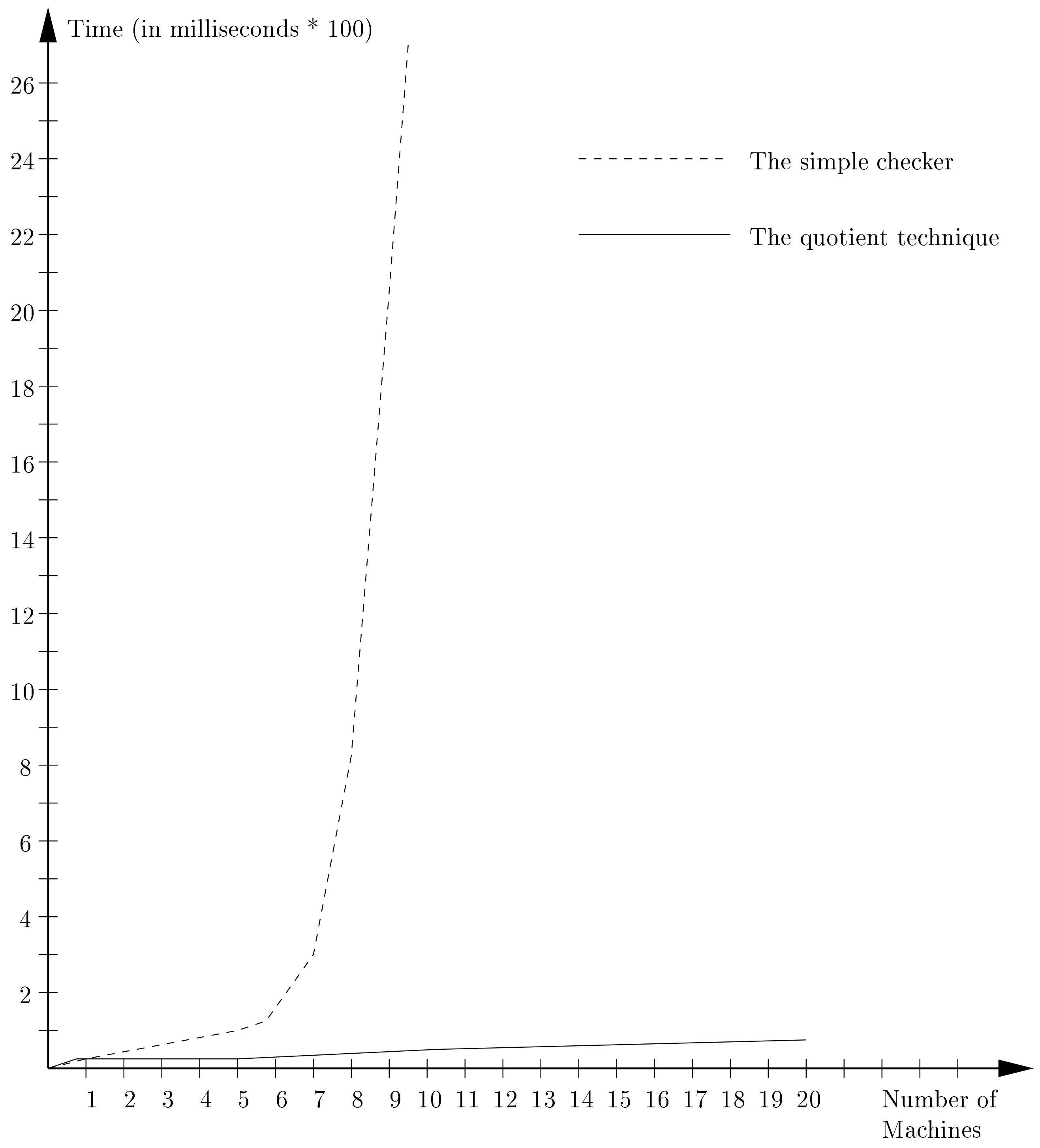

Figure 9.6: Time diagram for Test #2

Chapter 10

Conclusion

The main goal of this report was to develop further the theory introduced in [Ves00], and implement the technique, which this theory offers. Especially the general modality needed to be explored, but also asynchronous composition and simplification were to be studied more thoroughly.

10.1 Probabilistic Transition Systems

We have explored the theory of probabilistic alternating transition systems in more detail than we did in the above mentioned reference. Some basic probability theory has been introduced, in order to get a better understanding of the behavior of the systems. We have defined a probabilistic process calculus, and a probabilistic modal logic (PML). As we concentrate on the quotient technique, we have focused on the transition systems rather than the calculus. We have defined an asynchronous parallel operator, by introducing a π_P transition to each P, in order to be able to keep the two part syntax.

10.2 The Quotient Technique

We have defined the quotient technique for our transition systems and our logic, and shown that the original logic was not strong enough to support the technique. This is not a revolutionary result, as it was shown by Larsen and Skou in [LS92]. However we have defined the general modality in a more specific version than the one of Larsen and Skou's. We have also shown that the last part of the modality is indeed *one* linear inequality, and have furthermore given a quotient definition of this modality.

10.3 The General Modality

The question of how the general modality would behave when being part of a quotient formula, has been answered. There will never be more than one inequality concerning the instantiation of each variable, only the number of binding variables will change. As the number of variables in the inequality raise when continuously factoring out components of a system, we have explored some ways of reducing it.

Because of the fact that we most of the times are able to simplify the binding variables, thereby causing some of the constants in the inequality to be removed, we found that we can actually show that there is a big chance that the whole modality can be reduced to false after we have factored out some machines. This and the discovery that the right side of the inequality sometimes becomes negative and thereby causing the modality to reduce to true, gives a good hope for the applicability of the quotient technique.

10.4 Simplification

We have explored new ways of simplifying the quotiented formulas. For example have we discussed that making a check on the composed system, in order to choose a specific machine could be a promising way of obtaining formulas which easily simplifies. This can both be used in the case of asynchronous and synchronous composition of systems.

We discussed two of the most important simplification rules for probabilistic formulas above. Besides those two, we still have some basic rules, which help us in simplifying probabilistic formulas.

10.5 The Implementation

We have implemented our theory in the programming language Moscow ML. This has resulted in a model checker which uses the quotient technique to verify satisfiability of probabilistic alternating transition systems. We have also implemented a simple checker, which runs through all states in order to check the specification.

In our tests, we can clearly see the problem of state explosion, when using the simple checker. The quotient technique does exceptionally well, which was actually the intuition we had from the start. It should be clear by now, that if we can simplify the quotiented formulas quite early, then the quotient technique spends very little time verifying even large systems. It seems that the simplification rules we have implemented are quite effective, as our model checker is very fast, at least in our test examples.

10.6 Further Work

Although we have explored and discussed some of the the important areas in applying the quotient technique to probabilistic systems, there are still lots of interesting things to be explored.

Bisimulation is one interesting area to study. Another possible extension would be to add recursive properties to the logic, which again would call for more theory about simplification, by for example studying some fixed point theory.

Furthermore there is the question of adding time to probabilistic systems (or vice versa). This could enable us to express things like "certain events occurs with probability x within z seconds".

Bibliography

[And95] Henrik Reif Andersen. Partial model checking. In *LICS95*, IEEE Computer Society Press, 1995.

[BCGH$^+$97] C. Baier, E. Clarke, V. Garmhausen-Hartonas, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP'97*, number 1256 in LNCS 1256, pages 430–440. Springer, 1997.

[BLL$^+$95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, October 1995.

[BM99] Marius Bozga and Oded Maler. On the representation of probabilities over structured domains. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 261–273. Springer, 1999.

[BP00] Nicky O. Bodentien and Lasse O. Poulsen. The quotient verification technique applies to state/event systems. Master's thesis, Aalborg University, June 2000.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, pages 677–691, 1986.

[DeG89] Morris H. DeGroot. *Probability and statistics.* Addison Wesley, 1989.

[Han94] Hans A. Hansson. *Time and Probability in Formal Design of Distributed Systems*, volume 1. Elsevier, 1994.

[HJ89] Hans Hansson and Bengt Jonsson. A framework for reasoning about time and reliability. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, Santa Monica CA, December 1989.

[JS90] Chi-Chang Jou and Scott A. Smolka. Equivalences, congruences, and complete axiomatizations for probabilistic processes. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 367–383. Springer-Verlag, 1990.

[Kri98] Kaare Jelling Kristoffersen. *Compositional Verification of Concurrent Systems.* PhD thesis, Aalborg University, 1998.

[kro] Kronos Home Page.
URL: *http://www-verimag.imag.fr/TEMPORISE/kronos/*.

[Lar86] Kim G. Larsen. *Context-Dependent Bisimulation Between Processes.* PhD thesis, University of Edinburgh, 1986.

[LL95] François Laroussinie and Kim G. Larsen. Compositional model checking of real time systems. BRICS Report series RS-95-19, BRICS, March 1995.

[LS91] Kim Larsen and Arne Skou. Bisimulation through probabilistic testing. In *Information and Computation*, volume 94. September 1991.

[LS92] Kim G. Larsen and Arne Skou. Compositional verification of probabilistic processes. 1992.

[NJJ$^+$99] N.O.Bodentien, J.Vestergaard, J.Friis, K.J.Kristoffersen, and K.G.Larsen. Verification of state-event systems with acyclic dependencies by quotienting. 1999. Presented at NWPT'99.

[rHA97] Jørgen H. Andersen. *Compositional Modal Logics.* PhD thesis, Aalborg University, 1997.

[Seg95] Roberto Segala. *Modelling and Verification of Randomized Real-Time Systems.* PhD thesis, Massachusetts Institute of Technology, 1995.

[Ves00] Jacob Vestergaard. Quotienting probabilistic transition systems. Technical report, Aalborg University, January 2000.

[vGSST90] R.J. van Glabbeek, S.A. Smolka, B. Steffen, and C.M.N. Tofts. Reactive, generative and stratified models of probabilistic processes. In *Proceedings 5th Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1990.

Appendix A

The Quotient Technique for Finite State Systems

Theorem A.1
Given two processes P_1 and P_2 and two distributions π_1 and π_2 in CPS, then

$$P_1 \| P_2 \models F \Longleftrightarrow P_1 \models F/P_2$$

$$\pi_1 \| \pi_2 \models \varphi \Longleftrightarrow \pi_1 \models \varphi/\pi_2$$

Proof
We proof theorem A.1 by induction on the structure F respectively π.

(i) $F = tt$ is trivial

(ii) $F = F_1 \wedge F_2$:

$$\begin{array}{ll} & P_1 \| P_2 \models F_1 \wedge F_2 \\ \Leftrightarrow & P_1 \| P_2 \models F_1 \text{ and } P_1 \| P_2 \models F_2 \\ \overset{\text{IH}}{\Leftrightarrow} & P_1 \models F_1/P_2 \text{ and } P_1 \models F_2/P_2 \\ \Leftrightarrow & P_1 \models F_1/P_2 \wedge F_2/P_2 \\ \Leftrightarrow & P_1 \models (F_1 \wedge F_2)/P_2 \end{array}$$

(iii) The negation is trivial and thus not proved

(iv) $F = \langle a \rangle \varphi$:This case is divided in two parts, the synchronous and the asynchronous. First the synchronous case:

$$\begin{array}{ll}
 & P_1 \| P_2 \models \langle a \rangle \varphi \\
\Leftrightarrow & \exists j,k. P_1 \| P_2 \xrightarrow{a} \pi_j \| \pi_k \wedge \pi_j \| \pi_k \models \varphi \\
\overset{\text{IH}}{\Leftrightarrow} & \exists j,k. P_1 \xrightarrow{a} \pi_j \wedge P_2 \xrightarrow{a} \pi_k \wedge \pi_j \models \varphi/\pi_k \\
\Leftrightarrow & \exists j.(P_1 \models \langle a \rangle \varphi/\pi_k) \wedge P_2 \xrightarrow{a} \pi_k \\
\Leftrightarrow & P_1 \models \bigvee_{k. P_1 \xrightarrow{a} \pi_k} \langle a \rangle \varphi/\pi_k \\
\Leftrightarrow & P_1 \models \langle a \rangle \bigvee_{P_2 \xrightarrow{a} \pi_2} \varphi/\pi_2 \\
\Leftrightarrow & P_1 \models (\langle a \rangle \varphi)/P_2
\end{array}$$

Then the asynchronous case:

$$\begin{array}{ll}
 & P_1 |_A P_2 \models \langle a \rangle \varphi \\
\Leftrightarrow & \exists \pi_1, \pi_2.(P_1 \xrightarrow{a} \pi_1 \wedge \pi_1 |_A P_2 \models \varphi) \vee (P_1 \xrightarrow{a} \pi_1 \wedge P_2 \xrightarrow{a} \pi_2 \wedge \pi_1 |_A \pi_2 \models \varphi) \\
\Leftrightarrow & \exists \pi_2.(P_1 \models \langle a \rangle \varphi/P_2) \vee (P_1 \models \langle a \rangle \varphi/\pi_2 \wedge P_2 \xrightarrow{a} \pi_2) \\
\Leftrightarrow & P_1 \models \langle a \rangle \varphi/P_2 \vee \bigvee_{P_2 \xrightarrow{a} \pi_2} \varphi/\pi_2
\end{array}$$

(v) $\varphi = tt$ is trivial

(vi) $\varphi = \varphi_1 \wedge \varphi_2$:

$$\begin{array}{ll}
 & \pi_1 \| \pi_2 \models \varphi \wedge \varphi \\
\Leftrightarrow & \pi_1 \| \pi_2 \models \varphi_1 \text{ and } \pi_1 \| \pi_2 \models \varphi_2 \\
\overset{\text{IH}}{\Leftrightarrow} & \pi_1 \models \varphi_1/\pi_2 \text{ and } \pi_1 \models \varphi_2/\pi_2 \\
\Leftrightarrow & \pi_1 \models \varphi_1/\pi_2 \wedge \varphi_2/\pi_2 \\
\Leftrightarrow & \pi_1 \models (\varphi_1 \wedge \varphi_2)/\pi_2
\end{array}$$

(vii) Negation is trivial

(viii) $\varphi = \diamond_{\geq \mu} F$:

$$\begin{array}{ll}
 & \pi_1 \| \pi_2 \models \diamond_{\geq \mu} F \\
\Leftrightarrow & \displaystyle\sum_{Q \| P \models F} \pi_1(Q) \cdot \pi_2(P) \geq \mu \\
\overset{\text{IH}}{\Leftrightarrow} & \displaystyle\sum_{P} \sum_{Q \models F/P} \pi_1(Q) \cdot \pi_2(P) \geq \mu \\
\Leftrightarrow & \pi_1 \models [\diamond_{x_1}(F/P_1), \ldots, \diamond_{x_n}(F/P_n) : \mu_1 \cdot x_1 + \cdots + \mu_n \cdot x_n \geq \mu] \\
 & \text{where } \pi_2 \rightsquigarrow_{\mu_1} P_1, \ldots, \pi_2 \rightsquigarrow_{\mu_n} P_n \\
\Leftrightarrow & \pi_1 \models (\diamond_{\geq \mu} F)/\pi_2
\end{array}$$

(ix) $\varphi = [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \Phi(x_1, \ldots, x_n)]$:

$$\begin{aligned}
& \pi_1 \| \pi_2 \models [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \Phi(x_1, \ldots, x_n)] \\
\Leftrightarrow\ & \Phi(\nu_1, \ldots, \nu_n) = True, \text{ where } \nu_i = \textstyle\sum_{P, Q_j . P|Q_j \models F_i} \pi_1(P) \cdot \pi_2(Q_j) \\
\Leftrightarrow\ & \nu_i = \textstyle\sum_{Q_j} \underbrace{\pi_2(Q_j)}_{\mu_j} \cdot \overbrace{\displaystyle\sum_{P.P \models F_i/Q_j} \pi_1(P)}^{y_{ij}} \\
\Leftrightarrow\ & \pi_1 \models [\diamond_{y_{11}} F_1/Q_1, \ldots, \diamond_{y_{ij}} F_i/Q_j, \ldots, \diamond_{y_{nk}} F_n/Q_k : \Phi(\textstyle\sum_j \mu_j y_{1j}, \ldots, \sum_j \mu_j y_{nj})] \\
\Leftrightarrow\ & \pi_1 = [\diamond_{x_1} F_1, \ldots, \diamond_{x_n} F_n : \Phi(x_1, \ldots, x_n)]/\pi_2
\end{aligned}$$

$\square$